Understanding the implementation of IEEE MAC 802.11 standard in NS-2

Ke Liu

1 WANET simulation based on IEEE 802.11 with NS2

The components on a single node are (from bottom to up)

- Channel: class `WirelessChannel`, file ns2/mac/channel{.h,.cc}, inheriting from `Channel` in the same files;
- Network Interface (netif): class `WirelessPhy`, files ns2/mac/wireless-phy{.h,.cc}, inheriting from class `Phy`, in files ns2/mac/phy{.h,.cc};
- Propagation Model: a composed component of network interface, class `Propagation` and `MobileNode`;
- Media Access Control (MAC): class `Mac802_11` inheriting from the abstract class `MAC`;
- Outgoing Queue: queue has only one target (down-target), mac; it is not the up-target of mac. class `Queue` and many variations such as droptail, priqueue, etc;
- Link layer: class `LL`(inherits from `LinkDelay).`It has a composed component `ARP` which works as ARP procedure, mapping the protocol address (such as IP address) to Hardware address (such as MAC address);
- Network layer: routing agents with many variations such as DSDV, DSR, AODV, etc

1.1 WirelessChannel

The `WirelessChannel` simulates the physical media (air) for wireless communication. The class `WirelessChannel` inherits from class `Channel`. The implementation is in files `ns2/mac/channel{.cc,.h} .`

The `WirelessChannel` keeps a list of all nodes on this channel (mostly, all the nodes participate the simulation). The list is sorted based on the X-dimension values of nodes before it can be used. A state `delay_` is kept by the channel (inherited from `Channel`).

1.1.1 Class States

- `MobileNode* xListHead_`: the list of all nodes on channel;
- `numNodes_`: the number of nodes on channel (size of the list);
- `bool sorted_`: whether the list is sorted based on the x values of nodes location;
- `double distCST_`: the distance of the carrier sense threshold. It is used to decide where the receiving packets on channel going. Only the nodes in the range of the sender (decided by this threshold) would be assigned copy of packet. (Notice: a safty marge of 5 meters is used).

1.1.2 functionality

All funcationalities about the contention, resume, transmission in the implementation of `Channel` in previous version of ns2 have been removed to a better place, the mac. So even `WirelessChannel` inherits from the `Channel`, it works just like a physical media: receiving packets from nodes and assigning them to all possible destination (nodes in range of the sender).

- `SendUP(Packet*,Phy*)`: Let's recall the implementation of this function and its caller `recv(Packet*,Handler*)` in its parent class `Channel`. The function `recv()` is called only by the sender's physical layer (network interface). The `Handler*` is the physical layer of the sender indeed. So the function `SendUP` makes the forwarding decision based on this sender's physical layer. The function `SendUP` in class `Channel` just duplicates the receiving packet to all other nodes on this channel except the sender.

  In the wireless channel, only the nodes in the range of sender would receive a copy of the receiving packet on channel. This range is decided by the `distCST_` and the following inequations:

$$sender.x - (distCST\_ + sm) \leq receiver.x \leq sender.x + (distCST\_ + sm) \tag{1}$$
$$sender.y - (distCST\_ + sm) \leq receiver.y \leq sender.y + (distCST\_ + sm) \tag{2}$$

  where sm denotes safty margin (5 meters used by ns2).

- `MobileNode**`
  `getAffectedNodes(MobileNode*, double radius, int* numAffectedNodes)`: generates the list of the nodes in range of sender

- `void sortLists(void)`: sorts the list of nodes on channel, bubble sorting (simulation slowed)

- `void addNodeToList(MobileNode*)`: add a new node into the list of nodes on channel. The new node is appended at the tail of the existing list (simulation slowed).

1.1.3 Problems

As we can see, the implementation of the `Channel` does not consider the performance of the simulation running (not the results). More than enough copies of packets are assgined to nodes, caused by the big safty margin and the inequation used.

1.2 MobileNode

The class `MobileNode` simulates the mobile or wireless nodes in wireless ad hoc networks or sensor networks on wireless channel. It inherits from the class `Node`. The main difference is that there is no links on any `MobileNode`, the receiving and sending packets are based on the `WirelessChannel` not the links. It also simulates the mobility of nodes, based on the speed and destination locations. The implementation is in files `ns2/common/mobilenode{.cc,.h}` .

Mostly, class `MobileNode` is used to trace the mobility of nodes.

1.3 LinkLayer LL

The class `LL` simulates the link layer, inheriting from class `LinkDelay`.

1.3.1 Class State

- States inherited from class `LinkDelay`:
  - `double delay_`: the link layer delay, for wireless communication simulation, the default value would be set to $25\mu s$

- double bandwidth_: for wireless communication, this value is not used;

- macDA_: the mac address of the attached node;

1.3.2 Functionality

- recv(): called by up or down target
- sendUP(): checking the receiving packet with error or not, if not, forwarding to up-target with a link layer delay;
- sendDown(): inserting the mac address by looking up the arp table, or starting arp if not available (through ARP functionality); forwarding the packet to down-target with a link layer delay

1.4 ARP

The class ARP simulates the ARP procedure, inheriting from class LinkDelay, the same as the link layer.

1.4.1 Class State

- ARPEntry_List arphead_: the list of all known arp entries (the mapping from protocol address to mac address);
- MobileNode* node_: the attached mobile node;
- Mac* mac_: the mac address of the attached node;

1.4.2 Functionality

- ARPEntry* arplookup(nsaddr_t): return the corresponding ARPEntry to the given protocol address; It checks the list of the arphead_;
- arpresolve(nsaddr_t,Packet*,LL*): given by the protocol address of the destination, this function is used to set the mac address field in packet header if it is known in arp table; otherwise, it would start an arp request; if for a given protocol address, the times of un-successful arp request reaches the ARP_MAX_REQUEST_COUNT ; then dropping all the held packets waitingfor arp and calling the possible arp failure callback; (Note: there is a problem here, it seems only one packet can be held for waiting at arp procedure. If a new packet arrives for the same arp, the previous packet held would be dropped.)
- arprequest(nsaddr_t,nsaddr_t,LL*): creating a new packet typed as PT_ARP, as an arp request packet; broadcasting it through the downtarget of LL*(MAC layer), querying the mapping of the given destination address;
- arpinput(Packet*,LL*): called by attaching link layer object when receiving a packet of type PT_ARP(either an arp request or reply); if it is an arp request, reply with local mac address; if it is an reply, update the arp table, seeing whether there is any packet held for this address, sending it

1.5 Phy

The class Phy is a pure virtual class needed to be inherited explicitly.

1.5.1 Class State

- bandwidth_: the bandwidth of the physical layer.

1.5.2 Functionality

- `double txtime(Packet*)`or `txtime(int)`: return the transmission time of the given packet or the number of bytes on the physical layer according to the `bandwidth_`;
- `recv(Packet*,Handler*)`: this function is called by either up-target or down-target. Based on `direction` field in the common header of the packet, pure virtual functions `sendUP()` or `sendDown()`would be called.

1.6 Network Interface: wirelessPhy

The class `WirelessPhy` simulates the wireless physical layer, inheriting from the class `Phy`. Besides working for receiving and sending packets as any other physical layer, it deals with the propagation model (in ns2, mostly 3 models can be used: *free space, Two Ray Ground, Shadowing*), node sleeping (duty cycle management), energy model management, and maybe different antenna models and modulations. Here, I just describe the transmission behaviors and related features of this class. Any other thing, such as energy, sleeping, is not considered here.

Notice: for the energy consumption tracking simulation, the energy consumption configuration should be done through some commands of this `WirelessPhy` class, not the energy model class. This is not good implementation of ns2, as I think.

1.6.1 Class States

The `bandwidth_` is no more effective here. How to obtain the transmission time through the bandwidth has been moved to the implementation of MAC layer protocol such as `MAC802.11`. You may see a commented out `bind_bw("bandwidth_",&bandwith_)`.

- Member parameters can be configured through tcl script
  - `double Pt_`: the transmitted signal power in Watt;
  - `double freq_`: the frequency;
  - `double L_`: the system loss factor (mostly 1.0);
  - `double RXThresh_`: the receiving power threshold in Watt;
  - `double CSThresh_`: the carrier sense threshold in Watt;
  - `double CPThresh_`: the capture threshold in Watt;
- `double lambda_`: the wavelength in meters. It is calculated through `lambda_=SPEED_OF_LIGHT/freq_`;
- Channel Status: is one of the 4 status, SLEEP, IDLE, RECV, SEND, but only SLEEP, IDLE are actually used;
- other composed member objects:
  - `Antenna* ant_`: antenna
  - `Propagation* propagation_`: propagation model;
  - `Modulation* modulation_`: modulation scheme;

1.6.2 Functionality

- `sendDown(Packet*)`: sending the packet to the channel, updating the energy of node, stamping the packet with interface arguments;
- `sendUp(Packet*)`: called by channel, deciding whether the given packet can be received or captured or not according to the packet stamp (antenna, power), the attached propagation model, the energy model and the modulation scheme; then set the channel status to IDLE;

1.7 MAC

The class Mac simulates the mac layer object, working as parent class for all kind of mac types;

1.7.1 Class States

- The composed objects
  - LL* ll_: link layer object, up-target;
  - Phy* netif_: network interface, down-target;
  - Channel* channel_: down-target of down-target;
- Handler* callback_: whose down-target is this(mostly, the queue);
- MacState state_: deciding the state of the mac or channel, such as IDLE, RECV, SEND, COLL, RTS, CTS, ACK, POLLING;
- double bandwidth_: the really effective bandwidth of the transmission simulation;
- double delay_: the MAC layer computing overhead

1.7.2 Functionality

Since it is just an abstract class, most the functions of it would be overrided by inherited classes (see the following MAC802.11).

1.8 IEEE 802.11 MAC

1.8.1 Class States

- PHY_MIB phymib_: physical layer management information base (MIB) such as the minimal and maximal size of contention window (CWMin, CWMax), the slot time for each slot in contention window (SlotTime), the SIFS, DIFS, PIFS, EIFS, etc.
- MAC_MIB macmib_: mac layer MIB such as the threshold for packet size over which the RTS/CTS would be adapted (RTSThreshold), STA short or long retry limit (ShortRetryLimit LongRetryLimit), failure counters, etc.
- bss_id_: for network of infrastructured model;
- basicRate_: transmission rate for control packets such as RTS, CTS, ACK and Broadcast;
- dataRate_: transmission rate for mac layer data packets;
- mhNav_: NAV (network allocating vector) counting down timer;
- mhRecv_: receiving timer;
- mhSend_: sending timer;
- mhDefer_: defer timer;
- mhBackoff_: backoff timer;
- double nav_: NAV in seconds;
- rx_state_: receiving or incoming state (MAC_RECV or MAC_IDLE);
- tx_state_: sending or outgoing state

- `tx_active_`: transmitter is active or not;
- `Packet* pktRTS_`: outgoing RTS packet when sending RTS;
- `Packet* pktCTRL_`: outgoing control packet (CTS or ACK);
- `Packet* pktTx_`: the packet needed to be sent out such as data packet or any other packet from upper layer;
- `cw_`: current size of contention window;
- `ssrc_`: current STA short retry counter;
- `slrc_`: current STA long retry counter;

1.8.2 Functionality

- Misc functions
  - `set_nav(u_int16_t)`: set the NAV according to the given unsigned short integer value, times μs; if the existing NAV is later than this value, ignoring it, otherwise, update the NAV;
  - `is_idle(void)`: checking the receiving state, sending state and NAV, if any of them is not idle, then the state of MAC is not idle;
  - `inc_cw()`: increasing number of slots in the contention window when backoff; IEEE 802.11 standard specifies it from 63 to 1023, each time increasing by an order of 2
  - `rst_cw()`: reseting the contention window to the basic setting, 63 slots;
  - `sec(double)`: given an integer value of μs, return its equivalent value in seconds;
  - `usec(double)`: given a double value of seconds, return its integer value of μs
  - `double txtime(Packet*)`: return the transmission time field in common header `hdr_cmn::txtime()`;
- `recv(Packet*,Handler*)`: called by up-level agent (queue) or down-target object (network interface: WirelessPhy); if it is a outgoing packet (coming from upper level queue, then `hdr_cmn::direction()==hdr_cmn::DOWN`), calling function `send()`; otherwise, it is an incoming packet.
  - If it is in TRANSMIT model, setting the packet as an error
  - If the receiving state `rx_state_` is not `MAC_IDLE`, depending on the power of the packet transmission, capturing it and setting the backoff to EIFS, or collision happens;
  - If the receiving state is `MAC_IDLE,`setting the receiving state `rx_state_` to `MAC_RECV`, and the packet to the receiving packet `pktRx_`; calling the packet to be received at time `txtime(p)` through receiving timer `mhRecv_`;
- `transmit(Packet*,double timeout)`: the actually function places the packet on the channel no matter what the type of the packet is (control, RTS/CTS, Ack, Data); the `timeout` estimates when the sending finishes, setting the send timer `mhSend_` whose expiring calls the `sendHandler()`, leading to call the `send_timer()`;
- `void recv_timer()`: the effective function taking care of the receiving packet. The packet received and waiting to be processed is held by field `pktRx_`. The `pktRx_` would be checked for collision, error and the destination address (mac address) first. The NAV would be set through `set_nav()`; receiving energy is updated; and effective received packet would be forwarded to corresponding functions called here; and then the mac state would be set to IDLE;
- `recvRTS(Packet*)`: called by `recv_timer()` when the received packet held in `pktRx_` is a RTS packet. It calls `sendCTS()` which sets the `pktCTRL_` to a CTS packet, and calls `tx_resume()`;

- `recvCTS(Packet*)`: called by `recv_timer()` when a CTS packet is received, which means the RTS packet was received by the receiver; it clears the `pktRTS_` which holds the RTS pacekt for this CTS, stops the sending timer `mhSend_` if it is set; resets the STA short retry counter `ssrc_` and calls `tx_resume()`;
- `recvDATA(Packet*)`: called by `recv_timer()`. If the destination is not a broadcast address (it is a unicast packet), acknowledge would be sent through calling `sendACK()`, and `tx_resume()`would be called. Finally, the received data packet would be forwarded to up-target if it is not dropped (due to a control packet held in `pktCTRL_`if another transmission is on the way);
- `recvACK(Packet*)`: called by `recv_timer()` if acknowledge packet is received. It resets the STA retry counter `ssrc_` or `slrc_`, the size of contention window, and clears packet held in `pktTx_`; then calls the `tx_resume()`;
- `rx_resume()`: called by `recv_timer()`, setting the receiving state to `MAC_IDLE`;
- `tx_resume()`: called by `send_timer()`, `recvRTS()`, `recvCTS()`, `recvDATA()`, `recvACK()`;
  - checking the `pktCTRL_`: a control packet is held which means a CTS or ACK needs to be sent. It sets the defer timer `mhDefer`to SIFS through `phymib_.getSIFS()`, and sets the transmission state to `MAC_IDLE` then return;
  - checking the `pktRTS_`: a RTS packet is held and backoff timer is not set which means a RTS packet has been sent out and expire, a retransmission may need. It sets to a contention window time to defer timer, and sets the transmission state to `MAC_IDLE` then return;
  - checking the `pktTx_`: a packet needed to be sent (coming from higher layer such as data packet, ARP packet, routing control packet, etc.), which means some previous try of sending this packet failed, a retransmission is needed. It sets the defer timer to a new contention window time. (If the packet is smaller than the RTS threshold or it is a broadcast packet, the defer timer would just be set to SIFS), and sets the transmission state to `MAC_IDLE` then return;
  - If all the above checking fails, wich means there is no packet waiting to be sent, the call back handler held by `callback_` would be called (mostly, it is a callback function of the upper level agent such as queue's resume function), and sets the transmission state to `MAC_IDLE` then return;
- `check_pktCTRL()`: checking the packet held in `pktCTRL_`. Return 0 if succeeds (a control packet is held by this field), otherwise return -1 (no control packet is held or the control packet is on sending). There are 2 types of control packet: CTS and ACK. If the `pktCTRL_` is a CTS packet, it sets the transmission state to `MAC_CTS`through `setTxState()`, calculating the the transmitting time of CTS packet, calling the `transmit()` to place the CTS packet on channel. If the `pktCTRL_` holds an ACK packet, it would calculate the transmission time of an ACK and call the transmit() to send this ACK packet. It also sets the transmission state to `MAC_CTS` or `MAC_ACK` respectly;
- `check_pktRTS()`: checking the `pktRTS_` holding a RTS packet. Return 0 if it holds, otherwise return -1. If a RTS is held, either it would be sent by calling `transmit()`,setting the transmission state to `MAC_RTS`, or if channel is not idle (through calling `is_idle()`), backoff timer would be set (as well as increasing contention window size);
- `check_pktTx()`: checking the `pktTx_` holding a packet. Return 0 if it holds, otherwise return -1. If the channel is idle, the packet would be sent by calling `transmit()`, and the transmission state would be set to `MAC_SEND`; otherwise, the backoff timer would be set (increasing contention window size as well);
- `deferHandler()`: checking if there is a packet waiting to be sent by calling `check_pktCTRL()`, `check_pktRTS()`, `check_pktTx()`;

- `send_timer()`: the send timer would be set by `transmit()`; when the send timer expires, this function would check the transmission state `tx_state_`;
  - if it in `MAC_RTS`, it would call `RetransmitRTS()`;
  - if it is in `MAC_CTS`, which means a CTS was sent but the corresponding data was not received. It would clear the `pckCTRL_` which holds the CTS packet;
  - if it is in `MAC_SEND`, which means a data packet was sent but ACK was not received. It calls `RetransmitDATA()`;
  - if if is in `MAC_ACK`, which means an ACK packet was sent successfully, clearing the `pktCTRL_` which holds the ACK packet;
  - if it is in `MAC_IDLE,`do nothing;

  After all, it calls the `tx_resume()`;

- `sendRTS(int dst)`: checking if a RTS is needed (by checking the packet size against the RTS threshold, and the destination address against the broadcast address), allocating a RTS packet, assigning it to `pktRTS_`;
- `sendCTS(int dst, double rts_duration)`: allocating a CTS packet, assigning it to `pktCTRL_`;
- `sendDATA(Packet*)`: assigning the packet to `pktTx_`, if it is a broadcast, calculating the transmitting time based on `basicRate_`, otherwise based on `dataRate_`;
- `sendACK(int dst)`: allocating an ACK packet, assigning it to `pktCTRL_`;
- `RetransmitRTS()`: called by `send_timer()` when the send timer `mhSend_` expires (after transmission through calling `transmit()`), increasing RTS failure counter, STA short retry counter `ssrc_`; if the RTS transmitting fails (the retry counter exceeds), it would drop the `pktTx_`, clear the retry counter and reset the size of contention windows through calling `rst_cw()`; if not, it would increase the contention window by calling `inc_cw()`, start the backoff timer;
- `RetransmitDATA()`: if `pktTx_` holds a broadcast packet, it would just clear it, reset the contention window and start the backoff timer; otherwise, increasing the ACK failure counter, increasing the STA retry counter `ssrc_` or `slrc_`; if the retry counter exceeds, calling the sending failure callback (`hdr_cmn::xmit_failure()`) and clear the `pktTx_`, reset the retry counter and contention window; if not, it would call `sendRTS()`(**important**), increase the contention window `inc_cw()` and start backoff timer;

1.8.3 Control Flows

- Sending a data packet:
  1. Upper object (mostly, the queue) calls `recv()`;
  2. `recv()` calls `send()`;
  3. `send()` sets the data packet to `pktTx_` through calling `sendDATA(p)`, allocates a RTS packet to `pktRTS_` through calling `sendRTS()`, setting the deferTimer `mhDefer_` with a delay as DIFS + { random generated contention window slot };
  4. defer timer expiring would call `deferHandler()` which calls `check_pktRTS()`, sending the RTS packet through `transmit()` which sets the send timer;
  5. send timer expiring would call `sendHandler` calling `send_timer()` which may retransmit RTS packet through `RetransmitRTS()`;
  6. If a CTS packet is received, `recvCTS()` would delete the packet `pktRTS_`, stopping the send timer, and calling `tx_resume()` which may send the data packet held in `pktTx_`;

- Receiving a packet:
  1. Lower object (mostly, the network interface: WirelessPhy) calls the `recv()`;
  2. `recv()` sets the `pktRx_` to the received packet `p`, and the receiving timer `mhRecv_` to expire at time `txtime(p)`;
  3. the receiving timer `mhRecv_` expires would call `recvHandler` which calls the `recv_timer()`;
  4. `recv_timer()` checks the type and subtype of the received packet held in `pktRx_`, calling the corresponding functions such as Control packets : `recvRTS()` `recvCTS()` `recvACK()`, data packet: `recvData()`; packets of other type and subtype are not delt with now by ns2 code

1.8.4 Note:

- All the control packets would be transmitted through the `basicRate_` which is mostly 1Mbps, configuration is in the file ns2/tcl/lan/ns-mac.tcl; All the data packet would be transmitted through the `dataRate_` which is also mostly 1Mbps; You need to specify it through your tcl configuration simulation file if you need another value for them, especally a different `bandwidth_`;
- Although there is a field `delay_` in the parent abstract class `MAC` which simulates the MAC computational overhead, it is not used for any simulation in `MAC802_11`. So if a MAC layer overhead needs to be simulated, the `delay_` field needs to be inserted at 2 places:
  - simulating the send down overhead in `transmit()`: replacing `downtarget_->recv(p->copy(),this)` to `s.schedule(` **sometimer** `,p->copy(),delay_)`: the **sometimer** is a Handler or TimerHandler whose expiring would call the `downtarget_->recv()`;
  - simulating the send up overhead in `recvDATA()`: replacing `uptarget_->recv(p,(Handler*)0)` to `s.schedule(` **sometimer**, `p`, `delay_);`