Leaky Bucket Algorithm

Implementation of Congestion Control Algorithm (Leaky Bucket Algorithm).

Problem Statement

Consider a host `A` connected to a network. Host `A` produces **bursty traffic** (explained below) in the form of variable-length packets that must be transmitted to the receiver through a network.

Suppose Host A sends :

- `10` Mbps data for the first `2` seconds.
- `20` Mbps data for the next `3` seconds.
- No data for the next `5` seconds.
- `15` Mbps data for the next `2` seconds.

This traffic nature is called **bursty traffic**, coming from a source into the network.

Bursty traffic is a **sudden**, **unexpected network volume traffic peak**, and **depression**. It is the data that is transmitted in a short, uneven manner.

It can affect the network adversely and can lead to network congestion.

If many hosts are connected to a network and they keep sending bursty traffic to the network, it can lead to packet loss, and unexpected delays which decrease the quality of service, and packets can get backlogged. This unexpected packet delay and packet loss are termed **network congestion**, or in simple words, the network is said to be congested. It degrades the performance of the whole system.

In practice, the network needs to guarantee **QoS** (Quality of service) to its customers and users. The problem is to design an algorithm that can take up bursty traffic and make it smooth. This problem can be solved in many ways and one of which is the **Leaky Bucket Algorithm**.

Example

Let us take an example to understand what kind of traffic is expected by the network to prevent congestion.

Let us say Host A sends :

- `10` Mbps data for the first `2` seconds.

- `20` Mbps data for the next `3` seconds.
- No data for the next `3` seconds.
- `15` Mbps data for the next `2` seconds.

Now assuming that we fix a maximum transmission rate (number of bits that can be transferred per second from the host to the network) equal to `5` Mbps. Then we want to develop an algorithm that would send `5` Mbps data at a uniform rate, assuming that the data is available.

Example Explanation :

In the above example, a total of (10∗2+20∗3+15∗2)=110 Mb of data is sent over a period of `10` seconds. This traffic was bursty. When we apply the algorithm `5` Mbps of data will be sent over `22` seconds at a fixed rate. In both cases, we will send all `110` Mb of data. In this way, we can guarantee that the network traffic is uniform.

Constraints

- Size of data packet <= Leak Rate of the bucket.
- Leak Rate <= Dedicated Bandwidth of the network to the host.
- Packets arrive one at a time.

Traffic Shaping

Before moving on to the algorithm, let us discuss an important concept called **Traffic Shaping**. Traffic Shaping is a mechanism to control the amount of traffic sent to the network. It means regulating the average rate flow of data to the network to prevent congestion. It maintains a constant traffic pattern that is sent to the network. This constant rate can depend on the network bandwidth limits and the priority of packets sent by the host.

When the connection is made, the host or the sender can negotiate this constant rate with the network provider, at which traffic will be sent to the network. Each network guarantees a dedicated bandwidth for each host.

This can be used to set this limit. Traffic shaping is one of the most common techniques to prevent network congestion. When the network knows what kind of traffic is expected, it can handle it better.

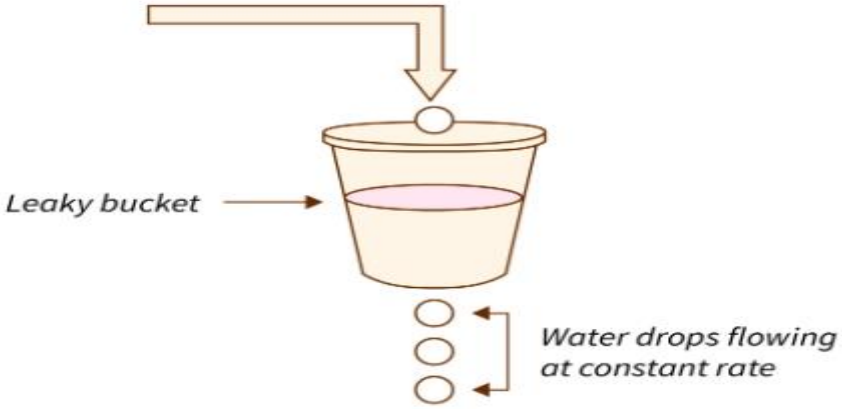
The **Leaky Bucket algorithm** is a traffic shaping algorithm that is used to convert bursty traffic into smooth traffic by averaging the data rate sent into the network.

Leaky Bucket Algorithm

The **leaky bucket algorithm** is a method of congestion control where multiple packets are stored temporarily. These packets are sent to the network at a constant rate that is decided between the sender and the network. This algorithm is used to implement congestion control through traffic shaping in data networks.

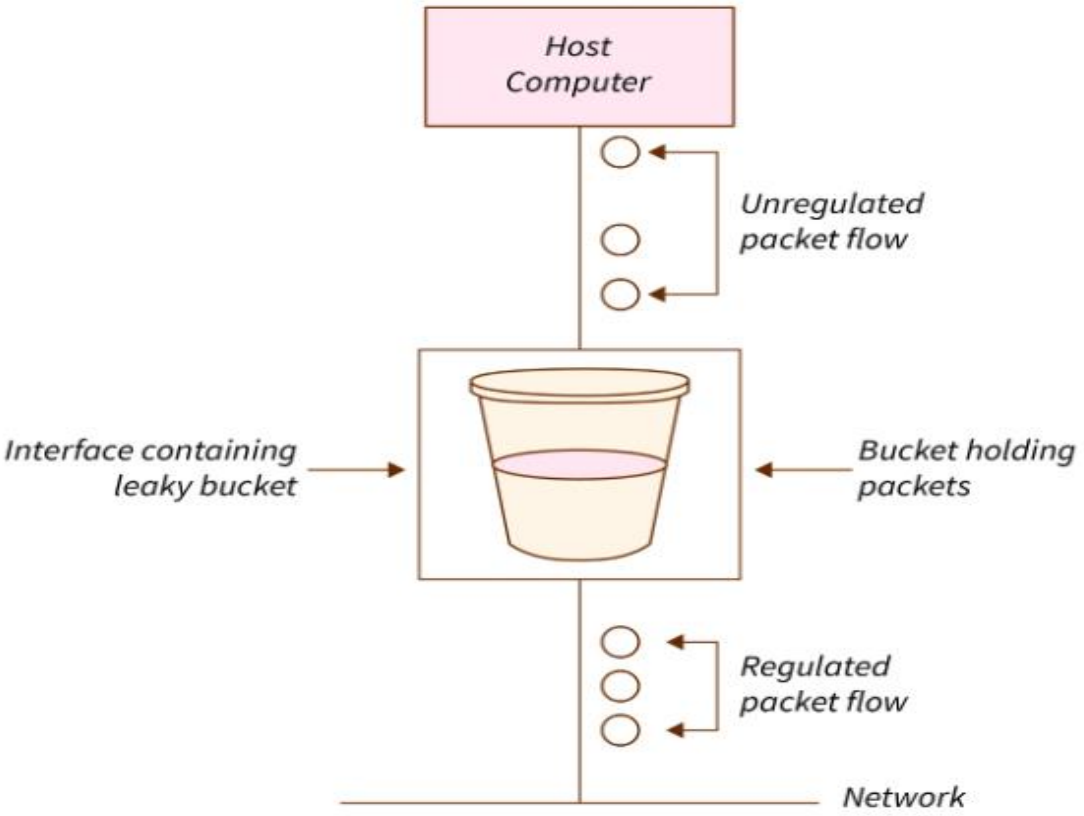
To understand the algorithm, let us first discuss the analogy of a leaky bucket.

Consider a bucket with a small hole at the bottom. Now imagine that water is poured into the bucket at random intervals. At each interval, the amount of water poured into the bucket is not fixed. Now it does not matter how much water is inside the bucket, the water comes out at a constant rate from the hole. Consider the image below for more clarity.



- The rate at which water leaks (called the leak rate) is independent of the amount of water inside the bucket.
- If the bucket becomes full, the water poured will be lost.

The same idea of the **leaky bucket** can be applied to the data packets.



- Consider that, each network interface has a leaky bucket.
- Now, when the sender wants to transmit packets, the packets are thrown into the bucket. These packets get accumulated in the bucket present at the network interface.
- If the bucket is full, the packets are discarded by the buckets and are lost.
- This bucket will leak at a constant rate. This means that the packets will be transmitted to the network at a constant rate. This constant rate is known as the Leak Rate or the Average Rate.
- In this way, bursty traffic is converted into smooth, fixed traffic by the leaky bucket.
- Queuing and releasing the packets at different intervals help in reducing network congestion and increasing overall performance.

This design can be simulated inside the host operating system, specifically in the transport and the network layer.

Implementation

The leaky bucket algorithm can be implemented using a **FIFO** (First In First Out) **queue**. Packets that arrive first in the bucket should be transmitted first.

- A queue acts as a bucket or a buffer to hold the packets. This is implemented in the host operating system.
- Packets from the host are pushed into the queue as they arrive.
- At some intervals, the packets are sent to the network depending upon the leak rate. Generally, this interval is a clock tick. A clock tick is an interrupt generated from the physical clock to the processor.
- This leak rate is predetermined by the network. A network will guarantee some dedicated bandwidth for each host. This dedicated bandwidth can be used to set up this leak rate.
- If this queue is full, then the packets that arrive will be discarded.

Pseudo Code

- **Step – 1 :**
  Initialize buffer size and leak rate.
- **Step – 2 :**
  At clock tick, Initialize `n` as the leak rate.
- **Step – 3 :**
  If the `n` is greater than the size of the packet at front of the queue, send the packet into the network.
- **Step – 4 :**
  Decrement `n` by the size of the packet.
- **Step – 5 :**
  Repeat Step `3` and Step `4` until `n` is less than the size of the packet at the front of the queue or the bucket is empty. No other packet can be transmitted till the next clock tick.
- **Step – 6 :**
  Go to Step 2.

Example

Suppose,
the **buffer size** = `10000` Mb, **leak rate** = `1000` Mb. Assume at some point we have the bucket as [200,500,400,500,100].

Now, in each iteration, we will transfer the packets such that the sum of the size of each packet is not greater than the leak rate.

- **Iteration – 1 :**
- **Bucket** = [200,500,400,500,100]
  We will transfer the first and the second packet into the network. We cannot transfer the third as the sum of (200+500+400) is greater than the leak rate.
- **Iteration – 2 :**
- **Bucket** = [400,500,100]
  We will transfer the next three packets in this iteration as their total size is not greater than the leak rate.
  n=1000. At front of the queue, we have the data packet of size `400`.
  Since 400<= `n`, we will send this packet and update the value of `n` as (1000–400)=600.
- **Bucket** =[500,100]
  Since 500<=n, we will send the packet at the front of the queue. Value of n=(600–500)=100.
- **Bucket** = [100].
  Since 100<=n, we will send the packet at the front of the queue. Value of n=(100–100)=0.
  Hence, after this iteration, our bucket will be empty.
  Now suppose if some new packets arrive, they will be pushed to the end of the queue.

Implementation in C++

Code :

```
#include <bits/stdc++.h>

#include <chrono>

#include <thread>

using namespace std;



class Packet {

  // Packet class to simulate data packets.

  int id, size;

public:

  Packet(int id, int size) {

    // Constructor to initialize packets.

    this->id = id;

    this->size = size;
```

```
    }
    int getSize() {
        return this->size;
    }
    int getId() {
        return this->id;
    }
};
class LeakyBucket {
    // Leaky Bucket class to simulate leaky bucket algorithm.
    int leakRate, bufferSizeLimit, currBufferSize;
    queue<Packet> buffer;
public:
    LeakyBucket(int leakRate, int size) {
        // Constructor to initialize leak rate, and maximum buffer size available.
        this->leakRate = leakRate;
        this->bufferSizeLimit = size;
        this->currBufferSize = 0;
    }
    void addPacket(Packet newPacket) {
        // Function to add new packets at the end of the buffer.
        if(currBufferSize + newPacket.getSize() > bufferSizeLimit) {
            // If the packet cannot fit in the buffer, then reject the packet.
            cout << "Bucket is full. Packet rejected." << endl;
```

```
        return ;
    }
    // Add packet to the buffer.
    buffer.push(newPacket);
    // Update current Buffer Size.
    currBufferSize += newPacket.getSize();
    // Print out the appropriate message.
    cout << "Packet with id = " << newPacket.getId() <<  " added to bucket." << endl;
}
void transmit() {
    // Function to transmit packets. Called at each clock tick.
    if(buffer.size() == 0) {
        // Check if there is a packet in the buffer.
        cout << "No packets in the bucket." << endl;
        return ;
    }
    // Initialize n to the leak rate.
    int n = leakRate;
    while(buffer.empty() == 0) {
        Packet topPacket = buffer.front();
        int topPacketSize = topPacket.getSize();
        // Check if the packet can be transmitted or not.
        if(topPacketSize > n) break;
        // Reduce n by packet size that will be transmitted.
```

```
            n = n - topPacketSize;
            // Update the current buffer size.
            currBufferSize -= topPacketSize;
            // Remove packet from buffer.
            buffer.pop();
            cout << "Packet with id = " << topPacket.getId() << " transmitted." << endl;
        }
    }
};

int main() {
    LeakyBucket* bucket = new LeakyBucket(1000, 10000);
    bucket->addPacket(Packet(1, 200));
    bucket->addPacket(Packet(2, 500));
    bucket->addPacket(Packet(3, 400));
    bucket->addPacket(Packet(4, 500));
    bucket->addPacket(Packet(5, 200));
    while(true) {
        bucket->transmit();
        cout << "Waiting for next tick." << endl;
        std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    }
}
```

Implementation in Java

Code :

```
import java.lang.Thread;

import java.util.*;

class Packet {

 // Packet class to simulate data packets.

 int id, size;

 public Packet(int id, int size) {

  // Constructor to initialize packets.

  this.id = id;

  this.size = size;

 }

 public int getSize() {

  return this.size;

 }

 public int getId() {

  return id;

 }

}

class LeakyBucket {

 // Leaky Bucket class to simulate leaky bucket algorithm.

 int leakRate, bufferSizeLimit, currBufferSize;

 Queue<Packet> buffer = new LinkedList<Packet>();
```

```
public LeakyBucket(int leakRate, int size) {
  // Constructor to initialize leak rate, and maximum buffer size available.
  this.leakRate = leakRate;
  this.bufferSizeLimit = size;
  this.currBufferSize = 0;
}
public void addPacket(Packet newPacket) {
  // Function to add new packets at the end of the buffer.
  if (currBufferSize + newPacket.getSize() > bufferSizeLimit) {
    // If the packet cannot fit in the buffer, then reject the packet.
    System.out.println("Bucket is full. Packet rejected.");
    return;
  }
  // Add packet to the buffer.
  buffer.add(newPacket);
  // Update current Buffer Size.
  currBufferSize += newPacket.getSize();
  // Print out the appropriate message.
  System.out.println(
    "Packet with id = " + newPacket.getId() + " added to bucket."
  );
}
```

```
public void transmit() {
 // Function to transmit packets. Called at each clock tick.
 if (buffer.size() == 0) {
  // Check if there is a packet in the buffer.
  System.out.println("No packets in the bucket.");
  return;
 }
 // Initialize n to the leak rate.
 int n = leakRate;
 while (buffer.size() > 0) {
  Packet topPacket = buffer.peek();
  int topPacketSize = topPacket.getSize();
  // Check if the packet can be transmitted or not.
  if (topPacketSize > n) break;
  // Reduce n by packet size that will be transmitted.
  n = n - topPacketSize;
  // Update the current buffer size.
  currBufferSize -= topPacketSize;
  // Remove packet from buffer.
  buffer.remove();
  System.out.println(
   "Packet with id = " + topPacket.getId() + " transmitted."
  );
 }
```

```
  }
}


class Main {
  public static void main(String args[]) throws InterruptedException {
    LeakyBucket bucket = new LeakyBucket(1000, 10000);
    bucket.addPacket(new Packet(1, 200));
    bucket.addPacket(new Packet(2, 500));
    bucket.addPacket(new Packet(3, 400));
    bucket.addPacket(new Packet(4, 500));
    bucket.addPacket(new Packet(5, 200));
    while (true) {
      bucket.transmit();
      System.out.println("Waiting for next tick.");
      Thread.sleep(1000);
    }
  }
}
```

Implementation in Python

Code :

```
import time

class Packet:
    # Packet class for simulating a packet to be transmitted.
```

```
    def __init__(self, id, size):
        # Initialize class variables.
        self.id = id
        self.size = size

    def getSize(self):
        return self.size

    def getId(self):
        return self.id

class LeakyBucket:
    def __init__(self, leakRate, size):
        self.leakRate = leakRate
        self.bufferSizeLimit = size
        self.buffer = []
        self.currBufferSize = 0

    def addPacket(self, newPacket):
        if self.currBufferSize + newPacket.getSize() > self.bufferSizeLimit:
            # If the packet cannot fit in the buffer, then reject the packet.
            print("Bucket is full. Packet rejected.")
            return
        # Add packet to the buffer.
```

```
    self.buffer.append(newPacket)
    # Update current Buffer Size.
    self.currBufferSize += newPacket.getSize()
    # Print out the appropriate message.
    print("Packet with id = " + str(newPacket.getId()) +  " added to bucket.")


def transmit(self):
    # Function to transmit packets. Called at each clock tick.
    if len(self.buffer) == 0:
        # Check if there is a packet in the buffer.
        print("No packets in the bucket.")
        return


    # Initialize n to the leak rate.
    n = self.leakRate
    while len(self.buffer) > 0:
        topPacket = self.buffer[0]
        topPacketSize = topPacket.getSize()
        # Check if the packet can be transmitted or not.
        if topPacketSize > n:
            break
        # Reduce n by the packet size that will be transmitted.
        n = n - topPacketSize
        # Update the current buffer size.
```

```
            self.currBufferSize -= topPacketSize
            # Remove packet from buffer.
            self.buffer.pop(0)
            print("Packet with id = " + str(topPacket.getId()) + " transmitted.")


if __name__ == '__main__':
    bucket = LeakyBucket(1000, 10000)
    bucket.addPacket(Packet(1, 200))
    bucket.addPacket(Packet(2, 500))
    bucket.addPacket(Packet(3, 400))
    bucket.addPacket(Packet(4, 500))
    bucket.addPacket(Packet(5, 200))
    while True:
        bucket.transmit();
        print("Waiting for next tick.");
        time.sleep(1)
```

Output :

```
Packet with id = 1 added to the bucket.
Packet with id = 2 added to the bucket.
Packet with id = 3 added to the bucket.
Packet with id = 4 added to the bucket.
Packet with id = 5 added to the bucket.
Packet with id = 1 transmitted.
```

Packet with id = 2 transmitted.

Waiting for the next tick.

Packet with id = 3 transmitted.

Packet with id = 4 transmitted.

Waiting for the next tick.

Packet with id = 5 transmitted.

Waiting for the next tick.

No packets in the bucket.

...

Complexity Analysis

Time Complexity

Suppose, at any given time, we have `K` data packets in the queue, waiting for transmission. Now, in the worst case, all these `K` packets can have a maximum size that is equal to the Leak Rate of the bucket. In this case, one packet per clock tick will be transmitted into the network. Hence, the time complexity is O(K).

Time Complexity: O(K) where `K` is the number of data packets in the queue.

Note that if there is no packet, the process will not use any CPU time. It will remain idle.

Space Complexity

In the implementation, we are using a queue to store the pointers to the data packet. Suppose, at any time, we have `K` data packets inside the queue. Hence the space complexity will be O(K).

Space Complexity : O(K) where `K` is the number of data packets in the queue.

Advantages of Leaky Bucket Algorithm

- Converts bursty traffic into smooth traffic. The output is free of any jitters.
- Less congestion in the network.
- Easy to implement.
- Priority queues can be used to implement a priority-based packet transmission scheme.

Disadvantages of the Leaky Bucket Algorithm

- The output is uniform only if the bucket is non-empty. In case the bucket is empty or there is some packet loss, then the output can contain gaps.
- The packets are transferred only on ticks. If the packet size is large, then the queue gets filled up quickly as the effective transmission rate is low. This can lead to packet loss as the queue is of finite size.

Conclusion

- The leaky bucket algorithm is essentially a congestion control mechanism. It controls the shape of the traffic entering the network.
- It helps in reducing the congestion in the network as each host is transmitting packets at a constant rate.
- Traffic shaping is implemented at the hardware level and many algorithms are used to shape the traffic depending on the requirements.
- The Leaky Bucket algorithm is implemented in the host operating system. It makes bursty traffic, smooth by storing the data packets temporarily and sending them into the network at regular intervals, depending upon the Leak Rate.