Programming Project

Remember the Titans

Due: Data and reports due 12/10 & 12/11 (code due 12/7)

In the paper "Measured Capacity of an Ethernet: Myths and Reality ," David Boggs, Jeff Mogul and Chris Kent contrasted the results of several analytical studies of the performance of variants of the CMSA/CD protocol used in Ethernet with experimental data they collected on an actual network of 24 Titan workstations connected to a 10 megabit/second Ethernet. Good science is supposed to be reproducible, so I would like you to reproduce some of their experimental work. While we probably could find enough 10 megabit Ethernet cards to recreate their network, it would be hard to find 24 Titan workstations these day. So, for this assignment I would like you to reproduce the experiments conducted by Bogg, et.al. by writing a computer program to simulate their experimental environment.

You will have to do some experimenting just to recreate the experiments described in the paper. Boggs, et.al. do not provide all of the details you will need about their environment explicitly. For example, while they say the cables between their repeaters were 1000 feet long, they never say how long the cables that connected the repeaters to the Titans were. You will have to make some educated guesses and then try various alternatives to see what parameters makes your simulation results match their experimental results most closely. Limit your attention to the basic utilization and delay graphs in their paper as you concentrate on this tuning process (Figures 3-5 and 3-7). Approach this task critically! Do not blindly assume that their data is totally correct (or that your simulator is totally correct!). Finally, don't expect an exact match. Even if they ran their experiment again on exactly the same equipment they would get slightly different results.

Once you have a simulator that appears to correctly reproduce a close approximation of the experimental testbed used by Metcalfe and Boggs, I would like you to conduct an experiment of your own choosing using this simulator. I describe several possibilities below. You are welcome to suggest others.

1. In our meetings, we discussed many interesting questions about fairness. Use your simulator to explore fairness in an Ethernet. For example, you could add code to your simulator to measure the degree to which the Ethernet protocol is fair. You could do this by measuring the degree to which the Ethernet capture effect skews the frequency with which computers take turns using an Ethernet. Alternately, you could look at whether Ethernet allocates bandwidth fairly to stations at different physical locations on the network (i.e., do computers at the ends of the network get more or less bandwidth than those in the middle?).

2. The data collected by Boggs, et.al. makes it clear that the formula for utilization provided by Metcalfe and Boggs in the original Ethernet paper,

$$\frac{\frac{P}{C}}{\frac{P}{C} + WT}$$

does not accurately predict the behavior of real Ethernets, particularly when P is small. In our discussions, we considered two possible explanations for this inaccuracy: the Ethernet capture effect and the inaccuracy of the assumption that the slot time T accurately described the length of both idle slots and collisions. You could investigate the significance of each of these factors with your simulator. One advantage of using a simulator rather than a real network is that you can very easily evaluate the difference between the rate of collisions produced using the actual exponential backoff algorithm and the rate that would result if each station actually transmitted with probability $\frac{1}{Q}$ in each slot. It will be a bit more challenging to measure the difference between T and the actual duration of average collisions because different stations see the impact of a collision at different times.

3. In one of the questions for the week when we read and discussed the Myths and Reality paper, I asked you to look for aspects of the data they reported that seemed questionable. With your simulator, you could repeat one of these measurements and either determine that their data conflicts with what you obtain with your simulator or explain why our expectations about the data were incorrect. For example, my favorite anomaly was the way in which "excess delay" increased with packet size. I don't see any reason that the time spent in collisions and idle slots should vary with packet size (beyond very small packets where interrupt processing delays play a major role). Collect your own data on excess delay and see what happens.

4. In our discussion of the "Idle Sense" paper by Heusse, et.al., I tried to suggest that the basic approach they proposed could be applied to both WiFI networks and Ethernets. You could evaluate their technique by replacing the exponential backoff algorithm in your simulator with a backoff procedure based on idle sense and see whether it performs better or worse than exponential backoff.

Teamwork and Scheduling

You are strongly encouraged to work on this project in pairs. In fact, I will require that anyone who wants to work alone discuss their reasons with me and get my permission. Each team should approach the project by first sketching out a design for the modules that will make up the final program at a sufficient level of detail to enable you to a) clearly assign programming tasks to individual team members and b) allow team members to work on these task as independently as possible.

I don't want to see you sitting around workstations in pairs for the next few weeks. To emphasize the design process, I want you to try to separate design from programming to the point where you can do a significant amount of the programming independently of the other member of your team.

I would like you all to form groups and inform me who you will be working with by our next meeting. By Monday of the week before Thanksgiving (11/23), you should turn in a sketch of the design of your simulators including type definitions and function/method headings specifying the interfaces between the modules of the simulators and a description of which components each team member will be responsible for implementing.

The plan is to devote our final meetings during the week of 12/10 to the discussion/presentation of your results. While this will include a discussion of your coding experience, the focus should be on the data your obtain. In particular, your code and graphs showing data comparable to that found in Figures 3-5 and 3-7 of the "Myths and Reality" paper will be due the Monday before our discussions or the code (12/7). This will ensure that you have the rest of that week to spend running experiments and analyzing data rather than debugging code. For your meetings on 12/10 and 12/11 you should prepare a short, typed report including and discussing the data you have gathered.

You can write your simulator in the language of your choice, with the warning that execution speed will be important if you want to simulate enough network activity to get statistically meaningful results. This means you probably should use C, C++, or Java, but not Ruby or Perl.

Some guidance

As you approach the design of your simulators, there are many suggestions I would like to give you to aim you in the right direction.

Events

The key to your simulator will be the ability to schedule events to occur in the simulated future and to handle scheduled events when they should occur. You will need a priority queue data structure in which events can be kept ordered by the time at which they should occur. Your simulator's main loop will repeatedly remove the event with the smallest scheduled time from this queue, set the current time equal to the time at which this event was supposed to occur, and then "handle" the event by executing a method/function whose code describes how the system changes as a result of the event (a process which will typically include the scheduling of additional events for the future).

Each event will be represented by a structure/object containing the time at which it will occur and a description of the event. Most events will describe things like the arrival of the preamble of a packet at a particular node. In this case, references to objects describing the node and the packet will be appropriate components of the event description.

If you choose to implement your simulator in C++, Java or any other object-oriented language, a natural way to handle events is to have separate classes for each type of event that all extend a common base class or implement an event interface. All event classes will implement a method that performs the actions that should occur when its type of event occurs. It will also be very useful to have a "toString" method for each event that can be invoked to obtain a message describing the event that could be displayed in an event log while debugging. Such messages should typically include the time the event occurred, the node where it occurred, and the number of the packet involved (if any). It is worth taking time to design a logging mechanism that can easily be turned on or off based on whether you are debugging or running your completed program to collect data.

If you choose to implement in C, you will have a single event struct type which includes an integer field encoding the event type. You will then have an array of function pointers associating the event type code with the function that handles the event.

The Event Queue

As mentioned above, events will be kept in a priority queue ordered by the time at which they are supposed to occur. The efficiency of the implementation of this structure will be critical to the efficiency of your simulator. Obviously, you will need operations to remove the earliest event from this data structure and to insert a new event. You will probably also want an operation to remove an arbitrary event. As an example, when a node begins transmitting a packet it will probably schedule an event corresponding to the completion of the packet's transmission. If the transmission is involved in a collision, the "completion of transmission" event will have to be aborted by removing it from the event queue.

You can use many possible data structures to store the event queue. Look for one where the expected execution times of the operations you implement all grow as the logarithm of the number of events in the queue.

Time

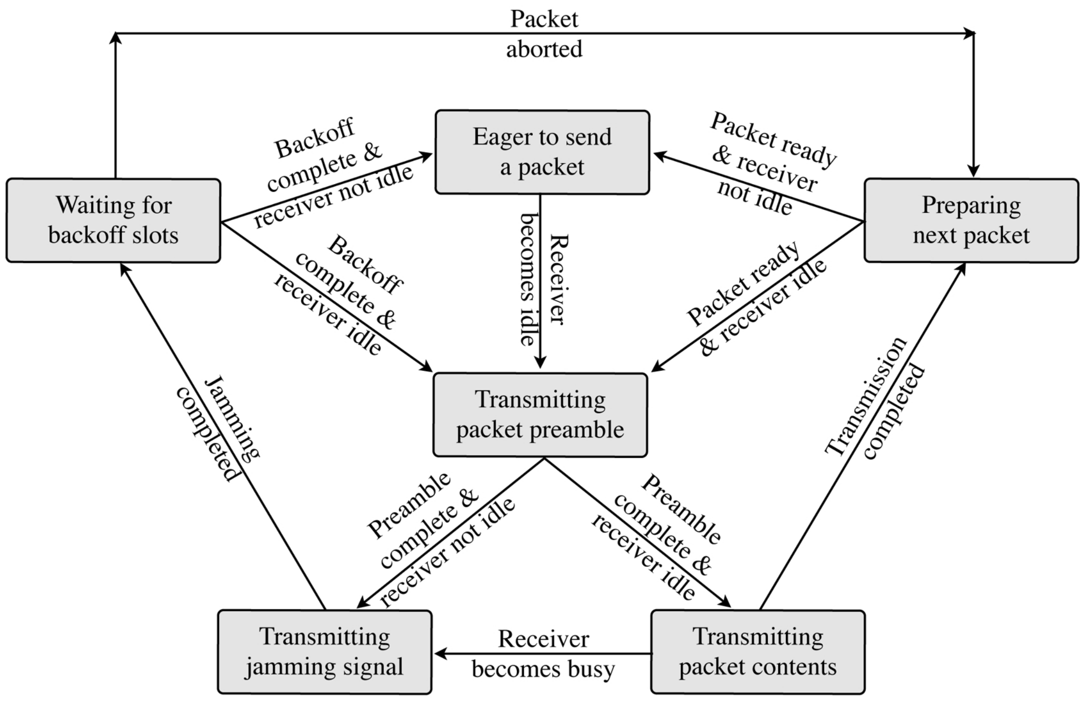
Time should be represented using as much precision as possible. When simulating a 10Mbps networks, it takes 24 bits of accuracy just to measure the passage of a single second with a resolution accurate enough to distinguish one bit time from another. Think “double”.

You may also find it easier to measure time in “bit transmission times” rather than in seconds. That way, an idle slot takes 512 time units rather than $5.12 * 10^{-5}$ seconds.

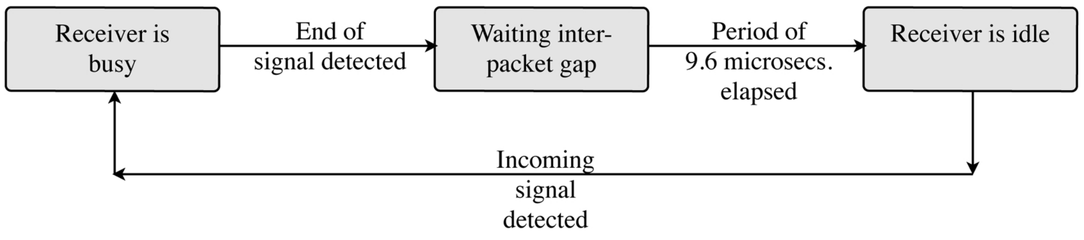
Transmitter and Receiver States

Even though we have discussed how an Ethernet functions quite thoroughly, I think you will find that capturing the protocol in code requires a more precise understanding of the algorithms involved. In particular, you will probably find the two state diagrams below and the explanation of the states that follows helpful as guides when implementing your simulator.

Ethernet Transmitter State Transitions:


Packet aborted
Backoff complete & receiver not idle
Eager to send a packet
Packet ready & receiver not idle
Waiting for backoff slots
Preparing next packet
Backoff complete & receiver idle
Receiver becomes idle
Packet ready & receiver idle
Jamming completed
Transmitting packet preamble
Transmission completed
Preamble complete & receiver not idle
Preamble complete & receiver idle
Transmitting jamming signal
Receiver becomes busy
Transmitting packet contents

Ethernet Receiver State Transitions:



Every node on an Ethernet has a transmitter and a receiver connected to the network cable. The first diagram shows the distinct states the transmitter can be in and the transitions between states that are possible. The second diagram shows the states and transitions for the receiver.

The receiver is clearly the simpler of the two. The main purpose of keeping track of the receiver's state is so that the transmitter will know when to defer to an ongoing transmission (the "CSMA" part of CSMA/CD) and to interrupt transmission when an incoming signal arrives from another node (the "CD" part). For these purposes, the two states that indicate that the receiver is busy or idle are critical. The third state is added to handle the rule that a station must wait for 9.6 microseconds (an interpacket gap) after the network becomes idle before it begins a new transmission.

In your implementation, all receivers will start in the idle state. Your simulator will include events corresponding to the arrival of the first and last bits of a packet at a node. You will use these events to keep a counter of the number of transmissions currently passing by a node on the network. If this counter becomes (or is) greater than 0 while a node is in the idle state, the node will transition to the busy state. If this counter becomes 0 while in the busy state, the node will transition to the "Waiting for interpacket gap" state. It is important to note that the only transition out of this "Waiting" state is to the idle state. When a node enters this state your simulator should schedule an event to move it to the idle state after 9.6 microseconds. If a new transmission arrives while in the waiting state, a node should not move to the busy state immediately. Instead, after the 9.6 microsecond wait elapses, it will move to the idle state, immediately notice that the count of transmissions is greater than 0, and move to the busy state. This is critical because the transition to the idle state will trigger a transition in the transmitter's state that will enable it to begin a transmission, even though this transmission may immediately encounter a collision.

The diagram for the transmitter is clearly a bit more complicated. It has 6 states:

Preparing next packet This is the state used to represent the activity required to respond to the interrupt that occurs at the end of a packet's transmission by getting the next packet ready for transmission and moving it to the Ethernet controller. It accounts for the time delay that made it impossible for a single computer to use 100% of the capacity of the Ethernet used by Bogg, et.al.

When a node enters this state, your simulator should schedule an event to move on to the "Eager" or "Transmitting Preamble" state after an appropriate amount of time has elapsed. Boggs, et.al. never explicitly say how long this time was. You will have to do some creative tuning here. To be most accurate there should probably be some amount of randomness in the processing time.

When a node first becomes active, it should start in this state.

Eager to send a packet This is the state where 1-persistence is implemented. A node in this state has a packet ready to send. It will begin transmission (and transition to the "Transmitting packet preamble") as soon as the receiver becomes "idle" (i.e. after waiting 9.6 microseconds after the network cable becomes idle).

Transmitting packet preamble An eager node will enter this state as soon as the receiver becomes idle and it will stay in this state for the 64 bit times it takes to send the preamble of its packet *regardless of changes in the receiver's state.* That is, a node never detects a collision during a preamble. To implement this, you should simply schedule an event 64 bit times after a node enters this state to figure out what it should do next (continue or abort due to a collision).

In addition to scheduling an event marking the end of the preamble at the sending node, the code you write to handle entering this state should also schedule events at all of the other nodes on the network marking the arrival of the sender's transmission at those nodes. These events should be scheduled at appropriate times based on the network's topology.

Transmitting packet contents If its receiver is still idle when a node completes the transmission of a packet's preamble, it will enter this state. When a node enters this state, your simulator should schedule an event corresponding to the completion of the transmission at a time in the future based on the length of the packet being sent. When this event occurs, the node will transition back to the "Preparing next packet" state. In this case, your simulator should also schedule events corresponding to the arrival of the end of the transmission at other nodes.

There is another way a node may leave this state. If the receiver becomes busy when a node is in this state (or is already busy when the node enters the state), it will transition to the "Transmitting jamming signal" state. If this occurs, your simulator must somehow cancel the event corresponding to the completion of the transmission.

Transmitting jamming signal A node enters this state when it detects a collision while transmitting a packet. It should stay in this state for the time required to send a jamming signal (32 bit times). When this time is complete, it will enter the backoff state. Your simulator will also need to schedule events at other nodes corresponding to the arrival of the end of the transmission (truncated packet plus jamming signal) at other nodes.

Waiting for backoff slots This state represents the condition of a node that is waiting a randomly selected number of idle slots before attempting to retransmit a packet. This is where you will implement most of the exponential backoff algorithm. In particular, when a node enters this state, you should simulate incrementing its backoff counter and choosing a random number of slot times to wait.

Recall that there is a limit on the number of backoff rounds and backoff slots used in Ethernet's exponential backoff algorithm. A node never chooses to delay more than 1023 slot times. That is after the 10th round of backoff, all rounds use 1024 slots. Also after 16 backoff rounds, a packet transmission is aborted.

Node and packets

Hopefully, it won't surprise you that two of the types you will need in your program will be used to represent the nodes in your network and the packets they generate. For each packet, you will want to encode its source, the time at which it was first ready to transmit (i.e. when the node left the "Preparing" state), and its length (in case you want to simulate bimodal distributions). For debugging purposes, assign each packet a sequence number. This will be handy if you need to create log files to isolate bugs.

Each node's representation should include its number (handy for debugging again) and an array containing the propagation time required to reach every other node in the network. You don't really need to know/represent the topology of the network. You just need to know how long it will take a signal to propagate from one node to another. In addition to this static information you will need to keep track of the node's transmitter state, its receiver state, the number of transmission currently passing by, its backoff counter, etc.

Statistical Information

You will basically only need to keep some simple counters for gathering data on the average delay and average load simulated. Note that in their experiments, Boggs, et.al. first let the network nodes "stabilize" for several seconds and then collected data after this stabilization period. You may want to go a bit further and break the process of data collection at each activity level into several shorter periods. This way, you can report both an average over all of the shorter periods and a range (minimums and maximums collected in any one period). This will give you some measure of the uncertainty in your data.

Packet Generation

In their paper, Boggs, et.al. talk about two queues of packets in each node. Within your simulator, such queues should be unnecessary. At any given time there is only one packet of interest at each node — the one it is currently trying to send. As soon as one packet's transmission is complete successfully, a node will generate a new one (incurring a delay that simulates interrupt processing on a Titan). As a result, you don't need to keep a packet queue. You just need to keep information about the packet the node is currently trying to send.