Lab 4: Wireless & Ethernet

Task 1 [files required: ETHERFLOOD, etherflood.c, and showethercollisions.sh]

The cnet "protocol" in etherflood.c and its corresponding topology file ETHERFLOOD simply write IEEE802.3 format frames to a single LAN segment. The frames are of random length, and written at random intervals with a requested average frequency (such as 1000usecs). No attempt is made to avoid, detect, or resolve collisions on the LAN segment and, thus, when frames are written at a high rate, collisions are observed.

The shellscript showethercollisions.sh executes the etherflood simulation several times, varying the average interframe interval (measured in microseconds) by providing it as a parameter after the name of the topology file. Run the shellscript (it may take 2 minutes) to observe the number of collisions.

Task 2

Next, take a copy of etherflood.c develop a "more correct" IEEE802.3 implementation which attempts to avoid, detect, and resolve collisions on the LAN segment, by employing the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) algorithm.

Each node may sense if the LAN segment to which it is connected is busy, by calling CNET_carrier_sense(). Naturally, if a node detects a signal on the segment, it should not transmit its frames as they will cause a collision, and it should reschedule an attempt to transmit in the future.

Each node may also detect frame collisions on the LAN segment by defining an event handler for the cnet event EV_FRAMECOLLISION. The handler will be called for each collision heard on the LAN segment, even if the node did not cause the collision.

Develop code to both sense the network and to detect frame collisions, and to employ the standard IEEE802.3 binary exponential backoff algorithm to resolve collisions.

Can you devise a mechanism to measure the utilization of the LAN segment?

Task 3

[files required: GEOROUTING, georouting.c, and mobility.c]

GEOROUTING is a topology file which defines the attributes of a simple wireless Ethernet (IEEE802.11) simulation. You'll observe that the simulation executes over a fixed area, and that we can define as many, or as few, nodes as required.

- georouting.c implements a very simple mobile/wireless routing protocol. Each node periodically generates and transmits messages for delivery to other nodes. As copies of each message arrive at the mobile nodes, the destination either accepts messages addressed to it, or other nodes forward the message if they are closer to the final destination (The protocol is unrealistic in that every node 'magically' knows the location of all other nodes).
- Note that georouting.c does not use cnet's built-in Application Layer to generate and receive messages (it could do so with a small change, though the AL would report many errors until you implemented Q4 of this labsheet). Instead, georouting.c just generates short random messages in its transmit() function, and some statistics are printed when cnet is terminated.
- mobility.c implements the movement "strategy" of each node. The random waypoint model simply involves choosing a random destination, walking towards that destination at a constant speed, pausing at the destination for a specified time, and then choosing a new destination. Mobility happens "in the background", and is independent of wireless transmissions (of course, we can both walk and talk, can't we?) There is no real need to understand how this node mobility code works, although it follows cnet's familiar event-driven model.

Just execute the provided code with the command:

`cnet -gN GEOROUTING`

You may like to consider the shellscripts which each produce plots either as HTML for display in your browser. Each shellscript invokes multiple georouting simulations, varying either the nodes' walking speed or the amount of time that they pause before choosing a new walking destination. Are you able to provide an explanation for such variation in the plots?

Task 4

In the basic georouting protocol, above, nodes do not attempt to detect, avoid, or resolve frame collisions in the wireless medium. In fact, they are oblivious to all problems because each node transmits its wireless frames by calling CNET_write_physical_reliable() which (silently) does not introduce any collisions.

Replace the use of CNET_write_physical_reliable() with the standard CNET_write_physical(). Frame collisions should (may) now be observed and are reported on the Statistics subwindow.

Implement the traditional IEEE802.11 Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) mechanism, involving new Request To Send (RTS), Clear To Send (CTS), and Acknowledgment (ACK) control frames.

Collisions may (rarely) still occur, so you should also employ the use of the CNET_carrier_sense() function, an EV_FRAMECOLLISION event handler, and a binary exponential backoff algorithm.