A Tutorial of 802.11 Implementation in ns-2

Yue Wang
MobiTec Lab, CUHK

1. Introduction to ns-2

1.1 ns-2

Ns-2 [1] is a packet-level simulator and essentially a *centric* discrete event scheduler to schedule the events such as packet and timer expiration. Centric event scheduler cannot accurately emulate "events handled at the same time" in real world, that is, events are handled one by one. However, this is not a serious problem in most network simulations, because the events here are often transitory. Beyond the event scheduler, ns-2 implements a variety of network components and protocols. Notably, the wireless extension, derived from CMU Monarch Project [2], has 2 assumptions simplifying the physical world:

(1) Nodes do not move significantly over the length of time they transmit or receive a packet. This assumption holds only for mobile nodes of high-rate and low-speed. Consider a node with the sending rate of 10Kbps and moving speed of 10m/s, during its receiving a packet of 1500B, the node moves 12m. Thus, the surrounding can change significantly and cause reception failure.

(2) Node velocity is insignificant compared to the speed of light. In particular, none of the provided propagation models include Doppler effects, although they could.

1.2 GloMoSim

GloMoSim [3] is another open-source network simulator based on a parallel discrete event scheduler. Hopefully, it can emulate the real world more accurately. However, it is hard to debug parallel programs. Although GloMoSim currently only supports pure wireless networks, it provides more physical-layer models than ns-2, as shown in Table 1 [4].

Table 1. Physical layer and propagation models available in GloMoSim, ns-2 and OPNET

Simulator	GloMoSim	*ns-2*	OPNET
Noise (SNR) calculation	Cumulative	Comparison of two signals	Cumulative
Signal reception	SNRT based, BER based	SNRT based	BER based
Fading	Rayleigh, Ricean	Not included*	Not included
Path loss	Free space, Two ray, etc.	Free space, Two ray	Free space

1.3 Ns-2 Basics

Ns-2 directory structure

As shown in Figure 1, the C++ classes of ns-2 network components or protocols are implemented in the subdirectory "ns-2", and the TCL library (corresponding to configurations of these C++ instances) in the subdirectory of "tcl".

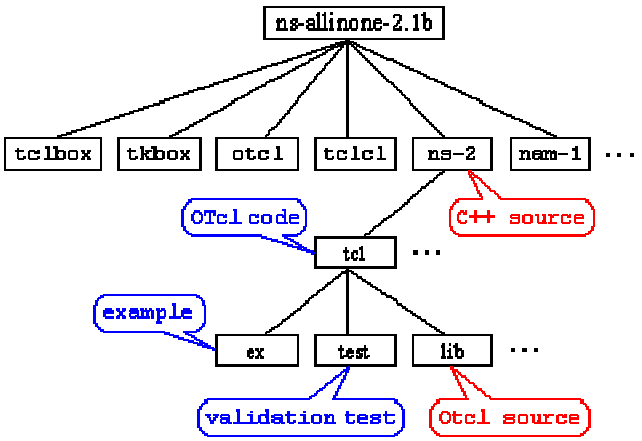


Figure 1. Ns-2 directory structure

Network Components

Network components are Node, Link, Queue, etc. Some of them are simple components, that is, they are created from the corresponding C++ classes; The other are compound components, that is, they are composed multiple simple C++ classes, e.g. Link are composed of Delay (emulating propagation delay) and Queue. In general, in ns-2, all network components are created, plugged and configured from TCL.

Example: Plug MAC into NetIF (Network Interface)

```
Class MAC {
    void send (Packet* p);
```

```
    void recv(Packet*, Handler* h);
    NsObject* target_ //an instance of NetIF
}
```

Event Scheduling

Events are something associated with time. class Event is defined by {time, uid, next, handler}, where time is the scheduling time of the event, uid is the unique id of the event, next is the next scheduling event in the event queue that is a linklist, and handler points to the function to handle the event when the event is scheduled. Events are put into the event queue sorted by their time, and scheduled one by one by the event scheduler. Note that class Packet is subclass of class Event as packets are received and transmitted at some time. And all network components are subclass of class Handler as they need to handle events such as packets.

The scheduling procedure (void Scheduler::schedule(Handler* h, Event* e, double delay)) is shown in Figure 2. The event at the head of the event queue is delivered to its hander of some network object. Then, this network object may call other network object, and finally some new events are inserted into the event queue.

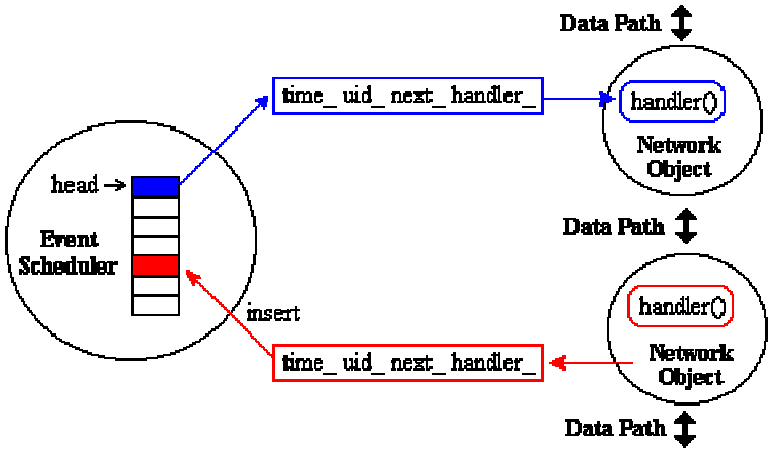


Figure 2. Discrete Event Scheduler

Example: A and B are two mobile nodes. And A sends packet p to B (suppose they are within the tx range).

```
    A::send (Packet* p) {target_->recv(p)} //target_ is B; call B::recv
    B::recv(Packet*, Handler* h = 0) {
    …
    Scheduler::instance().schedule(target_, p, tx_time) //target_ is B; schedule the packet at the
        // time of (current_time + tx_time)
    …
}
```

Example: Timer is another kind of Event that is handled by TimerHandler

```
class TimerHandler: public Handler
```

```
resched(double delay) //the time expires at the time of (current_time + delay)
handle(Event *e){
        expire (Event *e) //the virtual handling function overloading by users
}
```

Note: In ns, NO REAL time, timer, recv, send and packet flows in the sense of UNIX network programming.

2. 802.11 Implementation

2.1 Physical Layer

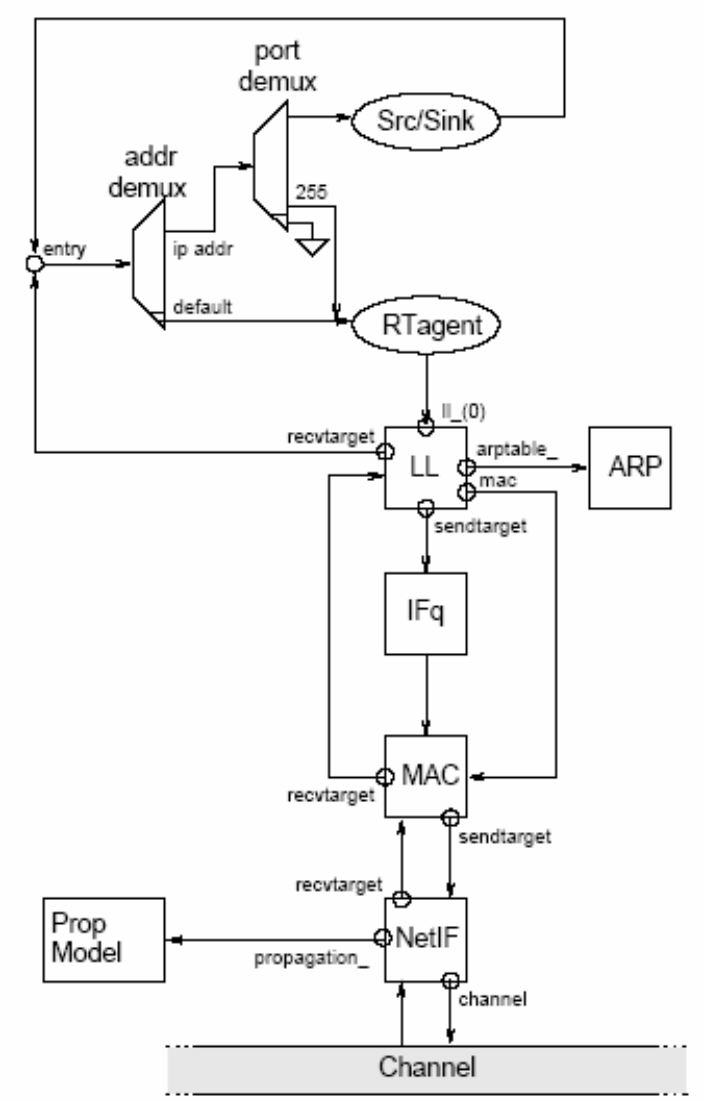


Figure 3 Schematic of a mobile node under the CMU Monarch wireless extensions to ns.

Figure 3 shows the network components in the mobile node and the data path of sending and receiving packets. In this section, we describe the basic function of the physical layer and MAC is

detailed in the next section.

Channel (channel.cc)

The function of class Channel is to deliver packets from a wireless node to its neighbors within the sensing range.

I. Stamp txinfo in the packets before sending:

```
p->txinfo_.stamp((MobileNode*)node(), ant_->copy(), Pt_, lambda_)
```

Note: Here node() are the pointer of the sending node, ant_->copy() is the antenna's parameters such as the height of the antenna, Pt_ is the transmitting power, and lamba_ is the wavelength of light. These information is used for the receiving node to calculate the receiving power.

II. Send packets to the nodes within the sensing range distCST_ to be sensed or received by these nodes.

```
distCST_ = wifp->getDist(wifp->getCSThresh(), wifp->getPt(), 1.0, 1.0,
            highestZ , highestZ, wifp->getL(),   wifp->getLambda());
```

Note: distCST is calculated by the parameters such as CS Threshold, transmission power, antenna gains, antenna heights, system loss factor, and wavelength of light.

NetIF (wireless-phy.cc)

The function of class WirelessPhy is to send packets to Channel and receive packet from Channel.

I. Packet Sending

```
channel_->recv(p, this);
```

II. Packet Reception, sendUp()

```
//calculate Rx power by path loss models
Pr = propagation_->Pr(&p->txinfo_, &s, this)

if (Pr < CSThresh_) {
      pkt_recvd = 0; // cannot hear it
      …
}
if (Pr >= CSThresh_ && Pr < RXThresh_){
      pkt_recvd = 1;
      hdr->error = 1; // error reception, for Carrier Sense
      …
}
if (Pr >= RXThresh_) {
      pkt_recvd = 1;
      hdr->error = 0; // maybe correct reception
      …
}
```

Note: First, ns-2 calculates the receiving power Pr by the tx_info_ of p and the receiver this. When Pr is less than CSThresh_ (Carrier Sense Threshold), the receiver cannot sense it; else, the receiver can sense it and even receive it without error in the case that Pr > RXThresh_ (Reception Threshold, and RXThresh_ > CSThresh_). Besides, successful reception also depends on the packet's SIR is larger than CPThresh_ (Capture Threshold), which is checked in MAC layer.

2.2 MAC

MAC (mac-802_11.cc)

The function of class Mac802_11 has 2 functions. On sending, it uses CSMA/CA medium access mechanism; On receiving, it adopts SIRT (SIR Threshold) based reception (Capture).

State Transition Diagram

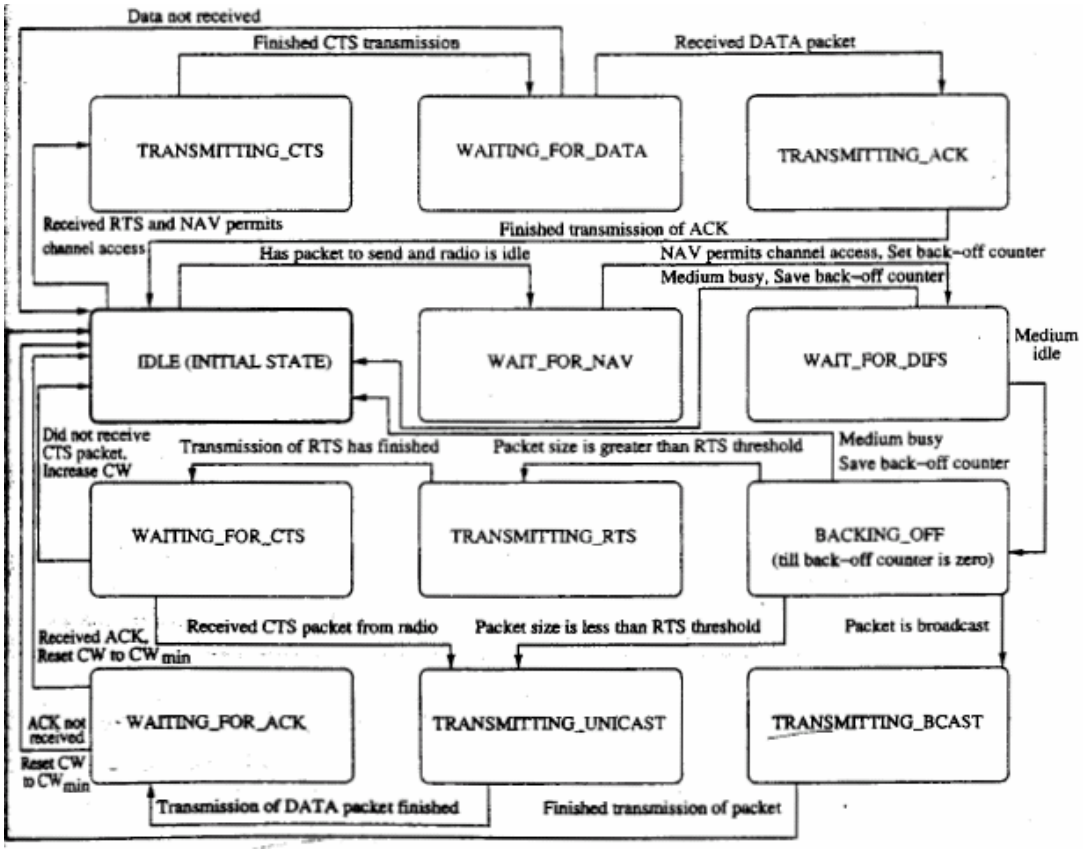


Figure 4. 802.11 MAC state transition diagram

State transition diagram can help us write or read network programs. Thus, before analyzing 802.11 source codes in ns-2, we first show the reference 802.11 MAC state transition diagram [5] in Figure 4 that is somewhat different with ns-2. First, we need to find out the basic states.

Elementary States

```
enum MacState {
        MAC_IDLE    = 0x0000,
        MAC_POLLING    = 0x0001, // ns 802.11 does not implement Polling
        MAC_RECV  = 0x0010,
        MAC_SEND  = 0x0100,
        MAC_RTS           = 0x0200,
        MAC_CTS           = 0x0400,
        MAC_ACK           = 0x0800,
        MAC_COLL  = 0x1000
};

MacState        rx_state_ //can be MAC_IDLE, MAC_RECV, MAC_COLL
MacState      tx_state_//can be MAC_IDLE, MAC_SEND, MAC_RTS, MAT_CTS, MAC_ACK
double           nav_ //expiration of Network Allocation Vector


//channel is idle
int is_idle() {
   if (tx_state_ == MAC_IDLE && rx_state_ ==    MAC_IDLE
          && nav_ <= NOW)
          return 1;
   else
          return 0;
}
```

Note: The above is_idle() check whether the channel is idle at the moment when it is called.

MAC Timers

Timers are very important in 802.11 in triggering channel access. The following shows the basic timers and their functions.

```
BackoffTimer mhBackoff_
        void start(int cw, int idle);// if is_idle(), start to count down; else freeze the timer
        void pause(); //freeze the timer when the
        void resume(double difs);//resume to count down after DIFS
        void handle(Event *); //send RTS or DATA after it times out
        int busy(); //Is counting down

DeferTimer mhDefer_
        void start(double defer);//start to count down
```

```
    void handle (Event *);//eg. send CTS or ACK after SIFS expires
    int busy(); //Is counting down

IFTimer     mhIF_;      // interface timer, set interface state active when transmitting
NavTimer    mhNav_;     // NAV timer
RxTimer     mhRecv_;        //completion of incoming packets, call recvHandler()
TxTimer     mhSend_;        //sending timeout (e.g. no ACK received), call sendHandler()

Recv/Send functions
    void setTxState (MacState newState) //For tx_state_
    void setRxState (MacState newState)//For rx_state_
        void checkBackoffTimer() {
            if(is_idle() && mhBackoff_.paused())
                mhBackoff_.resume(phymib_.getDIFS());
            if(! is_idle() && mhBackoff_.busy() && ! mhBackoff_.paused())
                mhBackoff_.pause();
        }
```

Note: the above sample codes show how receiving and sending will change MAC state and further control the backoff timer.

```
/* Note: nav_ expires also mean channel is idle, then call mhBackoff_resume() */
void set_nav(u_int16_t us) {
        double now = Scheduler::instance().clock();
        double t = us * 1e-6;
        if((now + t) > nav_) {
            nav_ = now + t;
            if(mhNav_.busy())
                mhNav_.stop();//reset nav_
            mhNav_.start(t);
        }
    }
```

Note: NAV timer is set by RTS or CTS to indicate the residual time of data transmission. However, it is extended in ns-2 to also reflect the residual time before channel becomes idle, and thus can replace the function of carrier sense. The usage is, update the NAV timer with the transmission time of either received packets or sensed packets, set_nav (txtime(p)). When NAV timer expires, navHandler() is called to resume backoff timer.

CSMA/CA

recv function is generally the entry of most network protocols (handling packets from both uplayer and downlayer). For outgoing packets, it will call send function that is the entry of CSMA/CA.

```
void recv(Packet *p, Handler *h) {
      struct hdr_cmn *hdr = HDR_CMN(p);
      //handle outgoing packets
      if(hdr->direction() == hdr_cmn::DOWN) {
                  send(p, h); //CSMA/CA
                  return;
            }
      …
      //else, handle incoming packets
}

void send(Packet *p, Handler *h) {
   …
   if(mhBackoff_.busy() == 0) {
            if(is_idle()) {
                  if (mhDefer_.busy() == 0) {
                        /*
                         * If we are already deferring, there is no
                         * need to reset the Defer timer.
                         */
                        rTime = (Random::random() % cw_)
                              * (phymib_.getSlotTime());
                        mhDefer_.start(phymib_.getDIFS() + rTime);
                  }
            } else {
                  /*
                   * If the medium is NOT IDLE, then we start
                   * the backoff timer.
                   */
                  mhBackoff_.start(cw_, is_idle());
            }
      }
}
```

Capture Model

Ns-2 uses a simplified capture model: When multiple packets collide at the receiver, only the *first* packet can be successfully received if its Rx Power should be larger than any of the other packets by at least CPThresh (10dB in ns-2).

```
void recv(Packet *p, Handler *h){
   …
```

```
//Handle incoming packets

/*
*    When there is no packet reception, log receiving p at pktRx_
*/
if(rx_state_ == MAC_IDLE) {
        setRxState(MAC_RECV);
        pktRx_ = p;
        mhRecv_.start(txtime(p));// schedule the reception of this packet in txtime
                                 //setRxState(MAC_IDLE) again after reception.
}
/*
*    When there is already a packet reception (in pktRx_), calculate the inference
*/
else {
        //Simplified SIR calculation (Comparison of two signals)
        if(pktRx_->txinfo_.RxPr / p->txinfo_.RxPr >= p->txinfo_.CPThresh) {
            capture(p);//pktRx_ can be correctly received;
                       //recalculate when the channel will be idle
        } else {
            collision(p);//stop receving pktRx_ (i.e. mhRecv.stop() )
                       //recalculate when the channel will be idle
        }
    }
}
```

2.3 An example of user extension: Add Fading

The probability density function of Rayleigh fading is $pdf(r) = \frac{r}{\sigma^2} e^{-\frac{r^2}{2\sigma^2}}$, where *r* stands for Voltage. As Power = c . r^2, the probability density function for Power is $pdf(P) = \frac{1}{\overline{P}} e^{-\frac{P}{\overline{P}}}$, where *P* stands for Power. And $\overline{P}$ is the mean of *P*, that is, Pr by path loss. So, we add the fading calculation after pass loss calculate Pr ().

```
#include <random.h>
…
int WirelessPhy::sendUp(Packet *p) {

double Pr;
…
```

```
if (propagation_) {
                    s.stamp((MobileNode*)node(), ant_, 0, lambda_);
                    Pr = propagation_->Pr(&p->txinfo_, &s, this);

                    /* Add Rayleigh fading (neglect time-correlation)*/
                    double mean = Pr;
                    Pr = Random::exponential(mean);

                    if (Pr < CSThresh_)
                    …
}
```

We do two experiments in 11Mbps 802.11 networks to see the impacts of Rayleigh fading. The default tx power Pt is 0.28, and thus tx range and sensing range are calculated as 250m and 550m.

The first experiment (Figure 5) is to test TCP performance. We vary the distance d from 50m to 250m. Figure 6 shows TCP throughputs as the function of the distance. When d becomes larger, fading can cause more packet loss and thus reduce TCP throughputs significantly.

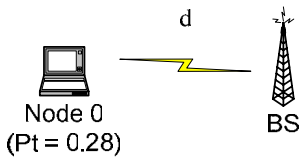


Figure 5. TCP under Rayleigh fading (Node 0 sends TCP packets to BS)

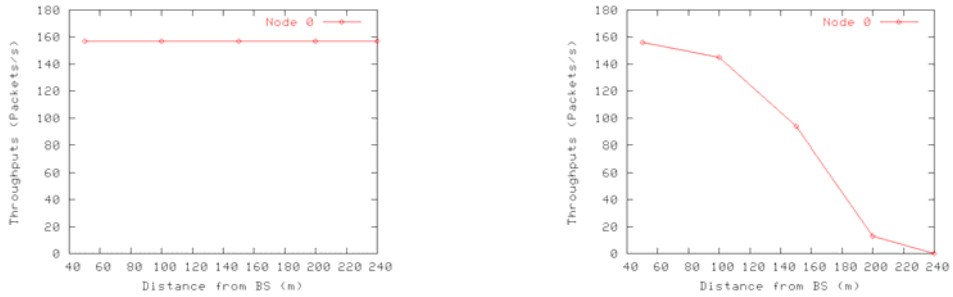


Figure 6. TCP Throughputs a function of the distance from BS (a) Without Fading (b) Rayleigh Fading

The second experiment (Figure 7) is to test UDP performance (assume saturate condition). We set Pt of Node 0 be 10 times of the default Pt (SIR is 10), where Node 0 must capture when its packets collide with Node 1 at BS suppose there is no fading. As shown in Figure 8, fading aggravates the unfairness of two senders as their loss rate is dominated by Pt instead of capture, when d becomes larger.

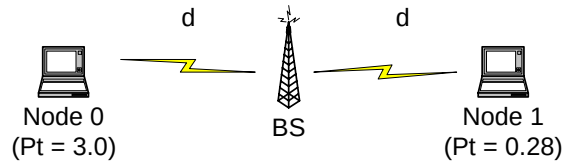
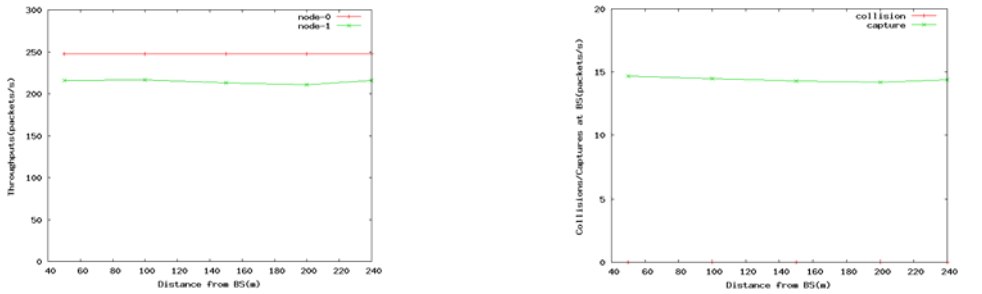
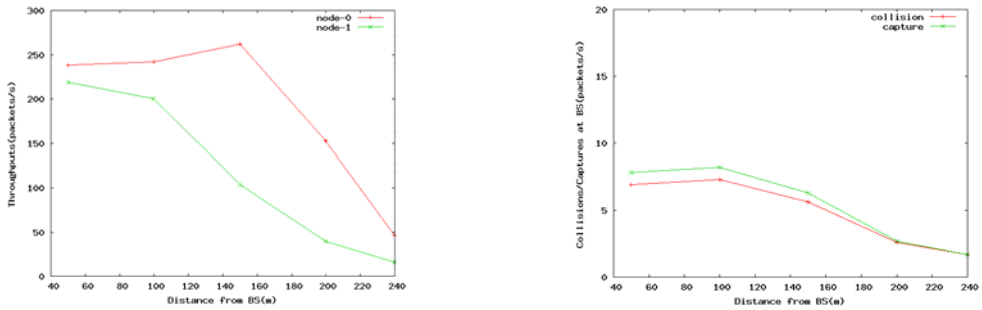


Figure 7. Capture under Rayleigh fading (Node 0 and 1 send CBR packets to BS)



(a)



(b)

Figure 8. UDP Throughputs and Captures/Collisions at a function of the distance from BS (a) Without Fading (b) Rayleigh Fading

Finally, our simple implementation of fading does not consider the time correlation. For this feature, please refer to CMU's fading patch [6].

2.4 Bugs in ns-2

Assumptions

Simulators always need assumptions to make their calculations viable (recall Section 1.1). However, when some assumptions are crucial in your simulations, you must be careful. For example, there is no scanning for WLAN (Discovery/Select/Authentication/Association) in ns-2, mobile nodes are associated with their BS automatically if they have the same pre-defined domain. If you want to study the overhead of scanning, you should make your extension.

Standard Misinterpretation

Ns-2 may misinterpret some network protocols, even standard for it is an open project. For example, we find ns-2 802.11 implementation seems abuse EIFS (set_nav(usec(phymib_.getEIFS() + txtime(p))) // whenever p is error, defer EIFS) [7]. Actually, in Figure 7, the node with Pt 0.28 will has more throughput before we let getEIFS() return 0 to eliminate the effect of EIFS.

3. Reference

[1] *The network simulator - ns-2*, http://www.isi.edu/nsnam/ns/

[2] *The CMU Monarch Project's Wireless and Mobility Extensions to ns*, http://www.monarch.cs.cmu.edu/

[3] *GloMoSim*, http://pcl.cs.ucla.edu/projects/glomosim/

[4] *Effects of Wireless Physical Layer Modeling in Mobile Ad Hoc Networks*, MOBICOM 2001

[5] *Ad Hoc Wireless Network*, PRENTICE HALL 2004

[6] *Additions to the NS network simulator to handle Ricean and Rayleigh fading*, http://www.ece.cmu.edu/wireless/

[7] *EIFS*, Section 9.2.3.4, ANSI/IEEE Std 802.11, 1999 Edition