DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

EEE413(18/19) Data Communication and Communication Networks

Lab 2 - Traffic Shaping with Token Bucket Filter EEE413

Lab Report

Student Name	:	Enge Xu
Student ID	:	1821635
Date	:	2018/12/23
Professor	:	Kyeong Soo (Joseph) Kim

ABSTRACT

This paper mainly focuses on introducing how to use python to realize the Traffic Shaping with Token Bucket Filter system. Through two tasks to basically illustrate the feasibility of using the token bucket filter (TBF) traffic shaper based on token and leaky buckets with a periodic on-off traffic and compare it with another way provided. Mainly in order to practice the python programming ability and better understanding of TBF theory.

BACKGROUND

1. Python

Python is an interpreted high-level programming language for general-purpose programming. Created by Guido van Rossum and first released in 1991, Python has a design philosophy that emphasizes code readability, notably using significant whitespace. It provides constructs that enable clear programming on both small and large scales. In July 2018, Van Rossum stepped down as the leader in the language community after 30 years.

Python features a dynamic type system and automatic memory management. It supports multiple programming paradigms, including object-oriented, imperative, functional and procedural, and has a large and comprehensive standard library.

Python interpreters are available for many operating systems. CPython, the reference implementation of Python, is open source software and has a community-based development model, as do nearly all of Python's other implementations.

Python's developers strive to avoid premature optimization, and reject patches to non-critical parts of CPython that would offer marginal increases in speed at the cost of clarity. When speed is important, a Python programmer can move time-critical functions to extension modules written in languages such as C, or use PyPy, a just-in-time compiler. Cython is also available, which translates a Python script into C and makes direct C-level API calls into the Python interpreter.

An important goal of Python's developers is keeping it fun to use. This is reflected in the language's name—a tribute to the British comedy group Monty Python—and in occasionally playful approaches to tutorials and reference materials, such as examples that refer to spam and eggs (from a famous Monty Python sketch) instead of the standard foo and bar.

2. Token Bucket Theory

The token bucket algorithm is based on an analogy of a fixed capacity bucket into

which tokens, normally representing a unit of bytes or a single packet of predetermined size, are added at a fixed rate. When a packet is to be checked for conformance to the defined limits, the bucket is inspected to see if it contains sufficient tokens at that time. If so, the appropriate number of tokens, e.g. equivalent to the length of the packet in bytes, are removed ("cashed in"), and the packet is passed, e.g., for transmission. The packet does not conform if there are insufficient tokens in the bucket, and the contents of the bucket are not changed. Non-conformant packets can be treated in various ways: 1. They may be dropped. 2. They may be enqueued for subsequent transmission when sufficient tokens have accumulated in the bucket. 3. They may be transmitted, but marked as being non-conformant, possibly to be dropped subsequently if the network is overloaded.

A conforming flow can thus contain traffic with an average rate up to the rate at which tokens are added to the bucket, and have a burstiness determined by the depth of the bucket. This burstiness may be expressed in terms of either a jitter tolerance, i.e. how much sooner a packet might conform (e.g. arrive or be transmitted) than would be expected from the limit on the average rate, or a burst tolerance or maximum burst size, i.e. how much more than the average level of traffic might conform in some finite period.

There is, however, another version of the leaky bucket algorithm. This is a special case of the leaky bucket as a meter, which can be described by the conforming packets passing through the bucket. The leaky bucket as a queue is therefore applicable only to traffic shaping, and does not, in general, allow the output packet stream to be bursty, i.e. it is jitter free. It is therefore significantly different from the token bucket algorithm.

These two versions of the leaky bucket algorithm have both been described in the literature under the same name. This has led to considerable confusion over the properties of that algorithm and its comparison with the token bucket algorithm. However, fundamentally, the two algorithms are the same, and will, if implemented correctly and given the same parameters, see exactly the same packets as conforming and nonconforming.

TASK INTRODUCTION

Based on the sample code for the object-oriented implementation of M/M/1 queueing system, you are to implement a TBF traffic shaper and a packet generator for the periodic on-off
traffic shown in Fig. 1.
We assume the following for simulations:

- There is no propagation delay.
- Transmission rates are set to 10MBs−1 for both packet generator and TBF.
- The packet generator generates 1000-B packets back to back during the on period

(i.e., no gap between packets).

• In the simulations, arrivals are discrete (i.e., not using the fluid model):

When a packet arrives at the TBF,

– If the amount of tokens in the token bucket is equal to or greater than the packet size, it can pass the TBF immediately.

– Otherwise, the packet should wait until there are enough tokens generated.

• TBF parameter values are set to those given by the analysis based on the fluid model in Sec. I. During the simulation, the delay of each packet should be displayed on the screen and also recorded at the packet sink; the average delay is calculated at the end of the simulation.

If the average packet delay from the simulation in 1) is two times of the packet transmission

time (i.e., 0.2 ms=2 × (1000 B/10 MB s − 1) × (1000 ms/1 s)), we can conclude that there has been no traffic shaping at the TB otherwise, it must be that some packets have been delayed at the TBF due to traffic shaping. Answer the following based on your simulation results from 1): • What is your conclusion (i.e., the on-off traffic passing through the TBF with or without shaping)? • If packets are shaped at the TBF, explain why. Also, find TBF parameters allowing packets to pass through the TBF and show that they work (i.e., no shaping) through another simulation with the newly-designed parameters.

METHODOLOGY

TASK 1

In the first task, the most important sentence is ‘There is no propagation delay’. Which means that when the packet generation rate is the same as the token generation rate. And last but not least, the token bucket size is larger than the packet size. There is no potential delay only in this way.

In this situation, both the set packet generator and the token generator will be like as below;

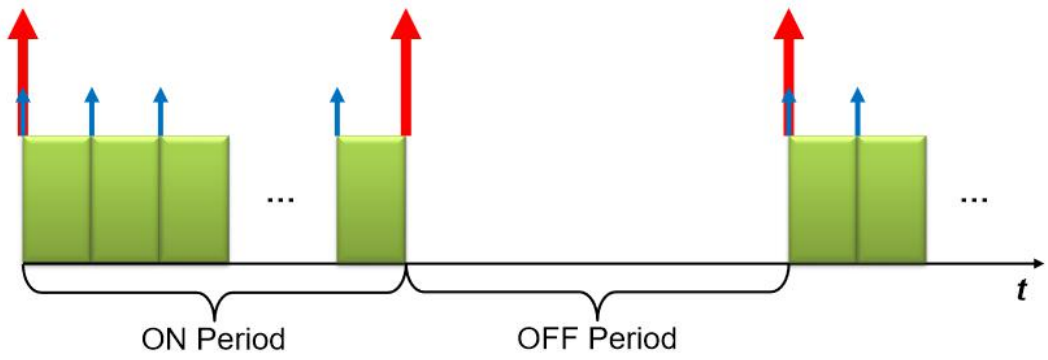

Figure 1

Because packet generator generates packet every 0.1ms, which can satisfy the request of the task 'The packet generator generates 1000-B packets back to back during the on period (i.e., no gap between packets) and in the simulations, arrivals are discrete (i.e., not using the fluid model)' And the interval between two blue up arrow represents 0.1ms. The on period and the off period all last 1s.

Also, the token generator can also in this way represent like this. Because the rate of both is the same. Once the token in the bucket is consumed by the packet(There is still enough token in the bucket), the new token can be generated in the same rate. It can make sure every time the packet comes, the bucket is full and can be consumed by the packet. In this way, the token bucket can operate in the normal way(the packet is not large enough to break the TBF).

Therefore, it can be imagining that the average waiting time(average delay time) can be zero.

TASK 2

In order to solve this problem, it is necessary to set several parameters to satisfy the no shape filter.

First of all, in order to satisfy the average packet delay from the simulation in 1) is two times of the packet transmission time (i.e., 0.2 ms=2 × (1000 B/10 MB s − 1) × (1000 ms/1 s)), the packet transmission time is fixed as 0.1ms.

Secondly, from the results of 1) and 2) in the coursework requirement, we obtain the minimum token bucket capacity of 5 KB. Set the capacity is 5KB.

Thirdly, we can get the equation:

(Packet_Size - Bucket_Capacity) / Token_generate_rate = Average_packet_delay

In this equation, Bucket_Capacity and Average_packet_delay is fixed, then set Packet_Size as 10KB. It can be calculated that Token_generate_rate is $5*10^7$ b/s.

Therefore, the packet generate rate can also be calculated as $1*10^8$ b/s.

By using these parameters. It can be seen that average two packet generate, and only one can be served. Therefore, though the packet generator is in an on-off traffic model like in the task 1. The token is always generating.

Therefore, it can conclude that there has been no traffic shaping at the TBF. Otherwise, it must be that some packets have been delayed at the TBF.

It can be concluded that when the average packet delay from the simulation in 1) is two times of the packet transmission, the token generator is always generating token and it is meaningless of the bucket, because it is not important in this transmission. It always delivers packets at maximum speed(This is the answer for question1).

For question 2, if packets are shaped at the TBF, explain why. By combining the lecture content and the task content, it can be seen that the packet is recognized as can be divided into small bits like fluid. And therefore it can be transmitted by token. If the token bucket always generates token. Then TBF image can not represent by the token generator. So, on the contrary, it can be said that it must be that some packets have been delayed at the TBF.
The parameters are shown as below;

Packet transmission time = 0.1ms;
Bucket Capacity = 5KB;
Packet Size = 10KB;

Token generate rate = $5*10^7$ b/s;

Packet generate rate = $1*10^8$ b/s.

CONCLUSION

Through this experiment, I have mastered the basic method for simple python programming, and for related functions also have a deep understanding, and can skilled application. The most important, is to master the internal principle of queueing theory and Little's theorem. Fundamentally find the changes between different code and function, instead of limit to the surface of function application
.
At the same time, in the experimental process, it also appeared many problems, such as the influence of parameters on the results, similar functions, differences, etc.. But through reading related books, search information, constantly test, I solved problems one by one. It is a good exercise of finding the problem, the ability to solve the problem.

Finally, thanks to the teacher giving us such an opportunity to learn, exercise, guidance. In the learning process in the future, I will make persistent efforts, fight for better complete of the task.

CODE

Sorry in advance for the incomplete code and it actually can not run myself. It spent lots of time to debug the code in different ways: using global variables, using import time, using self function in many ways and so on. Also it costs long time to search the use of several yield function using in simpy and with request as function and so on.

Therefore the code of task 1 of mine is shown as below;

```
import argparse
import numpy as np
import simpy
import time

class Packet(object):
    """
    Parameters:
    - ctime: packet creation time
    - size: packet size in bytes
    """
    def __init__(self, ctime, size):
        self.ctime = ctime
        self.size = size


class OnoffPacketGenerator(object):
    """Generate fixed-size packets back to back based on on-off status.

    Parameters:
    - env: simpy.Environment
    - pkt_size: packet size in bytes
    - pkt_ia_time: packet interarrival time in second
    - on_period: ON period in second
    - off_period: OFF period in second
    """
    def __init__(self, env, pkt_size, pkt_ia_time, on_period, off_period,
                 trace=False):
        self.env = env
        self.pkt_size = pkt_size
        self.pkt_ia_time = pkt_ia_time
        self.on_period = on_period
        self.off_period = off_period
        self.trace = trace
        self.out = None
```

```
        self.on = True
        self.gen_permission = simpy.Resource(env, capacity=1)
        self.action = env.process(self.run())  # start the run process when an instance is created

    def run(self):
        env.process(self.update_status())
        while True:
            with self.gen_permission.request() as req:
                yield req
                p = Packet(self.env.now, self.pkt_size)
                self.out.put(p)
                if self.trace:
                    print("t={0:.4E} [s]: packet generated with size={1:.4E} [B]".format(self.env.now, self.pkt_size))
            yield self.env.timeout(self.pkt_ia_time)

    def update_status(self):
        while True:
            now = self.env.now
            if self.on:
                if self.trace:
                    print("t={:.4E} [s]: OFF->ON".format(now))
                yield env.timeout(self.on_period)
            else:
                if self.trace:
                    print("t={:.4E} [s]: ON->OFF".format(now))
                req = self.gen_permission.request()
                yield env.timeout(self.off_period)
                self.gen_permission.release(req)
            self.on = not self.on  # toggle the status


class FifoQueue(object):
    """Receive, process, and send out packets.

    Parameters:
    - env : simpy.Environment
    """

    def __init__(self, env, trace=False):
        self.trace = trace
        self.store = simpy.Store(env)
        self.env = env
```

```
        self.out = None
        self.action = env.process(self.run())
        self._current_amount = 0
        self._last_consume_time = int(time.time())
        self._current_amount = 0
    def run(self):
        while True:
            #now = self.env.now
            #increment = (now - self.ctime) * 1000000
            #self._current_amount = min(increment + self._current_amount, 500)
            msg = (yield self.store.get())
            #if 1000 > self._current_amount
            # TODO: Implement packet processing here.
            yield self.env.timeout(msg.size - 10000000)
            #self._current_amount = 0
            #else:
                # self._current_amount = self._current_amount - self.msg_size

            self.out.put(msg)

    def put(self, pkt):
        self.store.put(pkt)


class PacketSink(object):
    """Receives packets and display delay information.

    Parameters:
    - env : simpy.Environment
    - trace: Boolean

    """
    def __init__(self, env, trace=False):
        self.store = simpy.Store(env)
        self.env = env
        self.trace = trace
        self.wait_times = []
        self.action = env.process(self.run())

    def run(self):
        while True:
            msg = (yield self.store.get())
            now = self.env.now
            now1 = now
```

```
                self.wait_times.append(now - msg.ctime)
                if self.trace:
                    print("t={0:.4E} [s]: packet arrived with size={1:.4E}
[B]".format(now1, msg.size))

    def put(self, pkt):
        self.store.put(pkt)


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument(
        "-S",
        "--pkt_size",
        help="packet size [byte]; default is 100",
        default=1000,
        type=int)
    parser.add_argument(
        "-A",
        "--pkt_ia_time",
        help="packet interarrival time [second]; default is 0.1",
        default=0.1,
        type=float)
    parser.add_argument(
        "--on_period",
        help="on period [second]; default is 1.0",
        default=1.0,
        type=float)
    parser.add_argument(
        "--off_period",
        help="off period [second]; default is 1.0",
        default=1.0,
        type=float)
    parser.add_argument(
        "-T",
        "--sim_time",
        help="time to end the simulation [second]; default is 10",
        default=10,
        type=float)
    parser.add_argument(
        "-R",
        "--random_seed",
        help="seed for random number generation; default is 1234",
        default=1234,
```

```
        type=int)
    parser.add_argument('--trace', dest='trace', action='store_true')
    parser.add_argument('--no-trace', dest='trace', action='store_false')
    parser.set_defaults(trace=True)
    args = parser.parse_args()

    # set variables using command-line arguments
    pkt_size = args.pkt_size
    pkt_ia_time = args.pkt_ia_time
    on_period = args.on_period
    off_period = args.off_period
    sim_time = args.sim_time
    random_seed = args.random_seed
    trace = args.trace
    env = simpy.Environment()
    pg = OnoffPacketGenerator(env, pkt_size, pkt_ia_time, on_period, off_period,
                                        trace)
    fifo = FifoQueue(env, trace)    # TODO: implemente FifoQueue class
    ps = PacketSink(env, trace)
    pg.out = fifo
    fifo.out = ps
    env.run(until=sim_time)

    print("Average waiting time = {:.4E} [s]\n".format(np.mean(ps.wait_times)))
```

USE INSTRUCTION

Firstly, download and install WinPython distribution is a requirement. Secondly, open the lab2.py file and the two subtask is combined in this file. Spyder or Ipython is recommended.