MAC Protocol simulation

November 30, 2017

1 Simulating MAC protocols

In this exercise, we simulate MAC several MAC protocols that make different assumptions about availability of knowledge of paket queues at different nodes.

The setup is as follows: * Assume there are a couple of stations sharing a medium * At each station, packets arrive for transmission over the shared medium (the packet's destination is not relevant here) * We make typical assumptions: time is slotted, only one transmission per timeslot * The scheduler has to decide, for each time slot, which station to serve * We look at both centralized and distributed schedulers

1.1 Setup and helper class

In [17]:
```
from collections import deque
import numpy as np
%matplotlib notebook
import matplotlib.pyplot as plt
```

1.1.1 A helper class to model queues

We use the following class to represent a queue at each station. This class adds a packet to the queue with a given probability. It removes a packet from the front of the queue when the station is served. It also has a couple helper functions to keep track of statistics, in particular, average queue length over time and averge waiting time of pacekts in the queue.

In [18]:
```
class ArrivalQueue:

    def __init__(self, rate):
        """Initialize with rate of a Bernoulli
        arrival process."""
        self.rate = rate

        # this is the actual queue, containing packets
        self.queue = deque()

        # just a helper list, to store queue lengths at different points
        # in time - to ease computation of statistics
        self.queuelength = [(0, 0)]
```

```
        self.waitingtime = 0
        self.numberserved = 0

    def arrive(self, now):
        if np.random.uniform() < self.rate:
            self.queue.append(now)
            self.queuelength.append( (now,len(self.queue), ) )

    def serve(self, now):
        try:
            el = self.queue.popleft()
            self.waitingtime += now - el
            self.numberserved += 1
            return 1
        except IndexError:
            return 0

    def stats(self, now):
        self.queuelength.append((now, len(self.queue)))
        try:
            x = sum([
                            (b[0]-a[0])*a[1]
                            for
                                (b, a) in
                            zip(self.queuelength[1:],
                            self.queuelength[:-1])
                            ])
            y = ( self.queuelength[-1][0] - self.queuelength[0][0])

            # print(x, y)
            avgql = x / y

        except ZeroDivisionError:
            avgql = -1

        try:
            avgwaiting = self.waitingtime/self.numberserved
        except ZeroDivisionError:
            avgwaiting = -1

        return (avgql, avgwaiting)
```

1.1.2 Convenience function

```
In [19]: def argmax(sequence):
             m = max(sequence)
             i = sequence.index(m)
             return i
```

1.2 Simulation framework

A couple functions to run the simulation and to visualize the results. Core mechanism is to have scheduling functions as Python generators that can take a value when called with `send`.

```
In [20]: def simulate(scheduler_fct, rates, timesteps=1000):
             """Execute one run of timesteps many time slots, using
             arrival rates as given in rates list.
             Call the scheduler_fct as a generator.
             Returns an array of ArrivalQueues, from which statistics
             can be extracted."""

             size = len(rates)
             queues = [ArrivalQueue(r) for r in rates]
             scheduler = scheduler_fct(len(rates), queues)

             now = 0
             transmit = 0

             serve = scheduler.send(None)
             while now < timesteps:
                 for q in queues:
                     q.arrive(now)

                 serve = scheduler.send(transmit)
                 if serve >= 0:
                     # it would be so much nicer if we could raise exceptions
                     # in a generator
                     # and still continue to use it...
                     transmit = queues[serve].serve(now)

                 # print("serving {} at {}: transmitting: {}".format(serve, now, tr

                 now += 1

             return queues

In [21]: def plot_queues(allqueues, titles=[]):
             """Plot the queue evolution for all runs contained
             in allqueues."""

             f, axarr = plt.subplots(len(allqueues), sharex=True)
             plt.tight_layout()

             # This is just to fix an inconvenient behavior of subplots, which
             # does not always return an iterable:
             if len(allqueues)==1:
                 axarr = [axarr]
```

```
            for ax, queues, title in zip(axarr, allqueues, titles):
                ax.set_title(title)
                for q in queues:
                    ql = q.queuelength
                    ax.step([x[0] for x in ql],
                            [x[1] for x in ql], )

            plt.show()
```

```
In [22]: def run_all(rates, timesteps, allschedulers):
             """Run simulation for multiple schedulers,
             with otherwise identical parameters."""

             allqueues = []

             print("Avg. QL, avg. waiting time")
             for f in allschedulers:
                 queues = simulate(f, rates, timesteps)

                 for q in queues:
                     print(q.stats(timesteps))
                 print("-----------")

                 allqueues.append(queues)

             plot_queues(allqueues,
                         titles=[s.__doc__ for s in allschedulers])
```

1.3 Schedulers

We will look at a couple of schedulers. They are implemented as Python generators, i.e., as infinite loops that produce a new value whenever they are called. A simulation look will initalize the schedulers passing them a pointer to the list of arrival queues. (TODO: perhaps nice to pass a list with the ArrivalQueues.queues?) Then, each call happens when the packet scheduler should decide which queue will be active next. Return that queue index, or a -1 if no queue is sending or a collission would occur.

1.3.1 Round-robin scheduler

Perhaps the easiest one: serve one terminal after the other, irrespective of queue levels. Easy to implement, even in distributed fashion.

```
In [23]: # Note: The scheduling functions are
         # PEP 342 style generators
         # that can accept a send() call

         def roundrobin(size, queues):
             """Round robin"""
             index = 0
```

```
    queues = queues

    while True:
        previous_success = yield index
        # print("previous transmit: {}".format(previous_success))
        index = (index + 1) % size
```

1.3.2 Longest queue first

In a centralized setting, this is an entirely plausible one: serve the longest queue first. No need to do complicated tie breaking if we are not worried about fairness.

```
In [24]: def longest_queue(size, queues):
             """Longest queue"""

             index = 0
             queues = queues

             while True:
                 previous_success = yield index
                 ### BEGIN SOLUTION
                 tmp = [len(q.queue) for q in queues]
                 ### END SOLUTION
                 index = argmax(tmp)
```

1.3.3 Queue with oldest packet first

Similar idea: serve oldest packet first.

```
In [25]: def oldest_queue(size, queues):
             """Queue with oldest packet"""

             index = 0
             queues = queues

             while True:
                 previous_success = yield index

                 ### BEGIN SOLUTION
                 tmp = [-1 * q.queue[0] if len(q.queue) > 0 else -9999
                        for q in queues ]
                 index = argmax(tmp)
                 ### END SOLUTION
```

1.3.4 Aloha

Pure Aloha is of course trivial to implement. It will, however, even in this simple setting fail at any nontrivial traffic loads if we do not add same simple backoff mechanism. Think about possible approaches.

```
In [26]: def aloha(size, queues):
             # convention: return -1 when there is nothing useful to do
             "Aloha"
             queues = queues
             index = 0

             while True:
                 previous_success = yield index

                 # which queues have at least a single packet?
                 busyqueues = [len(q.queue) > 0 for q in queues]
                 # print(busyqueues)

                 ### BEGIN SOLUTION
                 # how many of them are there?
                 numBusy = busyqueues.count(True)
                 if numBusy >= 1:
                     # randomly pick any of the true values, but do a bit of backo
                     busyindexes = [i
                                    for i, v in enumerate(busyqueues)
                                    if v and (np.random.uniform() < 0.5)]
                     # print(busyindexes)
                     # print("----")
                     if len(busyindexes) == 1:
                         index = busyindexes[0]
                     else:
                         index = -1
                 else:
                     index = -1

                 ### END SOLUTION
```

1.3.5 Complete information

And here is a clever idea, suitable for distributed implementation with good performance characteristics.

Suppose we are in a shared medium where everybody can observe all actions of every other terminal, without incurring any error. Suppose that in the previous time slot, terminal k had the right to send. Let us think about what we then can observe and can deduce about queue length – actually, we know nothing about upper bounds, but wen can deduce something about *an upper bound* on queue lengths:

- For all terminals, we do not know whether a packet has arrived or not in that time slot. Hence, we increment every terminals upper bounds by 1.
- If terminal k did send in the previous slot: we know that it has one packet less in its queue, and can hence reduce its upper bound by 1.
- It terminal k did *not* send in the previous slot, we know that it had *no packet in its queue* at the beginning of that previous slot. Hence, *now* terminal k might have at most a single packet in

its queue, arrived just during this past slot. Hence, the upper bound for terminal k's queue is no 1!

This is the key insight: we have a means of observing that some terminal's queue is (almost) empty. We hence simply do a scheduler based on the *longest upper bound* (rather than longest queue).

The second key point here is that this computation of upper bounds can be done by each terminal separately, and they will all come to the same conclusion! Hence, they all know which terminal has the largest upper bound, and can hence deterministically and consistently decide whose terminal's turn it is to send in this time slot.

Try to implent this scheduler in the code fragement below. The crucial aspect is updating the variable `bounds` based on the information the simulation loop tells it.

```
In [27]: def complete_information(size, queues):
             """Complete Information"""
             index = 0
             queues = queues
             bounds = [0 for i in range(len(queues))]

             while True:
                 previous_success = yield index
                 # previous_success olds information from simulation loop!

                 ### BEGIN SOLUTION
                 # in all queues, one packet MIGHT have arrived:
                 for i in range(len(bounds)):
                     bounds[i] += 1

                 # for the special case of the queue we just served,
                 # we gained additional knowledge:
                 if previous_success:
                     bounds[index] -= 1
                 else:
                     bounds[index] = 1

                 ### END SOLUTION

                 # next queue to serve: queue with smallest bound,
                 # ties broken in order of index (to be deterministic)

                 index = argmax(bounds)
```

1.4 Let's do it!

```
In [30]: # Setup simlaution parameters and run it:

         timesteps = 50000
```

```
        rates = [0.15, 0.15, 0.15]
        rates = [0.1, 0.2, 0.69]


        schedulers_to_run = [longest_queue,
                             oldest_queue,
                             roundrobin,
                             aloha,
                             complete_information,
                            ]

        run_all(rates, timesteps, schedulers_to_run)
```

```
Avg. QL, avg. waiting time
(5.4846, 45.66707218167072)
(5.92444, 24.53126248501798)
(6.357, 7.791269194974407)
-----------
(6.1283, 50.444092618246586)
(11.13336, 51.14282828282828)
(36.52038, 51.47692218485446)
-----------
(1.13696, 1.4192377495462796)
(1.49578, 2.487699568229742)
(8936.03226, 12919.358972820544)
-----------
(4.43506, 33.97917485265226)
(1301.71084, 6517.336414299307)
(13496.40532, 19818.85062683382)
-----------
(16.98592, 159.17932949811546)
(30.98408, 148.22571371927043)
(58.01554, 82.50822674418605)
-----------


<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```