Lab No. 6: Implementation of Error Detection and Error Correction Techniques.

Objectives

- Design and Implementation of Error detection in transmitted data at the receiver side using Parity check and Cyclic Redundancy Check.
- Design and Implementation of error detection and error correction in transmitted data at the receiver side using Hamming code.

Error Control Mechanism

Error control mechanism plays an important role in the transmission of data from one source to another. The noise also gets added into the data when it transmits from one system to another, which causes errors in the received binary data at other systems. The bits of the data may change (either 0 to 1 or 1 to 0) during transmission.

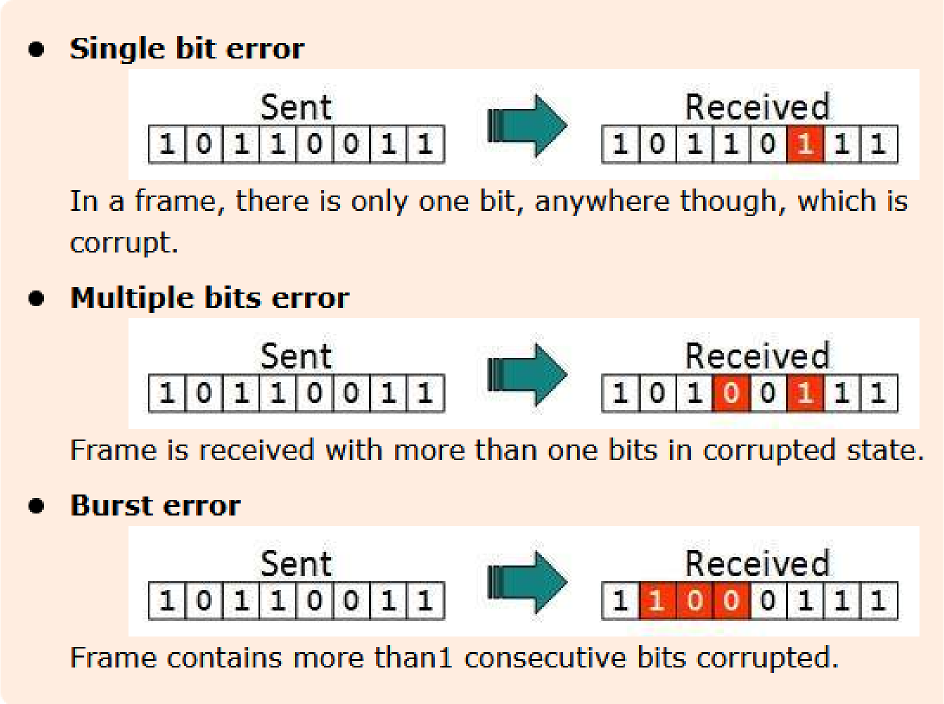
Error control mechanism may involve two possible ways:

- Error detection
- Error correction

It is impossible to avoid the interference of noise, but it is possible to get back the original data. For this purpose, we first need to detect either an error z is present or not using error detection codes. If the error is present in the code, then we will correct it with the help of error correction codes. The error detection codes are the code used for detecting the error in the received data bitstream. In these codes, some bits are included appended to the original bitstream.

Types of Errors

There may be three types of errors:



- **Single bit error**

In a frame, there is only one bit, anywhere though, which is corrupt.

- **Multiple bits error**

Frame is received with more than one bits in corrupted state.

- **Burst error**

Frame contains more than1 consecutive bits corrupted.

To detect errors, a common technique is to introduce redundancy bits that provide additional information. Various techniques for error detection include:: Parity check, Checksum, and CRC

Error-detecting codes encode the message before sending it over the noisy channels. The encoding scheme is performed in such a way that the decoder at the receiving can find the errors easily in the receiving data with a higher chance of success.

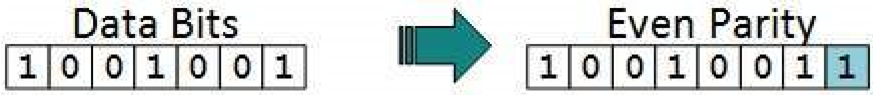
- These codes are used when we use message backward error correction techniques for reliable data transmission. A feedback message is sent by the receiver to inform the sender whether the message is received without any error or not at the receiver side. If the message contains errors, the sender retransmits the message.
- In error detection codes, in fixed-size blocks of bits, the message is contained. In this, the redundant bits are added for correcting and detecting errors.
- These codes involve checking of the error. No matter how many error bits are there and the type of error.
- Parity check, Checksum, and CRC are the error detection techniques.

Algorithm

Parity check:

One extra bit is sent along with the original bits to make number of 1s either even in case of even parity, or odd in case of odd parity.

The sender while creating a frame counts the number of 1s in it. For example, if even parity is used and number of 1s is even then one bit with value 0 is added. This way number of 1s remains even.If the number of 1s is odd, to make it even a bit with value 1 is added.



The receiver simply counts the number of 1s in a frame. If the count of 1s is even and even parity is used, the frame is considered to be not-corrupted and is accepted. If the count of 1s is odd and odd parity is used, the frame is still not corrupted.

If a single bit flips in transit, the receiver can detect it by counting the number of 1s. But when more than one bits are erro neous, then it is very hard for the receiver to detect the error.

At sender side:

1. Start the program
2. Total number of 1's in the data unit to be transmitted is counted.
3. The total number of 1's in the data unit is made even in case of even parity.
   The total number of 1's in the data unit is made odd in case of odd parity.
4. This is done by adding an extra bit called as parity bit.
5. The newly formed code word (Original data + parity bit) is transmitted to the receiver.
6. Stop the program

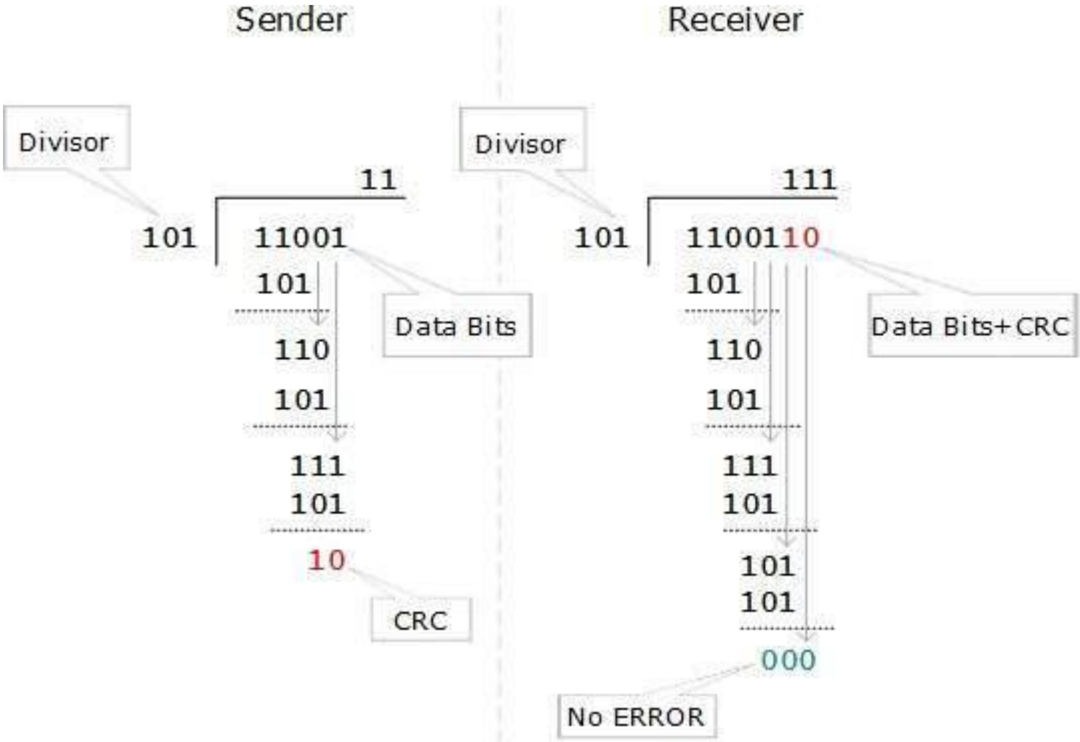
At receiver side :

1. Start the program
2. Receiver receives the transmitted code word.
3. The total number of 1's in the received code word is counted.

   Then, following cases are possible-
4. If total number of 1's is even and even parity is used, then receiver assumes that no error occurred.
5. If total number of 1's is even and odd parity is used, then receiver assumes that error occurred.
6. If total number of 1's is odd and odd parity is used, then receiver assumes that no error occurred.
7. If total number of 1's is odd and even parity is used, then receiver assumes that error occurred.
8. Stop

Cyclic Redundancy Check:

CRC is a different approach to detect if the received frame contains valid data. This technique involves binary division of the data bits being sent. The divisor is generated using polynomials. The sender performs a division operation on the bits being sent and calculates the remainder. Before sending the actual bits, the sender adds the remainder at the end of the actual bits. Actual data bits plus the remainder is called a codeword. The sender transmits data bits as codewords.



At the other end, the receiver performs division operation on codewords using the same CRC divisor. If the remainder contains all zeros the data bits are accepted, otherwise it is considered as there some data corruption occurred in transit.

At sender side:

1. Start the program
2. A string of n 0’s is appended to the data unit to be transmitted. Here, n is one less than the number of bits in CRC generator.
3. Binary division is performed of the resultant string with the CRC generator.
4. After division, the remainder so obtained is called as CRC. It may be noted that CRC also consists of n bits.
5. The CRC is obtained after the binary division. The string of n 0’s appended to the data unit earlier is replaced by the CRC remainder.
6. The newly formed code word (Original data + CRC) is transmitted to the receiver.
7. Stop

At receiver side:

1. Start the program
2. The transmitted code word is received.
3. The received code word is divided with the same CRC generator.
4. On division, the remainder so obtained is checked.
    a. If the remainder is zero, Receiver assumes that no error occurred in the data during the transmission. Receiver accepts the data.
    b. If the remainder is non-zero, Receiver assumes that some error occurred in the data during the transmission. Receiver rejects the data and asks the sender for retransmission.
5. Stop

Error Correction:

In the digital world, error correction can be done in two ways:

- **Backward Error Correction:** When the receiver detects an error in the data received, it requests back the sender to retransmit the data unit.

- **Forward Error Correction:** When the receiver detects some error in the data received, it executes error-correcting code, which helps it to auto-recover and to correct some kinds of errors.

The first one, Backward Error Correction, is simple and can only be efficiently used where retransmitting is not expensive. For example, fiber optics. But in case of wireless transmission retransmitting may cost too much. In the latter case, Forward Error Correction is used.

A single additional bit can detect the error, but cannot correct it. For correcting the errors, one has to know the exact position of the error. For example, If we want to calculate a single-bit error, the error correction code will determine which one of seven bits is in error. To achieve this, we have to add some additional redundant bits. Suppose r is the number of redundant bits and d is the total number of the data bits. The number of redundant bits r can be calculated by using the formula: $2^r >= d + r + 1$. The value of r is calculated by using the above formula. For example, if the value of d is 4, then the possible smallest value that satisfies the above relation would be 3. To determine the position of the bit which is in error, a technique developed by R.W Hamming is Hamming code which can be applied to any length of the data unit and uses the relationship between data units and redundant units.

There are several types of block codes, including:

- **Hamming Codes:** Hamming codes are a type of block code that can detect and correct single-bit errors. They are commonly used in digital systems to ensure the accuracy of transmitted data.

- **Reed-Solomon Codes:** Reed-Solomon codes are a type of block code that can correct multiple-bit errors. They are commonly used in storage systems, such as CD-ROMs and DVDs, to ensure the integrity of stored data.

- **BCH Codes:** BCH (Bose–Chaudhuri–Hocquenghem) codes are a type of block code that can correct a specific number of errors. They are commonly used in digital communication systems to ensure the accuracy of transmitted data.

Hamming Code

It is a technique developed by R.W. Hamming for error correction. The number of redundant bits can be calculated using the following formula: $2^r \geq m + r + 1$ where, r = redundant bit, m = data bit. Suppose the number of data bits is 7, then the number of redundant bits can be calculated using: = $24 \geq 7 + 4 + 1$ Thus, the number of redundant bits= 4 Parity bits.

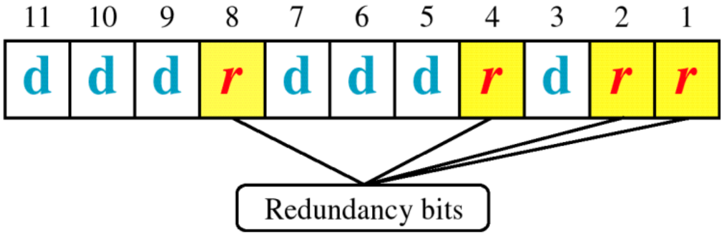
Hamming Code is simply the use of extra parity bits to allow the identification of an error.

At sender side:

1. Start the program

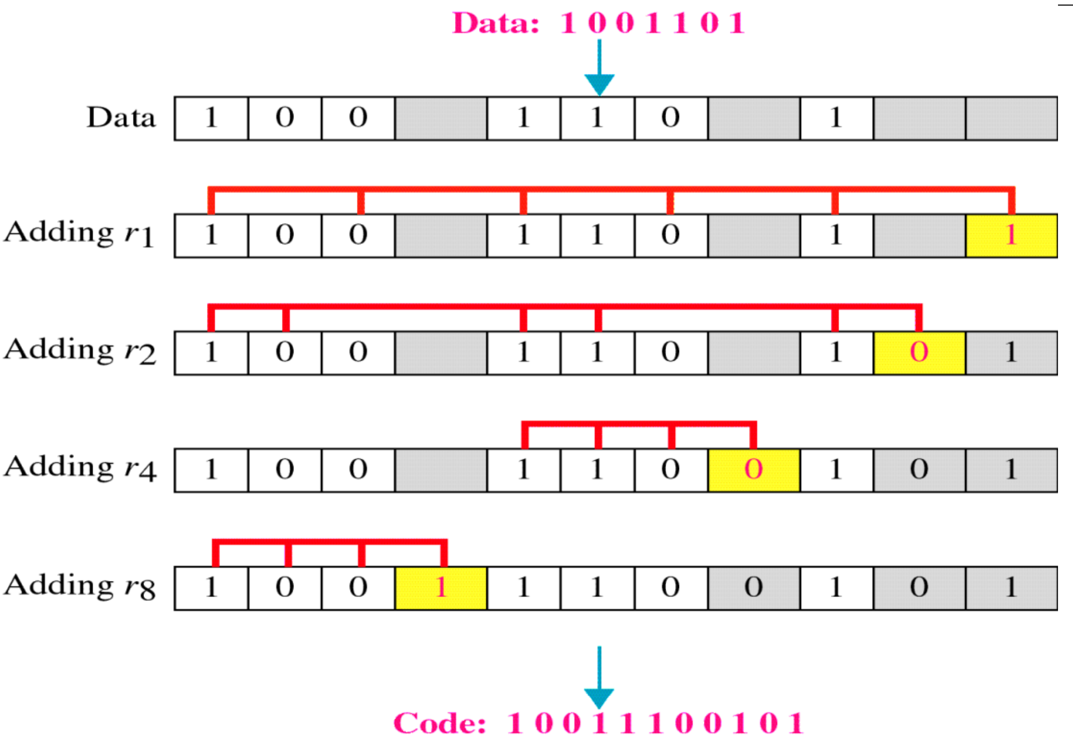
2. Write the bit positions starting from 1 in binary form (1, 10, 11, 100, etc).

3. All the bit positions that are a power of 2 are marked as parity bits (1, 2, 4, 8, etc). All the other bit positions are marked as data bits.



4. Each data bit is included in a unique set of parity bits, as determined its bit position in binary form.
    a. Parity bit 1 covers all the bits positions whose binary representation includes a 1 in the least significant position (1, 3, 5, 7, 9, 11, etc).
    b. Parity bit 2 covers all the bits positions whose binary representation includes a 1 in the second position from the least significant bit (2, 3, 6, 7, 10, 11, etc).
    c. Parity bit 4 covers all the bits positions whose binary representation includes a 1 in the third position from the least significant bit (4–7, 12–15, 20–23, etc).
    d. Parity bit 8 covers all the bits positions whose binary representation includes a 1 in the fourth position from the least significant bit bits (8–15, 24–31, 40–47, etc).
    e. In general, each parity bit covers all bits where the bitwise AND of the parity position and the bit position is non-zero.

5. Check for even parity set a parity bit to 1 if the total number of ones in the positions it checks is odd. Set a parity bit to 0 if the total number of ones in the positions it checks is even.

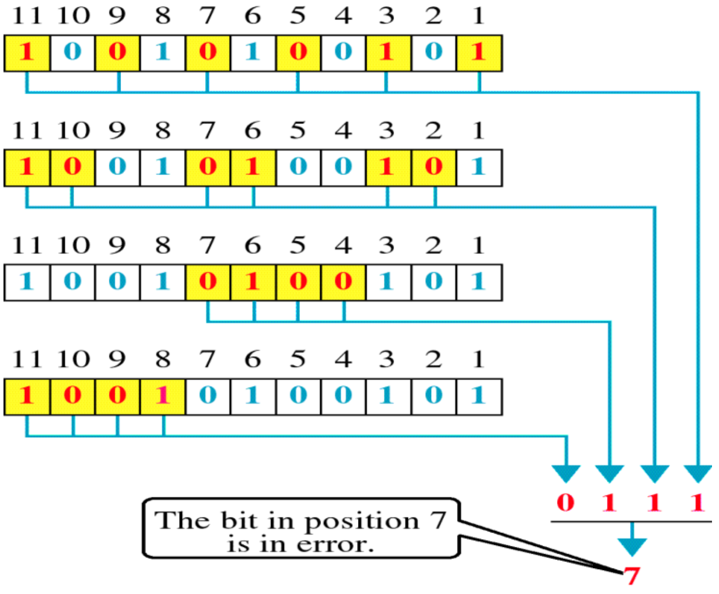


6. Stop

At receiver side:

1. Start the program

2. For all the parity bits we will check the number of 1's in their respective bit positions.
   a. For R1: bits 1, 3, 5, 7, 9, 11. We can see that the number of 1's in these bit positions are 3 and that's even so we get a 1 for this.
   b. For R2: bits 2,3,6,7,10,11 . We can see that the number of 1's in these bit positions are 3 and that's odd so we get a 1 for this.
   c. For R4: bits 4, 5, 6, 7 . We can see that the number of 1's in these bit positions are 1 and that's odd so we get a 1 for this.
   d. For R8: bit 8,9,10,11 . We can see that the number of 1's in these bit positions are 2 and that's even so we get a 0 for this.



3. The bits give the binary number 0111 whose decimal representation is 7. Thus, bit 7 contains an error. To correct the error the 7th bit is changed from 0 to 1.

4. Stop

Lab Exercises

1. Write a C program to transmit the data (binary values) from the sender side to receiver. The receiver side has to check whether the transmit data is corrupt or not using the parity technique – odd or even parity.

2. Write a C program to transmit the data (set of characters) from the sender side to receiver. The receiver side must check whether the transmit data is corrupted or not corrupted using the CRC polynomial techniques – CRC 12, CRC 16, CRC CCIP.

3. Implement the Hamming code method to detect the error in transmitted data and correct the error bit in the transmitted data.

Lab No. 7 : Application Development usingSocket Programming

Obj/ectives

- To apply the socket programming concepts in developing the real-world applications
- communication using Python)

In the network layer, before the network can make Quality of service guarantees, it must know what traffic is being guaranteed. One of the main causes of congestion is that traffic is often bursty. The congesting control algorithms are basically divided into two groups: open loop and closed loop. Open loop solutions attempt to solve the problem by good design, in essence, to make sure it does not occur in the first place. Once the system is up and running, midcourse corrections are not made. Open loop algorithms are further divided into ones that act at source versus ones that act at the destination.

In contrast, closed loop solutions are based on the concept of a feedback loop if there is any congestion. Closed loop algorithms are also divided into two subcategories: explicit feedback and implicit feedback. In explicit feedback algorithms, packets are sent back from the point of congestion to warn the source. In implicit algorithm, the source deduces the existence of congestion by making local observation, such as the time needed for acknowledgment to come back.

The presence of congestion means that the load is (temporarily) greater than the resources (in part of the system) can handle. For subnets that use virtual circuits internally, these methods can be used at the network layer. Another open loop method to help manage congestion is forcing the packet to be transmitted at a more predictable rate. This approach to congestion management is widely used in ATM networks and is called traffic shaping.

To understand this concept first we have to know little about traffic shaping. Traffic Shaping is a mechanism to control the amount and the rate of traffic sent to the network. Approach of congestion management is called Traffic shaping. Traffic shaping helps to regulate the rate of data transmission and reduces congestion.

There are 2 types of traffic shaping algorithms:

- Leaky Bucket
- Token Bucket

Leaky Bucket

Suppose we have a bucket in which we are pouring water, at random points in time, but we have to get water at a fixed rate, to achieve this we will make a hole at the bottom of the bucket. This will ensure that the water coming out is at some fixed rate, and also if the bucket gets full, then we will stop pouring water into it. The input rate can vary, but the output rate remains constant. Similarly, in networking, a technique called leaky bucket can smooth out bursty traffic. Bursty chunks are stored in the bucket and sent out at an average rate.

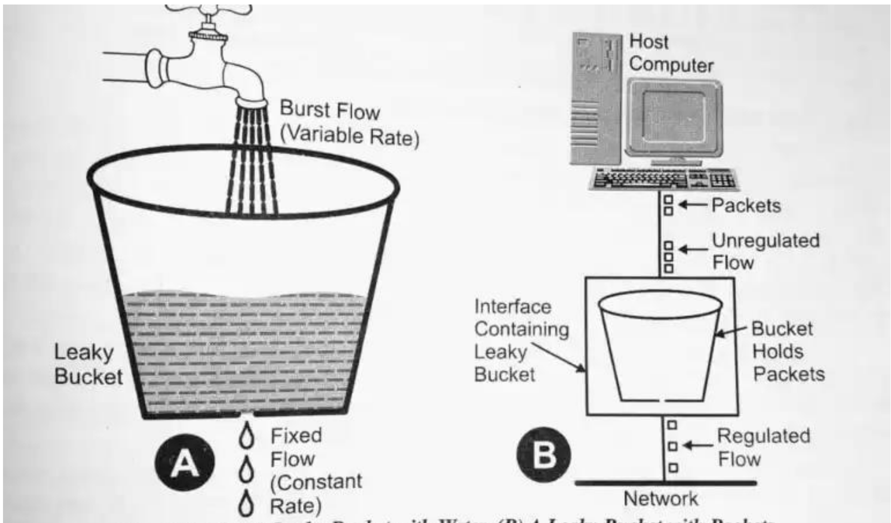
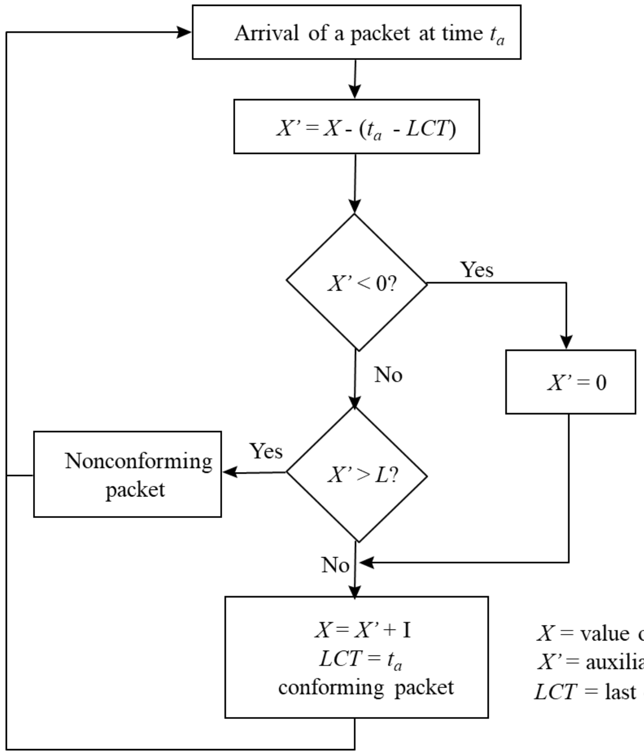


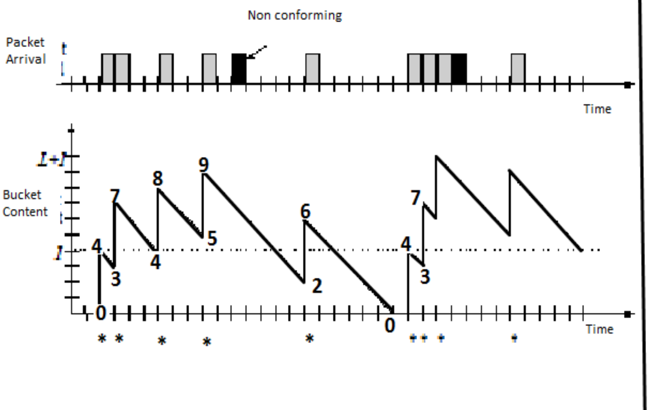
Fig. 5.07 : (A) A Leaky Bucket with Water. (B) A Leaky Bucket with Packets

Each host is connected to the network by an interface containing a leaky bucket, that is, a finite internal queue. If a packet arrives at the queue when it is full, the packet is discarded. In other words, if one or more process are already queued, the new packet is unceremoniously discarded. This arrangement can be built into the hardware interface or simulate d by the host operating system. In fact it is nothing other than a single server queuing system with constant service time.

The host is allowed to put one packet per clock tick onto the network. This mechanism turns an uneven flow of packet from the user process inside the host into an even flow of packet onto the network, smoothing out bursts and greatly reducing the chances of congestion.



X = value of the leaky bucket counter
X' = auxiliary variable
LCT = last conformance time



Behaviour of Leaky Bucket

TRACE:

X'=X-(ta-LCT) :ta:Arrival Time,LCT Last Confriming Time

Packet 1:

X'=0-[2-2]=0 X'>=0,X'<L {ta=2,LCT=2} L=6
X=X'+I=0+4=4
LCT=2

Packet 2:

X'=4-[3-2]=3
X=X'+I=3+4=7 X'>=0,X'<L
LCT=3

Packet 3:

X'=7-[6-3]=4 X'>=0,X'<L
X=4+4=8
LCT=6

Packet 4:

X'=8-[9-6]=5 X'>=0,X'<L
X=5+4=9
LCT=9

Packet 5:

X'=9-[11-9]=7 X'>=0,X'>L:NON CONFORMING
X=9[Unchanged]
LCT=9[Unchanged]

Packet 6:

X'=9-[16-9]=2
X'=2+4=6 X'>=0,X'<L
LCT=16

Packet 7:

X'=6-[23-16]=-1
X'=0 {X'<L}
X=0+4=4,LCT=23

Packet 8

X'=4-[24-23]=3
X=3+4=7
LCT=24

Algorithm:

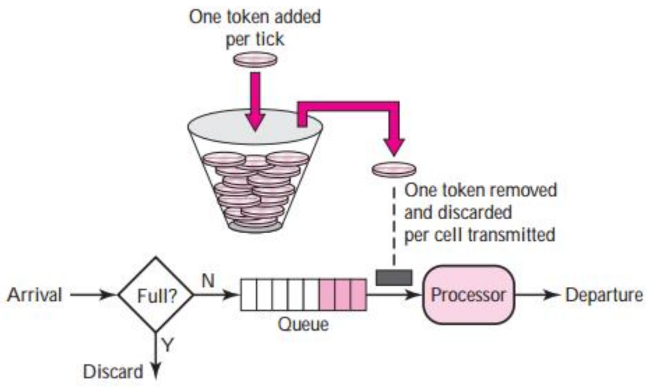
1. Start the program
2. Set the bucket size or the buffer size.
3. Set the output rate.
4. Transmit the packets such that there is no overflow.
5. Repeat the process of transmission until all packets are transmitted. (Reject packets where its size is greater than the bucket size)
6. Stop

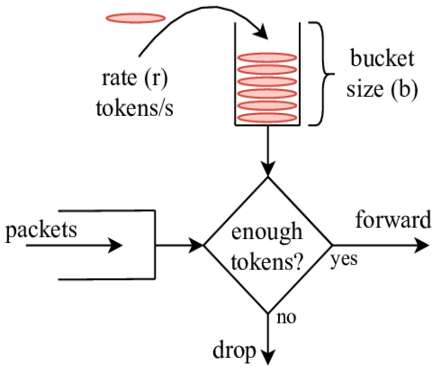
Token Bucket

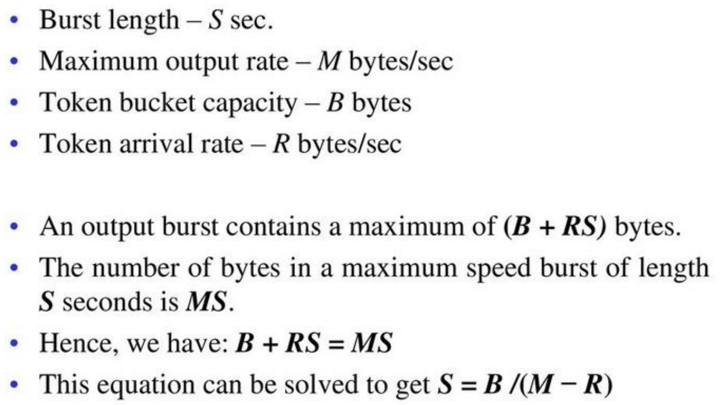
The leaky bucket algorithm enforces output patterns at the average rate, no matter how busy the traffic is. So, to deal with more traffic, we need a flexible algorithm so that the data is not lost. One such approach is the token bucket algorithm. When compared to the Leaky bucket the token bucket algorithm is less restrictive which means it allows more traffic. The limit of busyness is restricted by the number of tokens available in the bucket at a particular instant of time.

In some applications, when large bursts arrive, the output is allowed to speed up. This calls for a more flexible algorithm, preferably one that never loses information. Therefore, a token bucket algorithm finds its uses in network traffic shaping or rate-limiting. It is a control algorithm that indicates when traffic should be sent. This order comes based on the display of tokens in the bucket. The bucket contains tokens. Each of the tokens defines a packet of predetermined size. Tokens in the bucket are deleted for the ability to share a packet. When tokens are shown, a flow to transmit traffic appears in the display of tokens. No token means no flow sends its packets. Hence, a flow transfers traffic up to its peak burst rate in good tokens in the bucket.

The implementation of the token bucket algorithm is easy − a variable is used to count the tokens. For every t seconds the counter is incremented and then it is decremented whenever a packet is sent. When the counter reaches zero, no further packet is sent out.







- Burst length – S sec.
- Maximum output rate – M bytes/sec
- Token bucket capacity – B bytes
- Token arrival rate – R bytes/sec

- An output burst contains a maximum of $(B + RS)$ bytes.
- The number of bytes in a maximum speed burst of length S seconds is MS.
- Hence, we have: $B + RS = MS$
- This equation can be solved to get $S = B/(M - R)$

Algorithm:

1. Start the program
2. In regular intervals, tokens are thrown into the bucket f.
3. The bucket has a maximum capacity f.
4. If the packet is ready, then a token is removed from the bucket, and the packet is sent.
5. Suppose, if there is no token in the bucket, the packet cannot be sent.
6. Stop

Lab Exercises

1. Write a C program to avoid the congestion in network by using the leaky bucket method. Suppose the network packets arrive at a leaky bucket policer at time interval t = 1, 2, 3, 5, 6, 8, 11, 12, 15, 16, and 19 seconds with a packet size of 4 bytes each. Assume that the leaky bucket size X= 10 (queue size) bytes and the outgoing rate is one byte for every second. Identify the conforming and nonconforming packets in the queue and display the process on the screen.

2. Write a C program to avoid the congestion in network by using the Token bucket method. Assume that we have a token bucket shaper that has a replenishment rate r = 10 KBps, an infinite maximum rate R, a bucket size b = 50 Kbytes and that the bucket starts off full. Also assume that a sender emits 15 Kbytes packet size every 0.5 seconds in a periodic manner, starting at t = 0.5 seconds. For this question, you can assume that if sufficient tokens are available, packets pass through the token bucket instantaneously, otherwise they are queued until there are.
   i. How many tokens are left in the bucket after 1.5 seconds?
   ii. How long will it take until packets start to be queued or dropped?
   iii. Now, presume the sender can send as much as they want, whenever they want. If the token bucket is changed to enforce a maximum rate R of 20 KBps, what would the maximum possible burst size be?