Programming Project 1 — Protocol Simulation

Due: The week of November 5, 2008

In the paper "Contention protocols for ring networks with large bandwidth-delay products," David Feldmeier proposes two unusual medium access control protocols for use in networks with ring topologies.

One is a simple variation on the token ring protocol, which the author refers to as the "contention token" protocol. In this protocol, the stations in the ring circulate a token as they would in the standard token ring protocol. When a station has a packet to transmit, however, it is not required to patiently wait for the token. Instead, a node with a packet of its own to send is allowed to begin transmission whenever it is not already busy forwarding another station's packet. This, of course, opens the possibility that two stations will end up transmitting simultaneously leading to collisions. Feldmeier proposed mechanisms for resolving such collisions designed to yield the advantages of high efficiency at high loads associated with the standard token ring.

Feldmeier also proposes a token-less contention based protocol for rings. In this second protocol, he relies on randomly generated priorities associated with packets to resolve collisions efficiently.

To evaluate the behavior of these protocols, Feldmeier constructed computer simulations of rings based on his protocols. For this project I would like you to duplicate (or perhaps even extend) part of his simulated experiments).

While the primary purpose of this assignment is to make you familiar with the construction of a discrete event simulator, I also want you to approach the task of reproducing Feldmeier's experiments critically. Do not assume the correctness of his data (or yours). In particular, on page 117 he states that each node buffers packet headers before forwarding them. He never says exactly how big the header is, but Figure 3 suggests it is at least 5 bytes or 40 bits. The rings he simulates have 100 stations. This would imply that rings based on his protocols should have a buffer delay that could be larger than 4000 bits while a standard token ring would only require 100 bits worth of buffer delay for the same number of stations. Somehow, however, he shows round trip times as low as 2000 time units for his rings. Think about it.

Teamwork

You are strongly encouraged to work on this project in pair. In fact, I will require that students who wants to work alone discuss their reasons with me and get my permission.

Each pair 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.

Your program should simulate either one of Feldmeier's proposed protocols. Chances are, the contention token protocol will be somewhat simpler to simulate. I will take this into account when grading your programs.

When you are done, you should turn in both your code and a brief report (i.e. ≤ 5 pages) comparing the data on the performance of the protocol produced by your simulators.

I would like you all to inform me who you will be working with by our next meeting. At that meeting, you should also turn in a sketch of the design of your simulator including type definitions and procedure headings specifying the interfaces between the modules of the simulator and a description of which components each team member will be responsible for implementing.

The plan is to devote part of our meetings during the week of 11/1 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 and its comparison to Feldmeier's.

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.

Time

Time should be represented using as much precision as possible. When simulating 100Mbps or Gigabit networks, it takes a lot of bits to measure the passage of a single second. Think "double" or "long".

Node and packets

Hopefully, it won't surprise you that two of the types you will need will be used to represent the nodes in your ring and the packets they generate. For each packet, you will want to encode its source and destination, its creation time, and its length. For debugging purposes, assign each packet a sequence number.

Each node's representation should include a reference to its successor, its sequence number (handy for debugging again) and the propagation time to reach the next node. In addition to this static information you will need to keep track of the node's state (is it sending a packet of its own, idle, forwarding a packet, holding the token, etc.) and of the queue of packets generated by the node and awaiting transmission.

Transmissions

It may not be obvious that you will want a type to represent the transmission of a packet as a separate entity from the packet itself. To see why, think of Feldmeier's contention priority protocol. Each time a packet is transmitted in this protocol a random priority is associated with the packet. If a transmission is aborted due to a collision, you will still need to simulate the propagation of the partial packet around the ring. While the fragment is still circulating, the source node may get to make another attempt to transmit the still pending packet. A different priority may be associated with this second transmission. If one attempted to use the structure that represents the packet itself to represent both the fragment propagating around the ring and the new transmission one might accidentally change the priority associated with the fragment.

Depending on the details of your design, you may encounter other attributes you need to represent that are specific to a particular transmission of a packet rather than associated with the packet itself. For this reason, it is probably best to include "transmissions" as one of the types you implement in your simulator.

Of course, one of the main components of the structure used to represent a transmission will be a reference to the associated packet.

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 to implement a priority queue data structure in which events can be kept ordered by the time at which they should occur. Your simulators 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 (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 header of a transmission of a packet at a particular node. In this case, references to the node and the transmission will be appropriate components of the event description.

When an event is supposed to occur, you will invoke a function/method containing the code to simulate the actions associated with the event. One could do this by storing an integer code in each event structure/object and executing a switch statement using this code to invoke the appropriate function/method. A more direct approach, however, is to either store the address of the appropriate function (if you code in C) or to define each type of event as an extension of some (probably abstract) base class with a "doIt" method customized to simulate the steps that must occur when the event happens.

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 with an incoming packet, the "completion of transmission" event may 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.

A Psuedo-Example

It may help to clear things up if I give you a sketch for what an event handling routing will look like. You will not need to simulate the transmission and reception of every bit of every packet. Instead, you will only have events corresponding to the transmission or reception of significant units within a packet. Thus, you may find you need to perform special actions corresponding to the arrival or transmission of a packets preamble, its header or its trailer.

As our example, then, consider "headerSent", the routine that might be used to simulate the moment at which a station finished sending the header of one of its packets. A sketch of the code that might be used for this routine is shown below.

```
void headerSent(Event e)
{
InsertEvent(headerReceived, e->trans, e->node->nextnode,
currentTime + e->node->proptime);

e->node->nextEvent = InsertEvent(trailerSent, ... );

}
```

The main actions of this routine are to schedule two additional events for the future. One event is the arrival of the header that was just sent at the next node. The other event is the completion of the transmission of the current packet by the node at which the current event occurred. The code

shown assumes that the "insertEvent" routine returns a reference to the new event. In the case of the "trailerSent" event, the code saves the reference to the event descriptor in case this event has to be aborted due to a collision.

Statistical Information

You will basically only need to keep some simple counters for gathering data on the average delay and average load simulated. Feldmeier talks about running his simulator for 10^9 packet transmission times. This may not be practical for you. So, your data is likely to have a higher error component than his. To give you some estimate of the error, collect several sets of measurements for each load level. That is, if you can afford to run 100,000 packets, you might divide these into four subsets of 25,000 each and compute average delay and load figures for each subset. The variance (or lack thereof) between the figures obtained for each subset will give you a rough sense of their reliability.

Packet Generation

The generation of packets is rather simple. As part of your simulators initialization, you can schedule an event corresponding to the generation of a packet at a random time at each node. Then, whenever a packet is generated, the routine that handles the event can schedule another packet generation for the same node at a random time in the future.

To adjust the load, one merely needs to be able to control the expected delay between packets associated with the "random" times in the future this routine choses.

This brings us to the matter of how to randomly generate arrival times for new packets.

In describing his own simulation experiment, Feldmeier notes that "packets arrive according to a Poisson process." In your simulator, however, it will be best to model arrivals using an exponential distribution. To understand why, consider the task of describing the rate at which some randomly occurring events happen. One could talk about the average number of events per unit time. Alternately, one could talk about the average time between two events.

The Poisson distribution is a discrete probability distribution that takes the first approach. That is, it describes the probability of n occurrences of some type of random events in unit time. The exponential distribution takes the second approach. It describes the probability that the interval between two random events will be less than some value.

So, the Poisson and exponential distributions are two ways of describing the same process. For our purposes, it is much more useful to determine when the next packet should arrive than to determine how many should arrive in a given time interval. The next arrival time information can be directly used to schedule an event corresponding to the creation of a new packet.

The Unix and Java libraries include plenty of random number generators (type "man drand48" to read about one you might find useful). Unfortunately, these random number generators tend to produce values that are uniformly distributed over some range rather than exponentially distributed. It is, fortunately, easy to generate exponentially distributed values given a source of uniformly distributed values.

In general, given a probability density function f(y) with cumulative distribution function $F(Y) = P\{y <= Y\}$, if $\{x_i\}$ is a set of uniformly distributed random numbers between 0 and 1, then the values $y_i = F^{-1}(x_i)$ will have the distribution described by f(y) and F(Y).

As noted in the handout I gave you on probability, the probability density function for an exponentially distributed random variable is

$$f(t) = \lambda e^{-\lambda t}$$

for $t >= 0$ where λ is the expected number of events per unit time. The corresponding cumulative distribution function is

$$F(t) = 1 - e^{-\lambda t}$$

giving us

$$F^{-1}(x) = -\frac{\log(1-x)}{\lambda}$$

So, if x's are produced using a built-in, uniform distribution random number generator, plugging the x's into the preceding formula will give you exponentially distributed random values. All you have to do is choose λ appropriately based on your packet size and the load you wish to place on the network.

Efficiency and Memory Allocation

One thing you will discover about your simulator is that it does a lot of computing. As a result, efficiency matters. I've already encouraged you to be aware of this when implementing your event queue. Another place where careful attention to efficiency is appropriate is the dynamic allocation of the objects used to represent packets, events and transmissions.

In most programs, when you need a new object you just invoke the system or language's object creation primitive ("malloc" or "new"). Because these primitives are general purpose, they are fairly costly. You will find that you can speed up your simulator quite a bit by implementing a more specialized system for allocating objects.

The scheme I have in mind is just a form of recycling. For each type of object for which you expect to do a large number of "object creations", keep a linked list of instances of that object type that have been used and discarded. When you need to create a new object, try to take one of the discarded objects off the list for that object type and reuse it. If the list is empty, just do a "malloc" or "new". When the list is non-empty, however, you will save a lot of time by using it.

The cost of this scheme will be that you have to explicitly deallocate objects even if you chose to work in a language with a garbage collector that would have normally made this unnecessary.

Finally, in case you haven't guessed, you should take efficiency into consideration when choosing a programming language for this assignment. C would probably give you the fastest code. C++ would be just about as fast (but you will have to deal with how rusty my C++ is when you explain your code to me). Java would be slower but probably fast enough. ML is probably not a good idea.