TERMWORK

Title :- Leaky Bucket Algorithm

Objective :

* The primary objective is to successfully implement the Leaky Bucket algorithm in order to understand functioning and practical application in controlling data flow within a network.
* Investigating the influence of varying parameters such as bucket capacity and token addition rate, on the algorithm's behaviour.

Theory : The Leaky Bucket algorithm is widely used traffic shaping mechanism in computer networks designed to control the rate at which data is transmitted from a source to a destination. The concept is analogous to a bucket that can hold a limited amount of water, with a leak at the bottom. In the context of data transmission, the bucket represents a buffer with a fixed capacity and the leak symbolizes the allowed output rates. Tokens which symbolize units of data, are added to the bucket at a specific rate. If the bucket is full, excess tokens are discarded. When packets are to be transmitted, the algorithm checks if there are tokens available in the bucket. If tokens are present, packets are transmitted and the tokens are then leaked from the bucket. This process is fundamental for managing network traffic, providing a balance between resource utilization and preventing excessive data rates that could degrade network performance.

Algorithm:

1. Initialize leaky bucket with 'capacity', 'rate', 'tokens' and 'last_time'
2. In the add_tokens method, calculate elapsed time since the last update. Add tokens based on elapsed time and rate ensuring capacity is not exceeded update 'last_time'.
3. In the packet_process method, call add_tokens to update tokens based on time elapsed.
   Check if there are enough and print 'packet sent successfully' if no drop the packet and print 'packet dropped due to inssufficient tokens'.
4. Process each packet using 'process_packet' method.
5. Leaky bucket regulates packet transmission rate, avoiding network congestion.

Program:

```
class LeakyBucket:
    def __init__(self, capacity, rate):
        self.capacity = capacity
        self.rate = rate
        self.tokens = 0
        self.last_time = 0
    def add_tokens(self):
        current_time = time.time()
        time_elapsed = current_time - self.last_time
        self.tokens = min(self.capacity, self.tokens + time_elapse
                                        self.rate)
        Self.last_time = current_time
```

```
    def process_packet(self, packet_size):
        self.add_tokens()
        if packet_size <= self.tokens:
            self.tokens = packet_size
            print(f"Packet of size {packet_size} sent
                                successfully")
        else:
            print(f"Packet of size {packet_size} dropped due
                        to insufficient tokens")

import time
def main():
    capacity = 1000
    rate = 1
    packets = [200, 500, 600, 700, 450, 400, 200]
    bucket = LeakyBucket(capacity, rate)
    for packet_size in packets:
        bucket.process_packet(packet_size)

if __name__ == "__main__":
    main()
```

Outcomes :

* Learn to analyze the impact of varying parameters (e.g. bucket capacity, token addition rate) on algorithm behaviour.
* Recognizing the significance of the leaky bucket in mitigating network issues and ensuring the reliability of data transfer.

Conclusion :

The leaky bucket algorithm is an effective tool for controlling the flow of data in a network. By regulating the rate at which data is transmitted, it helps prevent congestion and ensures a smoother data transfer experience. Through, this experiment, a deeper understanding of traffic shaping and algorithm control has been achieved.

TERM WORK 6

Title : To show and demonstrate the use of Cyclic Redundancy Check.

Objective : To demonstrate the use of cyclic redundancy check and to understand the basic concepts of error detection.

Theory : Cyclic Redundancy Check (CRC) is an error detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data. Here's the theory behind it,

* Polynomial Codes – CRC is based on the algebraic structure of polynomial codes over finite fields, specifically the field of two elements.
* Division – The sender treats the data as a polynomial and divides it by a specific polynomial known as the generator polynomial
* Error Detection – The receiver performs the same division operation and if the remainder is non-zero, an error is detected

Algorithm :

* When the receiver gets the message, they perform the same calculations to get their own CRC checksum,
* If their checksum matches the one we sent, it means the message was transmitted correctly.
* If the checksums dont match, it mean there was an error in transmission.

Program :

```
def crc_remainder(input_bitstring, polynomial, initial_fill):
    dividend = input_bitstring + '0' * (len(polynomial) - 1)
    dividend = list(dividend)
    polynomial = list(polynomial)
    for i in range(len(input_bitstring)):
        if dividend[i] == '1':
            for j in range(len(polynomial)):
                dividend[i+j] = str(int(dividend[i+j]))
    remainder = ''.join(dividend)[-len(polynomial) + 1:]
    codeword = input_bitstring + remainder
    return remainder, codeword.
def crc_check(codeword, polynomial):
    remainder, _ = crc_remainder(codeword, polynomial, '0'*(len(polynom
    return remainder == '0' * (len(polynomial) - 1)
    remainder, codeword = crc_remainder(message, polynomial, '0'
    print("Original Message :", message)
    print("Polynomial :", polynomial)
    print("Transmitted Codeword :", codeword)
    print("Remainder :", remainder)

    choice = eval(input())
    if choice == 1:
        is_valid = crc_check(codeword, polynomial)
    else:
        error_position = eval(input("Enter the bit position:"))
        codeword = codeword[:error_position]
        print("Received Codeword with Error:", codeword)
        is_valid = crc_check(codeword, polynomial
```

```
if is_valid:
    print("No error detected!")
else:
    print("Error detected!")
```

Course Outcome :

* To understand error detection and error mitigation in computer networks
* To understand the basic concepts of CRC.

Conclusion :

In conclusion, the CRC program efficiently detects errors in data transmission. The code demonstrates key concepts such as polynomial codes, division and checksum calculation.

Reference :

Larry L Peterson and Bruce S Davie, Computer Networks, fifth edition, ELSEVIER