**Discrete Event Simulation of**

**M/M/1 Single Server**

**&**

**802.11 CSMA/CA**

**-**

**Kelvin Lu**

**Joseph McGee**

**-**

**ECS 152A**

**12 Mar 2015**

I. Discrete Event Simulation

The use of discrete event simulation is one analytical tool that can be used to observe the properties of a system under some operation. The process notes particular changes in the state of the system and treats them as events which can be simulated. One may also observe the state and record any observations on it.

In this project, we are particularly interested in using discrete event simulation to observe the behavior of two processes, the M/M/1 queue and CSMA/CA protocol of IEEE 802.11.

For many applications, discrete event simulation is a straightforward way to determine the results of a complex operation for some systems that may not be easily made with statistical analysis.

Creating implementations of processes by discrete event simulation also provides greater insight into the mechanisms behind the systems being simulated. There is a whole other reward for just building the simulation implementation!

II. Phase I Discussion

In Phase I we aim to simulate a typical M/M/1 queue, modeled by an arrival rate parameter and a processing rate parameter. This reduction is used to show how the typical link processor behaves with a queue of packets to process.

Two important points of observation are the mean queue length and utilization rate of the server. By varying the rate at which packets arrive at the queue and the rate at which packets take time to be processed in the server, these two properties change.

Another property not easily calculated is the number of packets dropped in a finite queue, which can be easily recorded and shown in a simulation.

Along with simulations and modeling, we can also apply statistical methods to analyze the behaviors of the queue.

We conclude that mean queue length grows exponentially and utilization grows linearly with increasing arrival rates.

III. Phase I Logic

To simulate the M/M/1 queue with discrete event simulation, we simply implement a list of time-based events to process and a queue of packets to process.

We generate events to represent two actions; the arrival of a packet to the system and the departure of a packet from the system. The timing of these events are determined by statistical distributions.

Events are stored in a global event list, implemented as a linked list in the simulation code. As we process events by running the simulation, we note the “current” time and perform the necessary actions to modify that state of the simulation. We also record statistics for later analysis.

IV. Phase II Description

Phase II is concerned with the behavior of a CSMA/CA-like protocol

V. Phase II Implementation

VI. Phase II Code

import sys

import Queue

import random

import math

DEBUG = False

# Constants

# Time to transmit a maximum 1500 (+44) byte MTU frame over a 11 Mbps network

DATA\_FRAME\_MAX\_TRANSMISSION = 0.00112

# Time to transmit a 20 (+44) byte ack frame over a 11 Mbps network

ACK\_FRAME\_TRANSMISSION = (64.0 / 1544.0) \* 0.00112

SIFS = 0.00005

DIFS = 0.0001

# BACKOFF\_T = 0.001

TIMEOUT\_T = 0.01

BACKOFF\_ATTEMPT\_LIMIT = 10

# Maths and distributions

def NEG\_EXP(PARAM):

# random.random is uniform over [0.0, 1.0)

# math.log is base-e with no base kwarg

return (-1.0 / PARAM) \* math.log(1.0 - random.random())

# Simlulation constructs

class Event(object):

def \_\_init\_\_(self, event\_time, host\_id):

self.event\_time = event\_time

self.host\_id = host\_id

def \_\_cmp\_\_(self, other):

return cmp(self.event\_time, other.event\_time)

class FrameArrival(Event):

def \_\_init\_\_(self, event\_time, host\_id):

super(FrameArrival, self).\_\_init\_\_(event\_time, host\_id)

class TransmissionAttempt(Event):

def \_\_init\_\_(self, event\_time, host\_id):

super(TransmissionAttempt, self).\_\_init\_\_(event\_time, host\_id)

class TransmissionStart(Event):

def \_\_init\_\_(self, event\_time, host\_id):

super(TransmissionStart, self).\_\_init\_\_(event\_time, host\_id)

class TransmissionCompletion(Event):

def \_\_init\_\_(self, event\_time, host\_id, frame):

super(TransmissionCompletion, self).\_\_init\_\_(event\_time, host\_id)

self.frame = frame

class TransmissionTimeout(Event):

def \_\_init\_\_(self, event\_time, host\_id, frame):

super(TransmissionTimeout, self).\_\_init\_\_(event\_time, host\_id)

self.frame = frame

class Frame(object):

def \_\_init\_\_(self, transmission\_time, bytes, source\_host\_id, dest\_host\_id):

self.transmission\_time = transmission\_time

self.bytes = bytes

self.source\_host\_id = source\_host\_id

self.dest\_host\_id = dest\_host\_id

self.corrupted = False

class DataFrame(Frame):

def \_\_init\_\_(self, PARAM\_MU, source\_host\_id, dest\_host\_id, creation\_time):

r = NEG\_EXP(PARAM\_MU)

r = 1.0 if r > 1.0 else r

transmission\_time = DATA\_FRAME\_MAX\_TRANSMISSION \* r

bytes = math.floor((1544 \* r) + 1)

super(DataFrame, self).\_\_init\_\_(transmission\_time, bytes, source\_host\_id, dest\_host\_id)

self.creation\_time = creation\_time

def \_\_repr\_\_(self):

return '<DataFrame {0} ({1} to {2})>'.format(id(self), self.source\_host\_id, self.dest\_host\_id)

class AckFrame(Frame):

def \_\_init\_\_(self, source\_host\_id, dest\_host\_id, data\_frame):

transmission\_time = ACK\_FRAME\_TRANSMISSION

super(AckFrame, self).\_\_init\_\_(transmission\_time, 64, source\_host\_id, dest\_host\_id)

self.data\_frame = data\_frame

def \_\_repr\_\_(self):

return '<AckFrame {0} ({1} to {2}, acknowledges {3})>'.format(id(self), self.source\_host\_id, self.dest\_host\_id, self.data\_frame)

class Host(object):

def \_\_init\_\_(self, host\_id, network, PARAM\_MU):

self.host\_id = host\_id

self.network = network

self.PARAM\_MU = PARAM\_MU

self.frame\_queue = Queue.Queue(maxsize=0)

self.ack\_queue = Queue.Queue(maxsize=0)

self.resend\_queue = Queue.Queue(maxsize=0)

self.sent\_frames = []

self.backoff = 0.0

self.is\_backing\_off = False

self.unsuccessful\_attempts = 0

def create\_arrival\_event(self, current\_time):

return FrameArrival(current\_time + NEG\_EXP(self.PARAM\_MU), self.host\_id)

def resend\_frame(self, frame):

self.resend\_queue.put(frame)

def enqueue\_frame(self, frame):

if (type(frame) == AckFrame):

self.ack\_queue.put(frame)

else:

self.frame\_queue.put(frame)

def dequeue\_frame(self):

if (not self.ack\_queue.empty()):

return self.ack\_queue.get()

elif (not self.resend\_queue.empty()):

return self.resend\_queue.get()

elif (not self.frame\_queue.empty()):

frame = self.frame\_queue.get()

self.sent\_frames.append(frame)

return frame

else:

return None

def has\_frame\_to\_send(self):

return (not self.resend\_queue.empty()) or (not self.frame\_queue.empty()) or (not self.ack\_queue.empty())

def acknowledge(self, frame):

self.sent\_frames.remove(frame)

def start\_backoff(self):

self.is\_backing\_off = True

self.unsuccessful\_attempts += 1

self.backoff = random.random() \* self.unsuccessful\_attempts \* BACKOFF\_T

def stop\_backoff(self):

self.is\_backing\_off = False

def reset\_backoff(self):

self.is\_backing\_off = False

self.unsuccessful\_attempts = 0

self.backoff = 0

def decrement\_backoff(self, backoff\_dec):

if self.backoff > 0 and self.is\_backing\_off:

self.backoff -= backoff\_dec

def get\_backoff(self):

if self.is\_backing\_off:

return self.backoff

else:

return None

class Network(object):

def \_\_init\_\_(self, N, PARAM\_LAMBDA, PARAM\_MU):

self.N = N

self.PARAM\_LAMBDA = PARAM\_LAMBDA

self.PARAM\_MU = PARAM\_MU

self.time = 0.0

self.transmitting = []

self.hosts = dict((i, Host(i, self, self.PARAM\_LAMBDA)) for i in xrange(N))

self.events = Queue.PriorityQueue(maxsize=0)

for i, host in self.hosts.iteritems():

self.events.put(host.create\_arrival\_event(0.0))

self.statistics = {

'bytes\_sent': 0,

'delay': 0.0,

'packets\_sent': 0,

}

def simulate(self, limit):

event\_n = 0

while((not self.events.empty()) and (event\_n < limit)):

event\_n += 1

event = self.events.get()

event\_type = type(event)

diff\_time = event.event\_time - self.time

# Reduce the backoff of all hosts if idle

rewind\_time\_backoff = False

if not self.transmitting:

backing\_off\_hosts = [host for host in self.hosts.values() if host.is\_backing\_off]

backoffs = [host.get\_backoff() for host in backing\_off\_hosts]

min\_backoff = min(backoffs + [diff\_time])

for host in backing\_off\_hosts:

host.decrement\_backoff(min\_backoff)

if (host.get\_backoff() == 0.0) and (host.is\_backing\_off):

rewind\_time\_backoff = True

if (host.unsuccessful\_attempts < BACKOFF\_ATTEMPT\_LIMIT):

host.stop\_backoff()

self.events.put(TransmissionStart(self.time + min\_backoff, host.host\_id))

else:

host.reset\_backoff()

dropped\_frame = host.dequeue\_frame()

host.sent\_frames.remove(dropped\_frame)

self.events.put(TransmissionAttempt(self.time + min\_backoff, host.host\_id))

if DEBUG:

print 'Host {0} has reached maximum unsuccessful attempts. Dropping frame.'.format(host.host\_id)

if rewind\_time\_backoff:

self.time += min\_backoff

self.events.put(event)

continue

self.time = event.event\_time

if DEBUG:

transmitting\_hosts = [frame.source\_host\_id for frame in self.transmitting]

print 'TIME: {time}s\t{transmitting\_hosts}'.format(time=self.time, transmitting\_hosts=transmitting\_hosts)

if (event\_type == FrameArrival):

# Create next frame arrival event

self.events.put(self.hosts[event.host\_id].create\_arrival\_event(self.time))

# Choose a random neighbor and create a data frame for transmission

destination\_host\_ids = [i for i in xrange(self.N)]

destination\_host\_ids.remove(event.host\_id)

destination\_host\_id = random.choice(destination\_host\_ids)

new\_data\_frame = DataFrame(self.PARAM\_MU, event.host\_id, destination\_host\_id, self.time)

self.hosts[event.host\_id].enqueue\_frame(new\_data\_frame)

self.events.put(TransmissionAttempt(self.time, event.host\_id))

if DEBUG:

print 'A new data frame has been generated at host {0}. Its content will take {1}s to transmit.'.format(event.host\_id, new\_data\_frame.transmission\_time)

if (event\_type == TransmissionAttempt):

host = self.hosts[event.host\_id]

if host.has\_frame\_to\_send():

if not self.transmitting:

self.events.put(TransmissionStart(self.time + DIFS, event.host\_id))

elif not host.is\_backing\_off:

host.start\_backoff()

if DEBUG:

print 'Host {0} has entered backoff with {1} unsuccessful attempts.'.format(host.host\_id, host.unsuccessful\_attempts)

if DEBUG:

print 'Host {0} has the intent of transmitting a frame.'.format(host.host\_id)

if (event\_type == TransmissionStart):

# We dont't care what's happening, start transmitting

frame = self.hosts[event.host\_id].dequeue\_frame()

if frame in self.transmitting:

raise Exception('Frame sent to network twice')

transmission\_time = frame.transmission\_time

self.events.put(TransmissionCompletion(self.time + transmission\_time, event.host\_id, frame))

self.transmitting.append(frame)

if (type(frame) == DataFrame):

self.events.put(TransmissionTimeout(self.time + TIMEOUT\_T, event.host\_id, frame))

if (len(self.transmitting) > 1):

for frame in self.transmitting:

frame.corrupted = True

if DEBUG:

print 'Corruption has occured.'

if DEBUG:

print 'Host {0} has begun to transmit {1} from host {2} to host {3}'.format(event.host\_id, frame, frame.source\_host\_id, frame.dest\_host\_id)

if (type(frame) == AckFrame):

print 'Host {0} has unacknowledged frames {1}.'.format(frame.dest\_host\_id, self.hosts[frame.dest\_host\_id].sent\_frames)

print 'Host {0} has frame\_queue {1}.'.format(frame.dest\_host\_id, self.hosts[frame.dest\_host\_id].frame\_queue.queue)

print 'Host {0} has resend\_queue {1}.'.format(frame.dest\_host\_id, self.hosts[frame.dest\_host\_id].resend\_queue.queue)

if (event\_type == TransmissionCompletion):

host = self.hosts[event.host\_id]

frame = event.frame

if (frame == None):

if DEBUG:

print 'last\_frame for completing transmission on host {host} is over'.format(host=host.host\_id)

self.transmitting.remove(frame)

frame\_type = type(frame)

if ((frame\_type == DataFrame) and (not frame.corrupted)):

# Enqueue an ack frame and attempt to transmit it

# sending host is set to a waiting state

# no frames can be sent until ack releases lock

ack\_response = AckFrame(frame.dest\_host\_id, frame.source\_host\_id, frame)

self.hosts[frame.dest\_host\_id].enqueue\_frame(ack\_response)

self.events.put(TransmissionStart(self.time + SIFS, frame.dest\_host\_id))

self.statistics['delay'] += self.time - frame.creation\_time

self.statistics['packets\_sent'] += 1

if ((frame\_type == AckFrame) and (not frame.corrupted)):

# Acknowledge successful reciept of data frame

# receiving host (which originally sent data) is not waiting

host = self.hosts[frame.dest\_host\_id]

host.acknowledge(event.frame.data\_frame)

host.reset\_backoff()

self.events.put(TransmissionAttempt(self.time, host.host\_id))

self.statistics['bytes\_sent'] += frame.data\_frame.bytes

if DEBUG:

print 'Transmission of {0} has completed. It was{1} corrupted.'.format(frame, '' if frame.corrupted else ' not')

if (event\_type == TransmissionTimeout):

host = self.hosts[event.host\_id]

frame = event.frame

if frame in host.sent\_frames:

host.resend\_frame(frame)

host.start\_backoff()

if DEBUG:

print '{0} from host {1} timed out at {2}s.'.format(frame, host.host\_id, self.time)

print 'Host {0} has entered backoff with {1} unsuccessful attempts.'.format(host.host\_id, host.unsuccessful\_attempts)

print 'Host {0} has backoff of {1}'.format(host.host\_id, host.get\_backoff())

if DEBUG:

print

if \_\_name\_\_ == '\_\_main\_\_':

BACKOFF\_T = 0.001

network = Network(20, 0.6, 0.5)

network.simulate(5000)

avg\_throughput = (network.statistics['bytes\_sent'] \* 8) / network.time

avg\_network\_delay = network.statistics['delay'] / network.statistics['packets\_sent']

print 'Avg. Throughput: {0}bps'.format(avg\_throughput)

print 'Avg. Network Delay: {0}s'.format(avg\_network\_delay)

print

VII. Phase II Results

XII. Phase II Results Analysis