CS4150 Computer Networks Laboratory – Assignment 5

Subhash S – 111801042
Computer Science and Engineering, IIT Palakkad

Traffic Shaping using Token Bucket

The aim is to simulate the token bucket shaping algorithm. All the specifications of the command-line arguments and input arguments are followed as mentioned in the question.

Overview

The reason to use token bucket is to shape the traffic so that the customer and the provider can agree on the *Service Level Agreement (SLA)*. The idea here is we fill the bucket with tokens and when a packet requests to be sent over the network, the algorithm sends the packet only if enough tokens are available, else it waits till enough tokens are available.

Additional Insights

Functions and their usages:

1. *add_tokens_to_bucket(N):* Adds a subset of *N* tokens to bucket until it overflows.
2. *try_to_send(L):* If there are enough tokens in the bucket, it sends the packet and returns true, else it returns false.
3. *simulate_token_bucket:* Simulates the token bucket traffic shaping algorithm.

Program Flow

1. Initialise the token bucket with various quantities like *bucket_size (S), rate (R), last_send_time (L)* and *number of tokens in bucket (N)*.
2. For each packet arriving,
    1. If the packet has arrived when no other packet is waiting to be sent, then we add tokens into the bucket from the last time a packet was sent with the rate R and set the *last_send_time (L)* to the *arrival_time* of the packet.
    2. If a packet has arrived while other packet is waiting for tokens, this packet will be sent after enough tokens have been generated for both the packets, this will be handled implicitly in 2.4..
    3. We try to send the packet by using the *try_to_send* function and passing the *packet_size,* if there are enough tokens to send, we reduce the number of tokens in the bucket by the *packet_size* and send the packet. Here we don't have to update the *last_send_time* since it was already updated in the previous step.
    4. If there are not enough tokens while sending the packet, the packet has to wait till there are enough tokens, since we can be sure that each packets will be sent sequentially, we go to the time in future when there will be enough tokens for the packet to be sent, we send the packet, set *last_send_time* to the time when the packet was sent and set the *number of tokens in bucket* to 0.
    5. We print the *last_send_time, packet_id* and *packet_length.*

Packet scheduling using FIFO queue

The aim is to simulate the FIFO queue packet scheduling algorithm. All the specifications of the command-line arguments and input arguments are followed as mentioned in the question.

Overview

The idea here is we add packets to the queue if there is enough space and send the packet in a controlled manner depending on the output rate of the FIFO queue. If there is not enough buffer space, we discard the packet.

Additional Insights

1. A *Packet* structure which stores the details of the packet. It has a field named *service_time* which tells the time at which the packet completely leaves the queue.
2. The *Queue* structure implements a double ended queue with a singly linked-list.
3. There are also other standard queue functions like *peek_front, peek_back, push* and *pop.*
4. The *service_time* same as the transmission completion time of the packet.

Program Flow

1. Initialise the queue.
2. For each arriving packet,
    1. If the packet size is larger than the queue's maximum capacity, we discard the packet and continue as it can never enter the queue.
    2. If there are packets at the front of the queue that have already been sent over the network before this packet arrived, we *pop* them and print the required details.
    3. If there is enough space for this packet to enter the queue completely, there are two cases to be handled:
        1. If there is an item at the back of the queue, this packet will be sent at the FIFO output rate only after the packet at the back of the queue has been sent.
        2. If there are no items in the queue, this packet can immediately be sent at the FIFO output rate.

        In the former case, we add (*packet_size÷output_rate)* to the *service_time* of the packet at the back of the queue and in the latter case, we add (*packet_size÷output_rate)* to the *arrival_time* of the packet.
    4. Now with the *service_time* obtained in 2.3. we create a new packet using the *createPacket* function and add this packet to the back of the queue.
    5. The steps from 2.1. to 2.4. are repeated till all the packets have been seen.
3. Now there might be packets in the queue, so we *pop* all these packets and print the required details. The transmission completion time is the *service_time.*

Bash script to find the largest rate of token bucket

The aim is to run the traffic through a token bucket traffic shaping algorithm followed by the FIFO queue algorithm and find the maximum rate (R) of the token bucket for which there are no packet losses.

Overview

An important assumption for this test case is that since the rate of the FIFO queue is, 10 we can safely assume that the rate R will surely be between 0 and 10. Due to this assumption, we can perform a binary search between 0 to 10 to find the maximum value at which there are no packet losses.

Program Flow

Binary Search

1. First we count the number of packets in the *arrivals.txt* file and store it in *IN_PACKETS_COUNT* which can be used to test for packet losses.
2. We continuously loop until the lower index is lower than the higher index and do the following,
    1. Find the mid-point between high and low using *bc*, the command *scale=$PRECISION_PLACES* is used to set the precision to 6 decimal places which is followed by the regular arithmetic operation.
    2. If mid-point hasn't changed from the last iteration, we have converged and this is the required rate so we break from the search loop.
    3. We check if the rate (R) equal to the mid-point has packet loss when traffic is run with the above mentioned configuration, If there is a packet loss, we know that we have to search below this value so we set *high=$mid* and if there is no packet loss, there might possibly be higher values without packet loss so we need to search above mid-point, hence we set *low=$mid.* This algorithm keeps running till it fails the condition in 2. or it has converged in 2.2.
3. We echo the value of mid which is a way to return values from bash functions.

Linear Search

The linear search would be useful for some extreme cases where the assumption mentioned in Overview fails but in this case, it holds true and moreover linear search takes a lot of time to converge to the required rate, therefore with a predetermined rate at which packets are lost, binary search can be much faster.

1. Linear search searches for the rate in an inefficient manner by increment the rate by 10^{-6} in every iteration and it checks if there is a packet loss, if there is no packet loss, it moves forward else it breaks out of the loop and returns the current rate at which there is no packet loss.