Ke Liu's NS2 Code (Copyright reserved) and Q & A

I graduated and no longer maintain this webpage, no longer working on any ns2 or research. Sorry can not help you on ns2 project any more. Good Luck!

Table of Content

Content

- **Suggestion of Using the NS2 Scheduler:**

  Please use the Heap Scheduler at any possible time, add

  ```
  $ns_ use-scheduler Heap
  ```

  after you instanced a new simulater

  ```
  set ns_ [new Simulator]
  ```

  Heap scheduler does not change your simulation but consumes a shorter simulating time.

- **How to change the Phy/Wireless Transmission Power w.r.t Trans Range?**

  The assumption here is that the receiving threshold is a Hardware feature which can not be changed. So you can use the below tcl script

  ```
  # Initialize the SharedMedia interface with parameters to make
  # it work like the 914MHz Lucent WaveLAN DSSS radio interface
  set RxT_ 3.652e-10 ;#Receiving Threshold which mostly is a hardware feature
  set Frequency_  914e+6 ;# Signal Frequency which is also hardware feature

  Phy/WirelessPhy set CPThresh_ 10.0
  Phy/WirelessPhy set CSThresh_ 1.559e-11
  Phy/WirelessPhy set RXThresh_ $RxT_ ;# Receiving Threshold
  Phy/WirelessPhy set Rb_ 2*1e6            ;# Bandwidth
  Phy/WirelessPhy set freq_ $Frequency_
  Phy/WirelessPhy set L_ 1.0

  set opt(Pt)             61.0 ;# Transmission Power/Range in meters
  ```

```
if { $opt(prop) == "Propagation/TwoRayGround" } {
    set SL_ 300000000.0 ;# Speed of Light
    set lambda [expr $SL_/$Frequency_]   ;# wavelength
    set lambda_2 [expr $lambda*$lambda]  ;# lambda^2
    set CoD_ [expr 4.0*$PI*$opt(AnH)*$opt(AnH)/$lambda] ;# Cross Over Distance

    if { $opt(Pt) <= $CoD_ } {;#Free Space for short distance communication
        set temp [expr 4.0*$PI*$opt(Pt)]
        set TP_ [expr $RxT_*$temp*$temp/$lambda_2]
        Phy/WirelessPhy set Pt_ $TP_ ;#Set the Transmissiont Power w.r.t Distance
    } else { ;# TwoRayGround for communicating with far nodes
        set d4 [expr $opt(Pt)*$opt(Pt)*$opt(Pt)*$opt(Pt)]
        set hr2ht2 [expr $opt(AnH)*$opt(AnH)*$opt(AnH)*$opt(AnH)]
        set TP_  [expr $d4*$RxT_/$hr2ht2]
        Phy/WirelessPhy set Pt_ $TP_  ;#Set the Transmissiont Power w.r.t Distance
    }
}
```

- **GPSR: Greedy Perimeter Stateless Routing code for version 2.26 or later (till 2.29 now)**

  Copyright reserved, referencing is required for use of any purpose Download my implementation tarball
  You can obtain its original implementation through GPSR official link which is just for old version of ns2.

  Notes:

  - Since this implementation is original based on ns2 version 2.26, you need to be careful to deal with it when you use a later vesion. Especially the ns-packet.tcl file, you need to do the change by yourself. The one in my tarball may not be suitable for you to use.
  - This implementation consists of 2 parts exactly the same as proposed by the original GPSR paper, and is a little bit different from the orginial ns2 implementation (see through the link above).
  - This implementation is a pure *Stateless* implementation, where the routing decision of each packet on any node is made purely only on the local information (one-hop neighborhood)
  - Both *GG* and *RNG* planarization methods are provided by this implementation.
  - Before your re-make the ns2 after you put all my files in the correct directory, DO NOT forget to touch the packet.cc in ns2/common directory. (It is just a common NS2 using problem, not about my code.
  - If you have any question for this code, please Do NOT email me. I graduated. You are on your own.

- **Shortest Path Routing for Sensor Networks (for ns2.27 or later)**

  tarball
  Note:
  - The shortest path routing in sensor networks is nearly the same as DYMO routing proposed recently (around 2000) by IEEE as a new routing standards (for MANET or meshNet). Although we use hop-count here, it is easily to adapt other features of link quality as the path choosing factor.

- **How to analyze your trace file through a perl script**

  Let's run the test tcl script and show how to analyze.
  The trace file is obtained by running ns2 with the script ns2/tcl/ex/wireless.tcl, and you can analyze it with this Perl script

- **About the ARP packet dropping**

  It may be error of ns2 (although, this thing is specified in the NS2 manual as on purpose behavior, but I think it is an error), since if a packet initializes an ARP, it would be held in the Arp Table before the destination address is resolved by ARP but if another packet arrives and initializes another ARP for the same destination, the previous packet would be dropped which must be a mistake apparently.
  See ns/mac/arp.cc in function ARPTable::arpresolve();
  If you face some problems like packet dropped due to ARP, you may fix it

- **Understanding the implementation of IEEE MAC 802.11 standard in NS2**

  Download the document pdf

- **The Generic implementation of Multiple (a lot) timers**

  Usually, for one protocol implementation, we need several timers. According to NS2's implementation, for each timer calling different firing function, one more class need to be declared and defined, which is boring and bothering.
  In fact, we can use the *pointer to function* to same time and coding (How to do in with C++ template, I am not sure.). As below:
  - Suppose you have one agent as *TestAgent*, and many firing functions, you can define a pointer to any firing function as

    ```
    typedef void(TestAgent::*firefunction)(void);
    ```

  - Then, you can create a general (generic) Timer class for all timers, as

    ```
    class TestTimer : public TimerHandler{
    public:
            TestTimer(TestAgent *a, firefunction f) : TimerHandler() {
                    a_ = a;
                    firing_ = f;
            }
    private:
            virtual void expire(Event *e);
            TestAgent *a_;
            firefunction firing_;
    };
    ```

  - Ususally, when we create a timer, we need to define the *expire* function to call specific firing function. Now we just need it to call the *Generic* firing funtion pointer, as

    ```
    void TestTimer::expire(Event *){
            (a_->*firing_)();
    }
    ```

  - Finally, when we need a specific timer calling a specific firing funtion, other than what we ususally do as in NS2, we now just need to declare a new timer as an instance of the Generic Timer, and initialize it in TestAgent's constructor, as

- In TestAgent's head file (.h file):

```
class TestAgent : public Agent {
...
        TestTimer timerA_;
        TestTimer timerB_;
...
        void firingA();       // these are 2 different firing functions
        void firingB();
...
};
```

- Then initialization in .cc file as :

```
TestAgent::TestAgent(): Agent(),
                        ...
                        timerA_(this, &TestAgent::firingA),
                        timerB_(this, &TestAgent::firingB),
                        ...
{
...
}
```

  - Even you have 100 timers calling 100 different firing functions in the same agent, you just need what I have shown you here to create **Only ONE** timer class, with only one definition of expire function.
  - Cheers!

- **How to scale your simulation to more nodes (500 nodes) and speed up it?**

The implementation of the Packet data structure of NS2 does not math the realities. The packet in ns2 simulation keeps all packet headers for any protocols implemented in NS2. For example, a DSR routing packet may keep DSDV, AODV, or even a PING application header. For this reason, till today, a packet used in ns2 simulation, would have a header size around 40~64KB. And *NO packet would be deleted* to release the memory it holds until the end of the simulation.
So for a typical simulation with 100 nodes in ns2 around 1M packets exchanged (of course, you may reuse the packets already being freed through Packet::free(Packet*). To learn the implementation of it, please check file *common/packet{.h,.cc}*), you may hold 10% of it, 100K packets, and you may use a memory at least 100K*64KB -> 6.4GB, which definitely would crash your computer (even it is a super server).
So How you can do it? And how you can run a simulation like 500 nodes there? You have 2 ways to do:

  - Suggested by ns2 manual, putting the below codes in your tcl script

```
remove-all-packet-headers
add-packet-header DSR ARP LL MAC CBR IP
```

    to simulation a CBR application on UDP with DSR as routing protocol in a wireless ad hoc network (OOPS!, UDP is not a header. This method is effective, but it requires you to understand most packets header you need.
  - Another way is my way, changing the tcl library for the packet headers. You may find the tcl library file for packet headers in *ns2/tcl/lib/ns-packet.tcl*, you may find a procedure starting as foreach prot {. You can comment out all the headers you don't recognize, like all the *Routing protocols* you dont know, all *Routers*, all *Multicast*, all *Transport Protocols* except you need, all *application layer protocols*, some of the *Wireless*, some of the *Mobility, Ad-Hoc Networks, Sensor Nets* and all the *Other*. Finally, you may just have all the below left

```
foreach prot {
# Common:
        Common
        Flags
        IP
# Transport Protocols
        TCP
# Wireless
        ARP
        LL
        Mac
# Mobility
        AODV
} {
        add-packet-header $prot
}
```

If you are creating your own packet header, put them here.
Typically, after this way, you may just have a packet size as 256B. So even you have 500 nodes, and 10M packets need to be exchanged during the simulation. You just need 256MB (if 10% packets held) for it, which is lower than any common configuration of current PCs.

- **How to interprete the NS2 tracefile for wireless simulation?**

To find the interpretation of all possible trace format when you do the wireless simulation, you'd better read the code of ns2 in file *ns2home/trace/cmu-trace{.h, .cc}* Mostly, the format would be as

```
ACTION: [s|r|D]: s -- sent, r -- received, D -- dropped
WHEN:   the time when the action happened
WHERE:  the node where the action happened
LAYER:  AGT -- application,
        RTR -- routing,
        LL  -- link layer (ARP is done here)
        IFQ -- outgoing packet queue (between link and mac layer)
        MAC -- mac,
        PHY -- physical
flags:
SEQNO:  the sequence number of the packet
TYPE:   the packet type
                cbr -- CBR data stream packet
                DSR -- DSR routing packet (control packet generated by routing)
                RTS -- RTS packet generated by MAC 802.11
                ARP -- link layer ARP packet
SIZE:   the size of packet at current layer, when packet goes down, size increases, goes up size decreases
[a b c d]:      a -- the packet duration in mac layer header
                b -- the mac address of destination
                c -- the mac address of source
                d -- the mac type of the packet body
flags:
[......]:       [
                source node ip : port_number
                destination node ip (-1 means broadcast) : port_number
                ip header ttl
                ip of next hop (0 means node 0 or broadcast)
                ]
```

So we can interpret the below trace

```
s 76.000000000 _98_ AGT  --- 1812 cbr 32 [0 0 0 0] ------- [98:0 0:0 32 0]
```

as Application 0 (port number) on node 98 sent a CBR packet whose ID is 1812 and size is 32 bytes, at time 76.0 second, to application 0 on node 0 with TTL is 32 hops. The next hop is

not decided yet.

And we can also interpret the below trace

```
r 0.010176954 _9_ RTR  --- 1 gpsr 29 [0 ffffffff 8 800] ------- [8:255 -1:255 32 0]
```

in the same way, as The routing agent on node 9 received a GPSR broadcast (mac address 0xff, and ip address is -1, either of them means broadcast) routing packet whose ID is 1 and size is 19 bytes, at time 0.010176954 second, from node 8 (both mac and ip addresses are 8), port 255 (routing agent).

- **Using Eclipse to Enhance your experience of NS2**

  There has been an endless debate of which is the REAL Editor -- vi, or emacs. In case you have any doubt, here is correct answer: Neither. (I have been using Emacs for several years, and still using it. My wife is using vi for a long time. So I understand this debate VERY well.)
  If you want to speed up your programming, and you do not want to waste your time to remember Classes/APIs/Functions, do not want to waste time to open/close/reopen files, switch files backforth, And the most important thing: if you want to understand the code architecture better and quicker, you SHOULD use some good IDE.
  In windows, there is NO doubt, the best IDE is Visual Studio (most of us do not really like Microsoft, but we have to admit, MS is the best software company on earth).
  However, for ns2, the best IDE should be Eclipse -- both Windows and Linux
  - Installation Eclipse with CDT package.
  - Build ns2 through (Windows, you need cygwin. And either for Windows/Linux, don't forget to install the tcl library first).
  - Use the ns2 root path as the workspace path of your Eclipse to launch it.
  - Use Eclipse's C/C++ Perspective. Create a project for your ns2: "New" --> "C++ Project". Select the "Makefile project" --> "Empty project". Give the same ns2 sub directory name as the project name.
  - "Project" --> "Build All". Now you should be able to use Eclipse the Code/Compile /Debug ns2.
  - The most important feature you would get there is the "intellisense". Although Vi/Emacs both provide "Code completion", they are No-comparable -- "intellisense" has much more good features than "Code completion".
