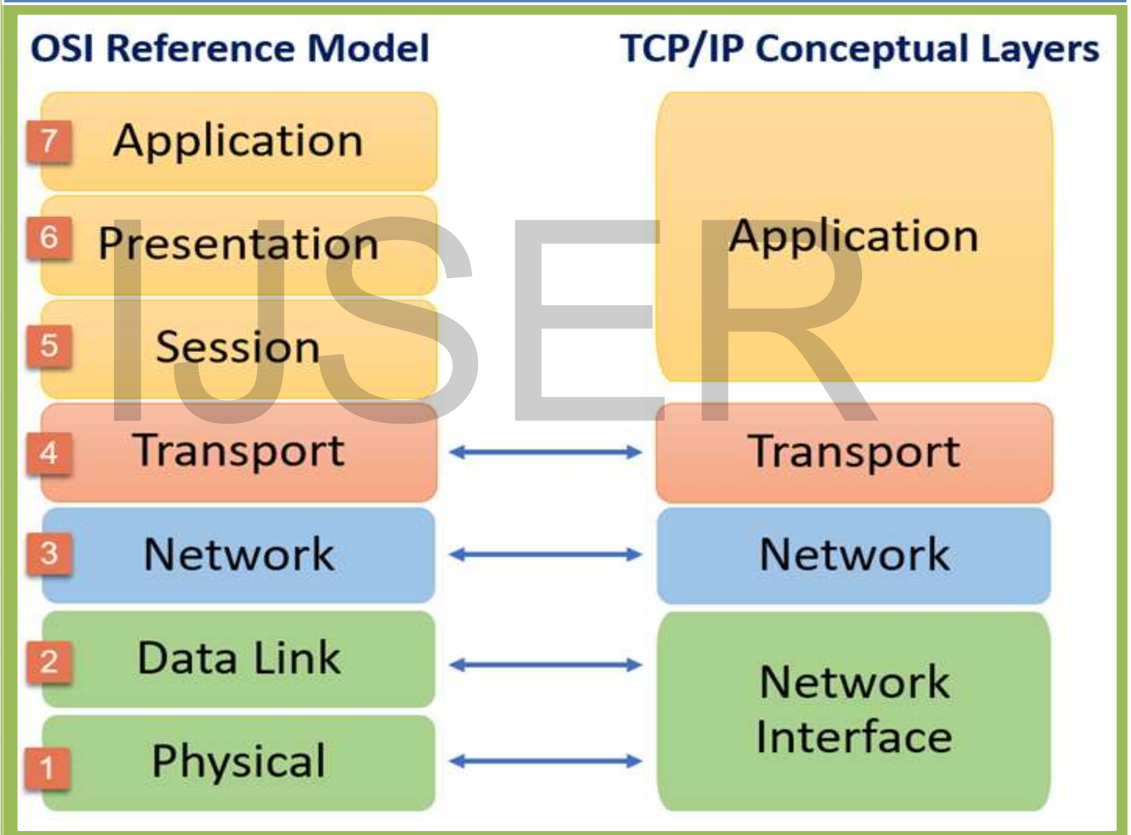
The Congestion Analysis of Transmission Control Protocol



Md. Shamim Hossain Biswas

Publication Partner: IJSER

12/1/2008

http://dx.doi.org/ 10.14299/ijser.01.12.2008

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

The Congestion Analysis of Transmission Control Protocol

A network congestion-avoidance technique

Md. Shamim Hossain Biswas

Publishing Partner:
IJSER.
www.ijser.org
ISSN: 2229-5518

http://dx.doi.org/ 10.14299/ijser.01.12.2008

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

Preface

Alhamdulillah, all praises to ALLAH (subhanahu wa ta′ala) who gives me the ability to complete this thesis work. I could not have finished my work if Almighty ALLAH did not make it possible.

I am thankful to my advisor Md. Shafiul Alam Khan who guided me throughout the work. He provided me with resources that I needed to carry out the work and gave me important guidelines whenever I was in a dilemma.

I would like to thanks our honorable professor Dr. Md. Mahfujur Rahman who suggested me to do thesis if I want to make a scientific carrier in Computer science. He also helped me during topic selection.

I also would like to vote of thanks our Chairman Dr. Md. Shamsul Alam who is the head of CS department due to encourage me to do simulation of desired outcome.

I would like to vote of thanks to our assistant professor Mr. Md. Kazi Jahidur Rahman who has arranged the presentation by his own effort. He has made the CS department light by leadership knowledge.

Finally I would like to thank my parents and friends who were supportive throughout the work. I also want to give special thanks to Scholar M. Mostafa kamal who was the renowned person made me proficient in English Language during this thesis activities and last but not least I would like to give special thanks Dr. Suria Pervin (Assistant professor, Department of computer science, Dhaka University) who gave me good guideline in completion this thesis.

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

Copyright and Trademarks

I hereby declare that this research monograph becomes the property of Md. Shamim Hossain Biswas and to be placed at the worldwide database access library for future computer security researchers and also be available Online.

Md. Shamim Hossain Biswas is the owner of this Monograph and own all copyrights of the Work. IJSER acts as publishing partner and author will remain owner of the content.

Md. Shamim Hossain Biswas

BSc in Computer Science & Engineering (Stamford University)

ACCA-Foundation (London School of Business and Finance, UK)

ORCID: 0000-0002-4595-1470

shamim44-165@diu.edu.bd

Cell: +8801531-262445

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

TABLE OF CONTENTS

	Page
PREFACE	iii
COPYRIGHT AND TRADEMARKS	iv
TABLE OF CONTENTS	v
LIST OF FIGURES	vii
LIST OF TABLES	viii
ABSTRRACT	ix
CHAPTER 1 INTRODUCTION	
1.1 Motivation and Background	1
1.2 Problem Specification	3
1.3 Method Specification	3
1.4 Thesis Organization	4
CHAPTER 2 BACGROUND	
2.1. Description of TCP	5
2.2 TCP Header	6
2.3 TCP Connection Establishment	7
2.4 TCP Connection Release	8
2.5 TCP Connection Management Modeling	8
2.6 TCP Transmission Policy	10
2.6.1: The Window Principle	10
2.6.2 Acknowledgments and Retransmissions	13
2.7 TCP Timer Management	13
2.8 Wireless TCP and UDP	16
2.9 TCP in Wireless or Cellular Mobile System	18
2.10 Link Characteristics that Impact TCP Performance	18
2.11 High Propagation Delay	18
2.12 Error Control	19
2.13 Retransmission	20
2.14 Forward Error Correction	20
2.15 Block Codes	20
2.16 Fountain Codes	24
2.17 Convolution Codes	24
2.18 Turbo Codes	24
CHAPTER 3 CONGESTION CONTROL METHODOLOGY	

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

3.1 Introductory Description 25

3.2 Description of Congestion Control Parameter 28

3.3 Congestion control Technique 29

3.4 Congestion Control Algorithms 38

CHAPTER 4 PROBLEM FORMULATION & RESULT DISCUSSION

4.1 Congestion control in TCP 43

4.2. Optimum Forward error correction 43

4. 3 TCP Routing Agent 44

4.4 TCP Congestion Control simulation details 45

4.4.1 Flowchart of TCP Congestion Control 54

4.4.2 Sender ACK Processing Flowchart 56

4.4.3 Congestion Control simulation Diagram 57

4.4.4 Summary of Forward Error Correction simulation 57

4.4.5 Forward Error correction 61

4.4.6 Parameter of FEC 64

4.5 Simulation Result of TCP Congestion Control 65

4.6 Simulation Result of Forward Error Correction 81

CHAPTER 5 CONCLUSION AND FUTURE WORK

5.1 Conclusion 88

5.2 Author Contribution 88

5.3 Recommendation for Further Study 89

REFERENCES 90

APPENDIX A 92

APPENDIX B 94

APPENDIX C 104

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

LIST OF FIGURES

NO		Page
Figure 1.1	TCP/IP protocol stack	1
Figure 1.2	Functions of layers of TCP/IP MODEL	2
Figure 2.1	TCP Header	6
Figure 2.2	(a)TCP connection establishment in the normal case. (b) Call collision.	7
Figure 2.3	TCP connection management finite state machines	10
Figure 2.4	Simple Transmission-Flows	10
Figure 2.5	Window Principles, Message Packet	11
Figure 2.6	Window Principles	11
Figure 2.7	Message Packets	11
Figure 2.8	Window Principles Applied to TCP	12
Figure 2.9	Acknowledgment and Retransmission Process	13
Figure 2.10	Timer management	14
Figure 2.11	Splitting a TCP connection into two connections	17
Figure 2.12	Block coding technique	21
Figure 2.13	Redundancy & Check bit calculation System	21
Figure 2.14	Constructing H (11, 7) from 7 bit data frame.	22
Figure 2.15	Redundancy bits calculation	23
Figure 2.16	Adding check bits	23
Figure 2.17	Error correction mechanism	24
Figure 3.1	(a) A fast network feeding a low-capacity receiver (b) A slow network feeding a high-capacity receiver	26
Figure 3.2	An Example of the Internet Congestion algorithm	28
Figure 4.1	Procedure of Receiver system	54
Figure 4.2	Sender Acknowledgement Segment	55
Figure 4.3	Sender segment processing System	56
Figure 4.4	Congestion Control simulation	57
Figure 4.5	Flowchart at the sender end	61
Figure 4.6	Flowchart of processing at the receiver end	62
Figure 4.7	Checksum Generator	63
Figure 4.8	Data unit and Checksum	63
Figure 4.9	Simulation Model of FEC	64

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

Figure 5.1 Logical Congestion window view 87

LIST OF TABLES

NO		Page
Table 2.1	Calculating the check bit for Hamming code.	22
Table 4. 1	The following mentioned result is for sender performance of 5 second simulation on run	65
Table 4. 2	The following mentioned result is for sender performance of 10 second simulation run	66
Table 4. 3	The following mentioned result is for sender performance of 30 second simulation run.	68
Table 4.4	The following mentioned result is for sender performance of 50 second simulation run.	72
Table 4.5	The following result is for Forward error correction of receiver side performance	82

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

ABSTRACT

This thesis deals with the investigation of transmission control protocol (TCP) performance in non-wire line (Cellular Mobile System) environments. TCP Congestion Control analysis has been analyzed in Java Programming Language. The congestion control has been simulated in Java programming language environment and it also investigates of the effects of combining forward error correction (FEC) with TCP in simulation environment of Java. FEC reduces the number of retransmissions and shows better congestion control behavior, which can effectively run the network at a higher load. In TCP/IP Protocol Architecture, TCP provides reliable end-to-end communication. Traditional TCP implementations are tuned to work well over wired networks. A packet loss is occurred in a wired network mainly due to network congestion. On the other hand in a wireless link packet losses are caused mainly due to bit errors resulted from noise, interference, and various kind of fading. TCP has no idea whether a packet loss is caused by congestion or bit error. TCP assumes loss is caused by congestion and turns on its congestion control algorithms to slow down the amount of data it transmits. So invoking congestion control algorithm for bit errors of wireless channel reduces TCP throughput drastically. Proposal of this thesis is to analysis congestion/packet loss. FEC means adding some redundancy information along with data that means this redundancy will consume some portion of the available bandwidth. But using FEC reduces the number of retransmission at TCP level and thus preserves time to send more data. As a result overall performance may be increased. By simulation in Java Programming Language and in Windows Environment the comparison between various TCP versions is done and it is demonstrated that in certain cases performance of accurate data transmission can be increased significantly.

Keywords: TCP/IP, Cellular Mobile System, forward error correction, throughput, congestion/packet loss. Wireless-link-packet, Java Programming, Eclipse Framework, Simulation, Add-hoc network.

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

CHAPTER 1

INTRODUCTION

1.1. Motivation and Background

Data communications has dramatically increased in popularity over the last decade. Millions of users exchange a wide variety of information, mainly by using the World Wide Web (WWW). Companies, Universities, Schools and millions of homes are connected and access the web daily for business and leisure activities. The basic access technology in computer networking has been wire based. However the increasing demand for connectivity from anywhere at any time has led to the development of wireless networking technologies. These new access media pose several problems to the traditionally used communication protocols. This thesis is concerned with the impact that wireless technologies have on one of the major protocols used in today's Internet: the Transmission Control Protocol (TCP). Before presenting the details of this protocol and the challenges it faces in non-wire line environments, general overview of the Internet is given. With non-wire line environments means all the access techniques that use wireless transmissions, like radio networks, wireless Local Area Networks (LAN) [2] or Cellular Mobile System. A picture of TCP/IP protocol stack [1] is shown in Figure 1. Each layer performs a specific task. The discussion of this stack is given from the bottom up.

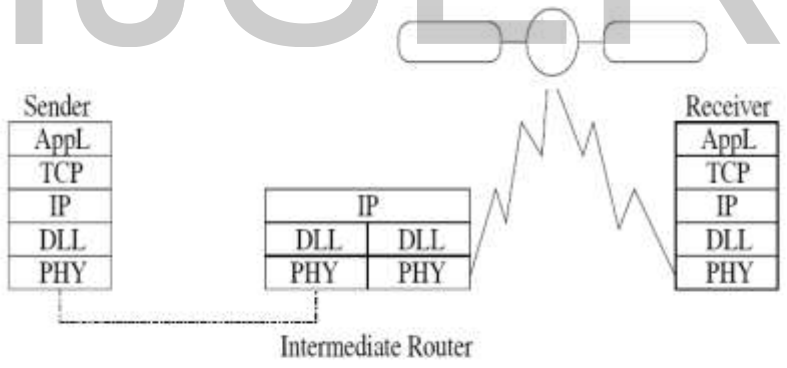


Figure 1.1: TCP/IP protocol stack

The lowest layer is the physical layer (PHY), which represents the physical medium used for communication. A wide range of transmission media can be used, for example coaxial cable, fiber-optic, or twisted pair. Next is the Data Link Layer (DLL) which is split into two subs-layers: Medium Access Control (MAC) layer, that controls the access to the physical medium, and Logical Link Control (LLC) layer, that can provide reliable, connection oriented service between two neighboring network elements. The center piece of the Internet protocol stack is the Internet Protocol (IP). It is located on the third (network) layer in the stack. Every computer connected to the Internet needs to

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

run this protocol. There is no alternative. IP is responsible for finding a path through the network from the sending to the receiving end-host, a task called routing. Intermediate nodes, called routers, forward the packets from one hop to the next until the destination is reached. The IP protocol is considered to be unreliable, which means it is allowed to lose or re-order data packets during transmission.

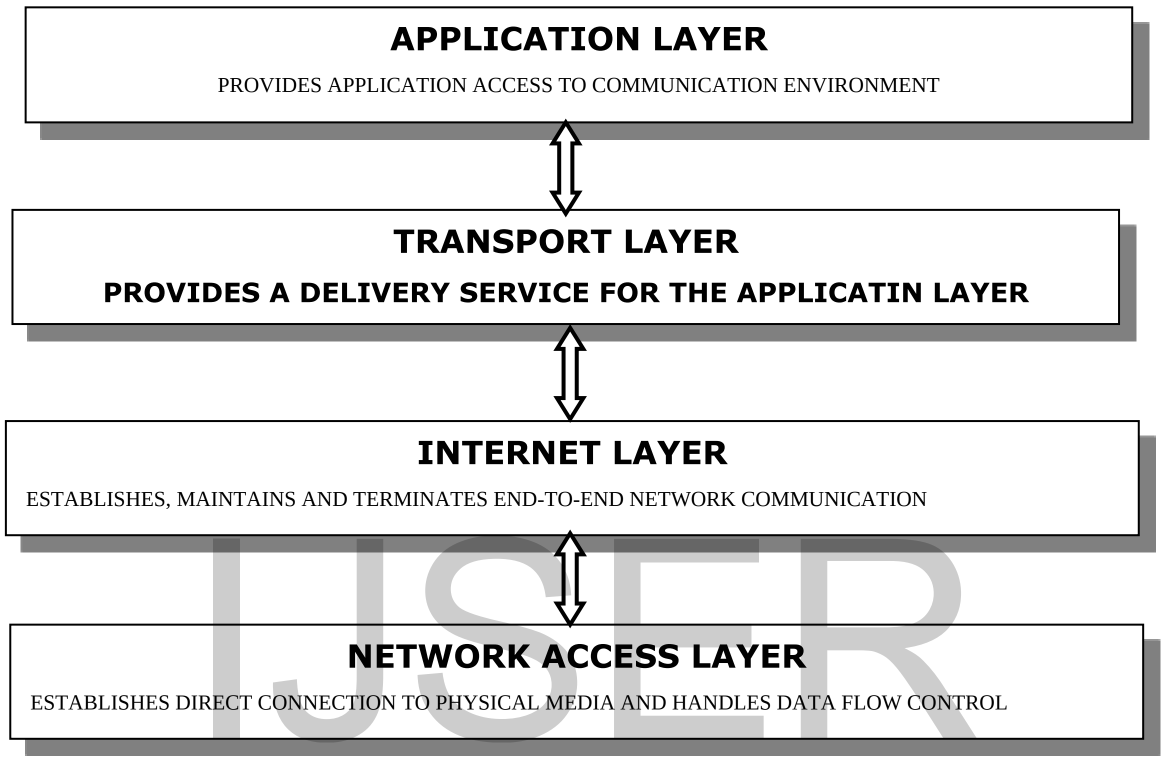

APPLICATION LAYER
PROVIDES APPLICATION ACCESS TO COMMUNICATION ENVIRONMENT
TRANSPORT LAYER
PROVIDES A DELIVERY SERVICE FOR THE APPLICATIN LAYER
INTERNET LAYER
ESTABLISHES, MAINTAINS AND TERMINATES END-TO-END NETWORK COMMUNICATION
NETWORK ACCESS LAYER
ESTABLISHES DIRECT CONNECTION TO PHYSICAL MEDIA AND HANDLES DATA FLOW CONTROL

Figure 1.2: Functions of layers of TCP/IP MODEL

Above the network layer is the transport layer. This layer generally only exists in end hosts of the Internet, not in the routers. There are two protocols used at the transport layer: User Datagram Protocol (UDP), which is unreliable and only provides addressing to the specific application in the end-computer, and Transmission Control Protocol (TCP), which is a reliable transport protocol. The highest layer in the Internet protocol stack is the Application layer. In this layer a unlimited variety of application protocols can exist. The most common ones are the World Wide Web (WWW), e-mail and File Transfer (FTP). Traditionally the physical media used to interconnect computers have been wire-based. The common characteristics of these media are the very low probability of data loss due to bit errors, the fast transmission of one packet between two end hosts, usually in the millisecond range, and the same bandwidth availability for the forward (sender to receiver) and the return (receiver to sender) path. In recent years the use of non-wire line physical media has become more and more common in computer networks. Wireless networks can support user mobility and can be deployed with much less infrastructure then their wired counterparts. Cellular Mobile System can provide

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

access to users in geographically remote areas, serve as an emergency backup if the wired infrastructure is destroyed, or connect distant network islands. These new media have quite different characteristics to the wired network. The delay between packet transmission and reception can be much higher, because of the limited bandwidth of the wireless media or the long propagation delay a Cellular Mobile System hop. Wireless links are often noisy, which means that packet loss due to bit errors is quite likely. There is also the possibility of different bandwidth on the forward and return channel.

1.2 Problem Specification

A packet loss is occurred in a wired network mainly due to network congestion. On the other hand in a wireless link packet losses are caused mainly due to bit errors resulted from noise, interference, and various kind of fading. When TCP detects a packet loss, it assumes this loss is caused by congestion and turns on its congestion control algorithms [3] and eventually slows down the amount of data it transmits to adjust with the low capacity of the network. When a packet is lost, TCP has no idea whether this loss is caused by congestion or bit error. As a result when packets are lost due to bit errors of wireless channel TCP wrongly interprets these losses as due to congestion and invokes congestion control algorithms, and reduces data transfer rate. This wrong act of TCP makes matters worse.

1.3 Method Specification

Now the question is how to get rid of these losses. These problems can be reduced by congestion control techniques what are TCP Tahoe, TCP Reno, TCP New Reno, TCP SACK, TCP Vegas and TCP FACK. Two intuitive strategies to hide the packet losses from TCP are Retransmission and Forward Error Correction (FEC). In Retransmission technique for congestion control, when a sender detects a packet loss, it just transmits it again. In FEC technique the sender add enough redundancy information along with data so that the receiver is able to figure out what the original data was. Every wireless technology has a mean to recover the losses resulted from the uncertainty of wireless channel. Common practice is to use FEC [4], retransmission or a combination of both at the link level. There are reasons why error recovery techniques are usually performed at link level. Because retransmission is faster at link level, and advantage of applying FEC at the link level is that more information about the channel can be utilized. One important thing is that rely only on retransmission is not possible, because which itself may be in error. But in practice many existing wireless standards use very limited or no FEC. So there is every possibility that some of the bit errors at wireless channel appear at TCP level [5].

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

1.4 Thesis Organization

An outline of the thesis is given in this section. Chapter 1 gives a motivation and background, Problem Specification, Method Specification and out line. Chapter 2 gives a description of TCP, such as, Introduction TCP Protocol, TCP Header, TCP Connection Establishment, TCP Connection Release, TCP Connection Management Modeling, TCP Transmission Policy, The Window Principle, Acknowledgments and Retransmissions, TCP Congestion Control, Introductory Description, Description of Congestion Control Parameter, Different Congestion control techniques, Tahoe TCP, RFC 896 – Nagle's Algorithm, Karn's Algorithm, Jacobson's Congestion Control Algorithms, Reno TCP – Fast Recovery, RFC 2018- Selective Acknowledgements, a Reno TCP extension, RFC 2582 – New Reno TCP, Vegas TCP – A Proactive Approach, Congestion Control Algorithms, Slow Start and Congestion Avoidance, Fast Retransmit/Fast Recovery, Re-starting Idle Connections, Loss Recovery Mechanisms, TCP Timer Management, Wireless TCP and UDP, TCP in Wireless or Cellular Mobile System, High Bit Error Rate, High Propagation Delay, Error Control, Retransmission, Forward Error Correction, Block Codes, Hamming Code , Fountain Codes, Convolution Codes, Turbo Codes, Forward Error Correction using Java, Checksum, Designing the Sender, Designing the Receiver, Simulation Model of FEC, Parameters of FEC. Chapter 3 gives the congestion control methodology. Chapter 4 introduces the implementation of Congestion Analysis using Java language and problem formulation of Congestion Control and Forward Error Correction on cont and also represent simulation results. It Chapter 5 represents the discussion and further study of this thesis. The last of the chapter, two simulation of Congestion Control one TCP Congestion Control using Tahoe and Reno in Appendix B and another one is Forward Error Correction in Appendix C both have been simulated using Java programming Language. In appendix A, a simple sender receiver program has been done.

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

CHAPTER 2

BACKGROUND

2.1. Description of TCP

The Transmission Control Protocol (TCP) is the most commonly used transport layer protocol in the Internet. It provides reliable, end-to-end, non- real-time data transfer. Since its original specification in 1981 TCP has undergone several changes and enhancements. This chapter will give an overview of the development of TCP, describe the most commonly used versions in today's Internet and introduce some experimental versions investigated in this thesis. In this chapter, we will give a general overview of the TCP protocol [6]. In the next one, we will go over the protocol header, field by field. A key feature of TCP, and one which dominates the protocol design, is that every byte on a TCP connection has its own 32-bit sequence number. When the Internet began, the lines between routers were mostly 56-kbps leased lines, so a host blasting away at full speed took over 1 week to cycle through the sequence numbers. At modern network speeds, the sequence numbers can be consumed at an alarming rate, as we will see later. Separate 32-bit sequence numbers are used for acknowledgements and for the window mechanism: The sending and receiving TCP entities exchange data in the form of segments. A TCP segment consists of a fixed 20-byte header (plus an optional part) followed by zero or more data bytes. The TCP software decides how big segments should be. It can accumulate data from several writes into one segment or can split data from one write over multiple segments. Two limits restrict the segment size. First, each segment, including the TCP header, must fit in the 65,515-byte IP payload. Second, each network has a maximum transfer unit, or MTU, and each segment must fit in the MTU. In practice, the MTU is generally 1500 bytes (the Ethernet payload size) and thus defines the upper bound on segment size.The basic protocol used by TCP entities is the sliding window protocol. When a sender transmits a segment, it also starts a timer. When the segment arrives at the destination, the receiving TCP entity sends back a segment (with data if any exist, otherwise without data) bearing an acknowledgement number equal to the next sequence number it expects to receive. If the sender's timer goes off before the acknowledgement is received, the sender transmits the segment again. Although this protocol sounds simple, there are a

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

number of sometimes subtle ins and outs, which we will cover below. Segments can arrive out of order, so bytes 3072-4095 can arrive but cannot be acknowledged because bytes 2048-3071 have not turned up yet. Segments can also be delayed so long in transit that the sender times out and retransmits them. The retransmissions may include different byte ranges than the original transmission, requiring a careful administration to keep track of which bytes have been correctly received so far. However, since each byte in the stream has its own unique offset, it can be done. TCP must be prepared to deal with these problems and solve them in an efficient way. A considerable amount of effort has gone into optimizing the performance of TCP streams, even in the face of network problems. A number of the algorithms used by many TCP implements will be discussed below.

2.2. TCP Header

Every TCP segment begins with a fixed-format, 20-byte header [6]. The header fields shown in figure 2.1 are as follows:

The Source Port and Destination Port fields identify the source and destination ports, respectively. These two fields plus the source and destination IP addresses, combine to uniquely identify each TCP connection. The sequence number identifies the byte in the stream of data from the sending TCP to the receiving TCP that the first byte of data in this segment represents. The Acknowledgement number field contains the next sequence number that the sender of the acknowledgement expects to receive. This is therefore the sequence number plus 1 of the last successfully received byte of data.

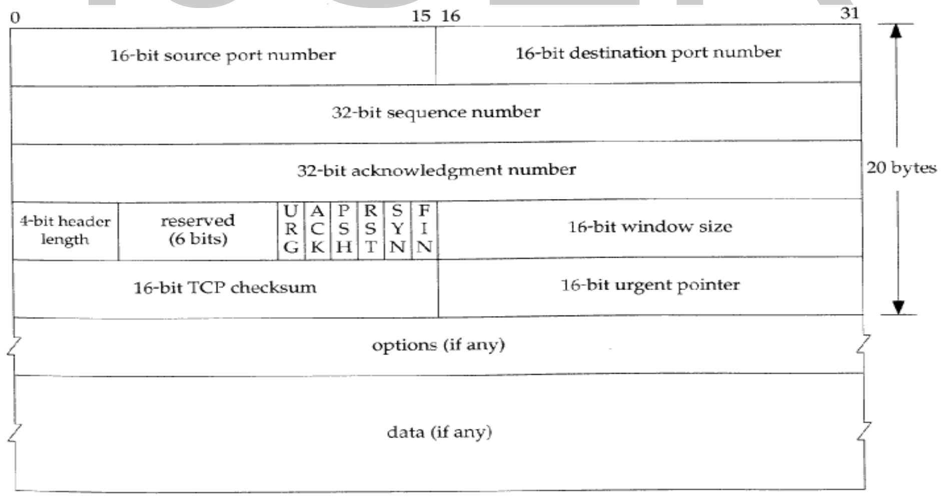


Figure 2.1: TCP Header

The header length gives the length of the header in 32-bit words. This is required because the length of the options field is variable. The 6-bit Flags field is used to relay control information between TCP peers. The possible flags include SYN, FIN, RESET, PUSH, URG, and ACK [9]. The SYN and Fin flags are used when establishing and terminating a TCP connection, respectively. The ACK flag is set any

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

time the Acknowledgement field is valid, implying that the receiver should pay attention to it. The URG flag signifies that this segment contains urgent data. When this flag is set, the Urgent Pointer field indicates where the non-urgent data contained in this segment begins. The PUSH flag signifies that the sender invoked the push operation; which indicates to the receiving side of TCP that it should notify the receiving process of this fact. Finally, the RESET flag signifies that the receiver has become confused and so wants to abort the connection. The Checksum covers the TCP segment, the TCP header and the TCP data. This is a mandatory field that must be calculated by the sender, and then verified by the receiver. The Option field is the maximum segment size option, called the MSS. Each end of the connection normally specifies this option on the first segment exchanged. It specifies the maximum sized segment the sender wants to receive. The data portion of the TCP segment is optional.

2.3. TCP Connection Establishment

Connections are established in TCP by means of the three-way handshake discussed. To establish a connection, one side, say, the Server passively waits for an incoming connection by executing the LISTEN and ACCEPTS primitives, either specifying a specific source or nobody in particular. The other side, say, the client executes a CONNECT primitive, specifying the IP address and port to which it wants to connect, the maximum TCP segment size it is willing to accept, and optionally some user data. The CONNECT primitive sends a TCP segment with the SYN bit on and ACK bit on off and waits for a response. When this segment arrives at the destination, the TCP entity there checks to see if there is a process that has done a LISTEN on the port given in the Destination port field. If not, it sends a reply with the RST bit on to reject the connection. If some process is listening to the port, that process is given the incoming TCP segment. IT can then either accept or reject the connection. If it accepts, an acknowledgement segment is sent back. The sequence of TCP segments sent in the normal case is shown in Fig :(a). Note that a SYN segment consumes 1 byte of sequence space so that it can be acknowledged unambiguously [6].

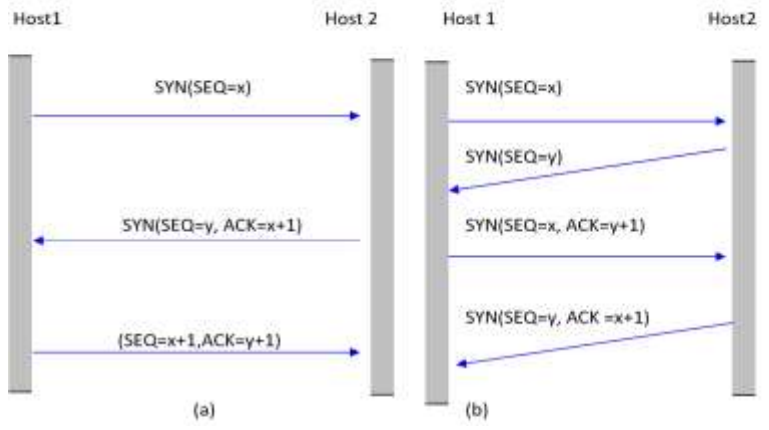


Figure: 2.2(a) TCP connection establishment in the normal case. (b) Call collision.

In the event that two hosts simultaneously attempt to establish a connection between the same two sockets, the sequence of events is as illustrated in Fig.2.2 (b).The result of these events is that just one connection is established, one two because connections are identified by there end points. If the first setup results in a connection identified by (x, y) And the second one does too, only one table entry is made,. Namely, for(x, y) The initial sequence number on a connection is not) for the reasons we discussed earlier. A clock-based scheme is used; with a clock tick every 4 sec. For additional safety, when a host crashes, it may not reboot for the maximum packet lifetime to make sure that no packets from previous connection are still roaming around the internet somewhere.

2.4 TCP Connection Release

Although TCP connection are full duplex, to understand how connections are released it is best to think of them as a pair of simplex connections. Each simplex connection is released independently of its sibling [6]. To release a connection, either party can use send a TCP segment with the FIN bit set, which means that it has no more data to transmit. When the FIN is acknowledged, the direction is shut down for new data. Data may contain to flow indefinitely in the other direction, however. When both directions have been shut down, the connection is released. Normally, four TCP segments are needed to release a connection, one FIN and one ACK for each direction. However, it is possible for the first ACK and the second FIN to be contained in the same segment, reducing the total count to three. Just as with telephone calls in which both people say goodbye and hang up the phone simultaneously, both ends of a TCP connection may send FIN segments at the same time. These are each acknowledged in the usual way, and the connection is shut down. There is, in fact, no essential difference between the two hosts releasing sequentially or simultaneously. To avoid the two-army problem, timers are used. If a response to a FIN is not forthcoming within two maximum packet lifetimes, the sender of the FIN releases the connection. The other side will eventually notice that nobody seems to be listening to it any more and will time out as well. While this solution is not perfect, given the fact that a perfect solution is theoretically impossible, it will have to do. In practice, problems rarely arise.

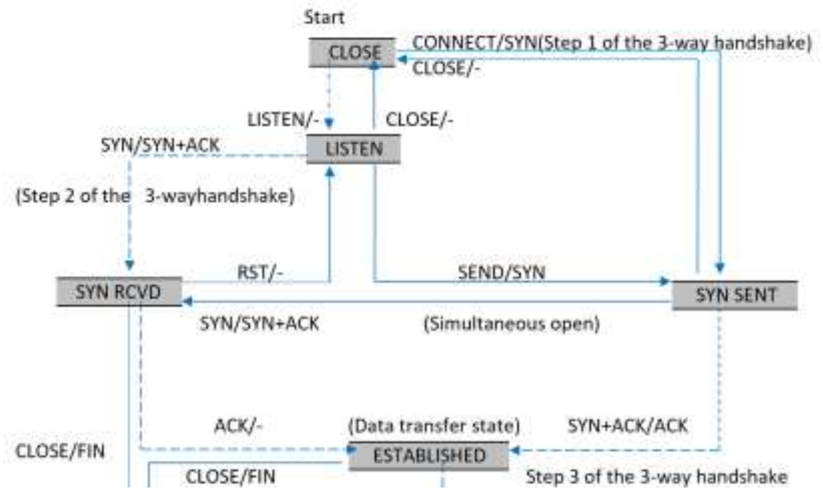
2.5 TCP Connection Management Modeling

The steps required establishing and release connections can be represented in a finite state machine with the 11 states listed in Fig.2.3 [6] in each state, certain events are legal. When a legal event happens, some action may be taken. If some other event happens, an error is reported. Each connections starts in the CLOSED state. It leaves that state when it does either a passive open (LISTEN), or an active open (CONNECT). If the other side does the opposite one, a connection is established and state becomes ESTABLISHED. Connection release can be initiated by either side. When it is complete, the state returns to CLOSED. The finite state machine itself is shown in Fig.2.3.

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

The common case of a client actively connecting to a passive server is shown with heavy lines-solid for the client, dotted for the server. The lightface lines are unusual event sequences. Each line in Fig. 2.3 is marked by an *event/action* pair. The event can either be a user-initiated system call (CONNECT, LISTEN, SEND or CLOSE), a segment arrival (SYN, FIN, ACK or RST), or in one case, a timeout of twice the maximum packet lifetime. The action is the sending of a control segment (SYN, FIN or RST) or nothing indicated by – Comments are shown in parenthesis. One can best understand the diagram by first following the path of a client (the heavy solid line), then later following the path of a server (the heavy dashed line). When an application program on the client machine issues a CONNECT request, the local TCP entry creates a connection record, marks it as beginning in the SYM SENT state and sends a SYN segment. Note that many connections may be open (or being opened) at the same time. on behalf of multiple applications, so the state is per connection and recorded in the connection record. When the SYN+ACK arrive, TCP sends the final ACK of the three-way handshake and switches into the ESTABLISHED state. Data can now be sent and received. When an application is finished, it executes a CLOSE primitive, which causes the local TCP entity to send a FIN segment and wait for the corresponding ACK (dashed box marked active close). When the ACK arrives, a transition is made to state FIN WAIT 2 and one direction of the Connection is now closed. When the other side close. Too, a FIN comes in which is acknowledged. Now both sides are closed, but TCP waits a time equal to the maximum packet lifetime to guarantee that all packets from the connection have died off, just in case the acknowledgement was lost. When the time goes off, TCP deletes the connection record. Now let us examine connection management from the server's viewpoint. The server dose a LISTEN and settles down to see that turns up. When a SYN comes in, it is acknowledged and server goes to the SYN RCVD state. When the server's SYN is itself acknowledged, the three-way handshake is complete and the server goes to the ESTABLISHED state. Data transfer can now occur. When the client is done, it dose, it does a CLOSE, which cause a FIN to arrive al the server. The server is then signaled. When the client's acknowledgement shows up, the server releases the connection and deletes the connection record.



Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

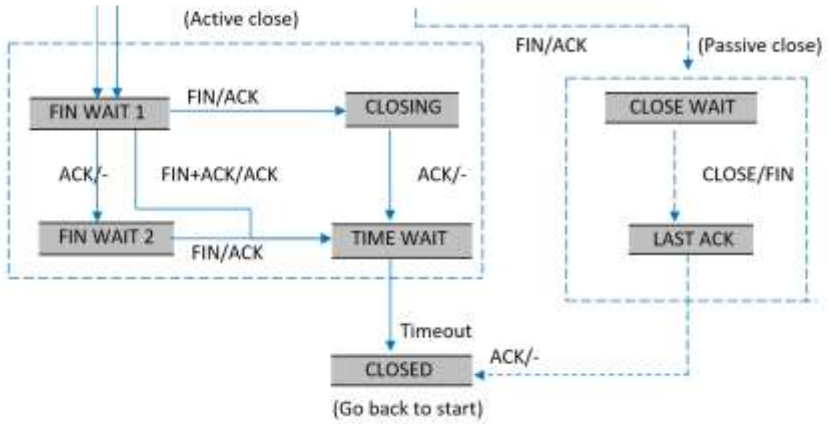


Figure: 2.3 TCP connection management finite state machines [6]

The heavy solid line in Figure 2.3 is the path for a client. The heavy dashed line is the path for a server. The light lines are unusual events. Each transition is labeled by the event causing it and the action resulting from it, separated by a slash.

2.6 TCP Transmission Policy

A simple transport protocol uses the following principle, send a packet and then wait for an acknowledgment from the receiver before sending the next packet. If the ACK is not received within a certain amount of time, retransmit the packet.

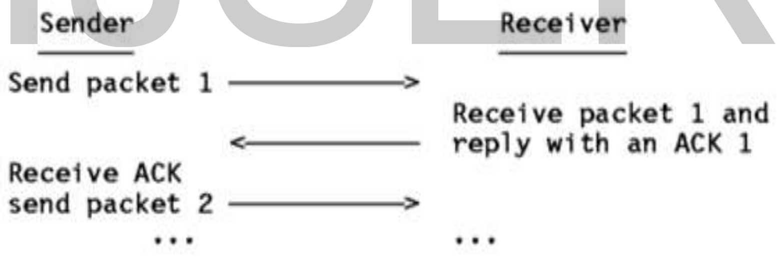
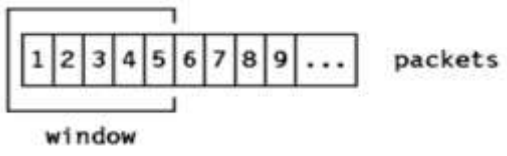

Sender
Receiver
Send packet 1
Receive packet 1 and reply with an ACK 1
Receive ACK send packet 2
...
...

Figure 2.4: Simple Transmission-Flows

While this mechanism ensures reliability, it only uses a part of the available network bandwidth. The transmission-flows is depicted in figure 2.5.

2.6.1: The Window Principle

Consider now a protocol where the sender groups its packets to be transmitted as in figure 2.5 and use the following rules:



Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

Figure 2.5: Window Principles, Message Packet

The sender may send all packets within the window without receiving an ACK, but must start a timeout timer for each of them.

- The receiver must acknowledge each packet received, indicating the sequence number of the last well-received packet.
- The sender slides the window on each ACK received.

In our example, the sender may transmit packets 1 to 5 without waiting for any acknowledgment:

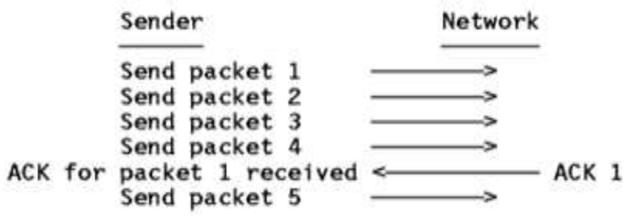


Figure 2.6: Window Principles

At the moment the sender receives the ACK 1 (acknowledgment for packet 1), it may slide its window to exclude packet 1. The window principle is depicted in figure 2.7. Message Packets at this point, the sender may also transmit packet 6 which is shown in figure 2.8.

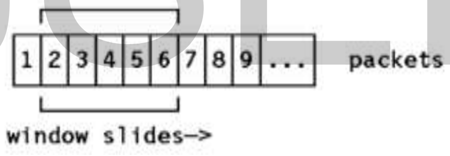


Figure 2.7: Message Packets

Imagine some special cases:

- Packet 2 gets lost: the sender will not receive an ACK 2, so its window will remain in the position 1 (as last picture above). In fact, as the receiver did not receive packet 2, it will acknowledge packets 3, 4 and 5 with an ACK 1, since packet 1 was the last one received ``in sequence''. At the sender's side, eventually a timeout will occur for packet 2 and it will be retransmitted. Note that reception of this packet by the receiver will generate an ACK 5, since it has now successfully received all packets 1 to 5 and the sender's window will slide four positions upon receiving this ACK 5.
- Packet 2 did arrive, but the acknowledgment gets lost: the sender does not receive ACK 2, but will receive ACK 3. ACK 3 is an acknowledgment for all packets up to 3 (including packet 2) and the sender may now slide his window to packet 4.

This window mechanism ensures:

- Reliable transmission.
- Better use of the network bandwidth (better throughput).
- Flow-control, as the receiver may delay replying to a packet with an acknowledgment, knowing its free buffers available and the window-size of the communication.

The Window Principle Applied to TCP. The above window principle is used in TCP, but with a few differences:

- As TCP provides a byte-stream connection, sequence numbers are assigned to each byte in the stream. TCP divides this contiguous byte stream into TCP segments to transmit them. The window principle is used at the byte level; that is, the segments sent and ACKs received will carry byte-sequence numbers and the window size is expressed as a number of bytes, rather than a number of packets.
- The receiver determines the window size, when the connection is established, and is variable during the data transfer. Each ACK message will include the window-size that the receiver is ready to deal with at that particular time.

The sender's data stream can now be seen as shown in figure 2.8:

Where:

A- Bytes that are transmitted and have been acknowledged.

B- Bytes that are sent but not yet acknowledged.

C- Bytes that may be sent without waiting for any acknowledgment.

D- Bytes that may not yet be sent.

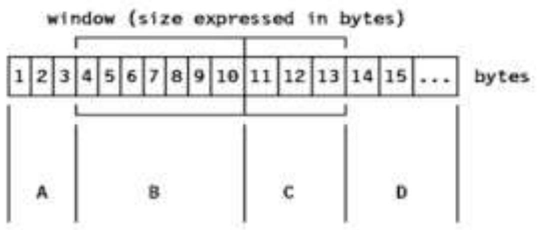


Figure 2.8: Window Principles Applied to TCP

Ensure that TCP will block bytes into segments, and a TCP segment only carries the sequence number of the first byte in the segment.

2.6.2 Acknowledgments and Retransmissions

TCP sends data in variable length segments. Sequence numbers are based on a byte count. Acknowledgments specify the sequence number of the next byte that the receiver expects to receive.

Now suppose that a segment gets lost or corrupted. In this case, the receiver will acknowledge all further well-received segments with an acknowledgment referring to the first byte of the missing packet. The sender will stop transmitting when it has sent all the bytes in the window. Eventually, a timeout will occur and the missing segment will be retransmitted. Suppose a window size of 1500 bytes, and segments of 500 bytes. A problem now arises, since the sender does know that segment 2 is lost or corrupted, but doesn't know anything about segments 3 and 4. The sender should at least retransmit segment 2, but it could also retransmit segments 3 and 4 (since they are within the current window).

1. Segment 3 has been received, and for segment 4 it is not known: it could be received, but ACK didn't reach us yet, or it could be lost also.
2. Segment 3 was lost, and received the ACK 1500 upon the reception of segment 4.

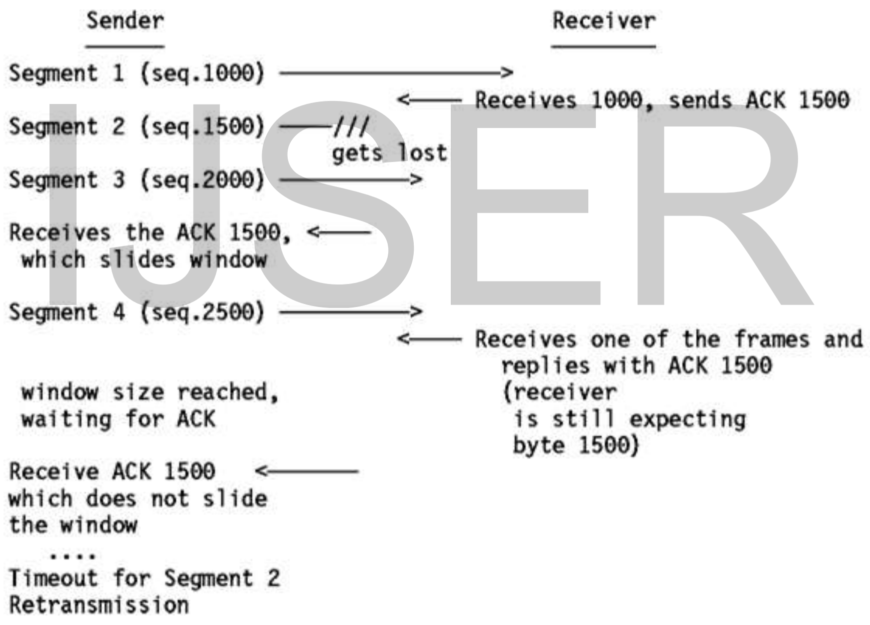

Sender
Receiver
Segment 1 (seq.1000)
Receives 1000, sends ACK 1500
Segment 2 (seq.1500)
gets lost
Segment 3 (seq.2000)
Receives the ACK 1500, which slides window
Segment 4 (seq.2500)
Receives one of the frames and replies with ACK 1500 (receiver is still expecting byte 1500)
window size reached, waiting for ACK
Receive ACK 1500 which does not slide the window
....
Timeout for Segment 2 Retransmission

Figure 2.9: Acknowledgment and Retransmission Process

2.7 TCP Timer Management

TCP uses multiple timers (at least conceptually) to do its work [6]. The most important of these is the retransmission timer. When a segment is sent, a retransmission timer is started. If the segment is acknowledged before the timer expires, the timer is stopped. If on the other hand, the timer goes off before the acknowledged comes in, the segment is retransmitted (and the timer started again). The question that arises is: how long should the timeout interval be? This problem is much more difficult in the Internet transport layer than in the generic data link protocols. In the latter case, the expected delay is highly predictable, so the timer can be set to go off just slightly after the acknowledgement is

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

expected ,as shown in figure (a). Since acknowledgements are rarely delayed in the link layer(due to lack of congestion), the absence of an acknowledgement at the expected time generally means either the frame or the acknowledgement has been lost.

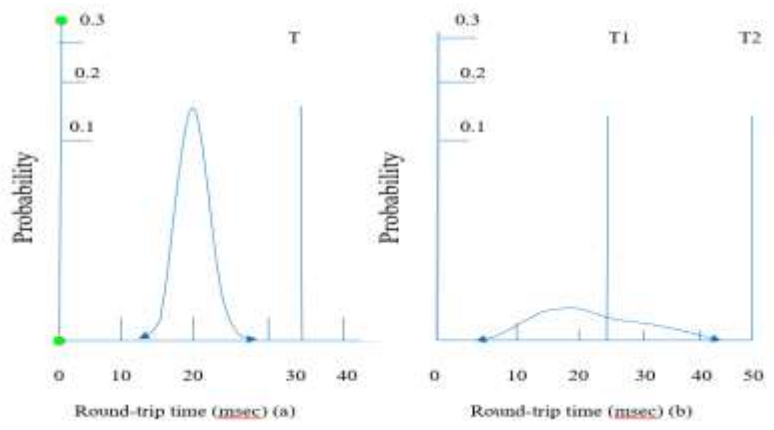


Figure 2.10: Timer management [6]

Figure 2.10: (a) Probability density of acknowledgement arrival times in the data link layer. (b) Probability density of acknowledgement arrival times for TCP. TCP is faced with a radically different environment. The probability density function for the time it takes for a TCP acknowledgement to come back looks more like Fig 2.10:(b) than Fig:(a). Determining the round-trip time to the destination is tricky. Even when it is known, deciding on the timeout interval is also difficult. If the timeout is set too short, say T1 in Fig 2.10 :(b) Unnecessary retransmission will occur, clogging the Internet with useless packets. If it is set too long, (e.g.., T2), performance will suffer due to the long retransmission delay whenever a packet is lost. Furthermore, the mean and variance of the acknowledgement arrival distribution can change rapidly within a few seconds as congestion builds up or is resolved. The solution is to use a highly dynamic algorithm that constantly adjusts the timeout interval, based on continuous measurements of network performance. The algorithm generally used by TCP is due to Jacobson (1988) and works as follows. For each connection, TCP maintains a variable, RTT that is the best current estimate of the round-trip time to the destination in question. When a segment is sent, a timer is started, both to see how long the acknowledgement takes and to trigger a retransmission if it takes too long. If the acknowledgement gets back before the timer expires, TCP measures how long the acknowledgement took, say M. It then updates RTT according to the formula

$$RTT = \alpha RTT + (1-\alpha)\, M \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots \textit{Equ.(1)}$$

When a smoothing factor determines how much weight is given to the old value. Typically $\alpha = 7/8$.

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

Even given a good value of RTT, choosing a suitable retransmission timeout is a nontrivial matter. Normally, TCP uses βRTT, but the trick is choosing β. In the initial implementations, β was always 2, but experience showed that a constant value was inflexible because it failed to respond when the variance went up. In 1988 , Jacobso proposed making β roughly proportional to the standard deviation of the acknowledgement arrival time probability density function so that a large variance means a large β, and vice versa. In particular, he suggested using the mean deviation as a cheap estimator of the standard deviation. His algorithm requires keeping track of another smoothed variable, D, the deviation, whenever an acknowledgement comes in, the difference between the expected and observed values, |RTT – M|, is computed. A smoothed value of this maintained in D by the formula

$$D = \alpha D + (1 - \alpha)\ |RTT - M\ | \ldots\ldots\ldots\ldots\ldots\ \textit{Equ.(2)}$$

Where may or may not be the same value used to smooth RTT. While D is not exactly the same as the standard deviation, it is good enough and Jacobson showed how it could be computed using only integer ads, subtracts and drifts - a big plus. Most TCP implementation now uses this algorithm and set the timeout interval to

$$\text{Timeout} = RTT + 4*D\ldots\ldots\ldots\ldots\ldots\ldots\ \textit{Equ.(3)}$$

The choice of the factor 4 is somewhat arbitrary, but it has two advantages. First, multiplication by 4 can be done with a single shift. Second, it minimizes unnecessary timeout and retransmission because less than 1 percent of all packets come in more than four standard deviations late. (Actually, Jacobson initially said to use 2, but later work has shown that 4 gives better performance) One problem that occurs with the dynamic estimation of RTT is what to do when a segment times out and is sent again. When the acknowledgement comes in , it is unclear whether the acknowledgement refers to the first transmission or a later one. Guessing wrong can seriously contaminate the estimate of RTT. Phil Karn discovered this problem the hard way. He is an amateur radio enthusiast interested in transmitting TCP/IP packets by ham radio, a notoriously unreliable medium (on a good day, half the packets get through). He made a simple proposal: do not update RTT on any segments that have been retransmitted. Instead, the timeout is double on each failure until the segments get through the first time. This fix is called Karn's algorithm. Most TCP implementation uses it.The retransmission timer is not the only timer TCP uses. A second timer is the persistence timer. It is designed to prevent the following deadlock. The receiver sends an acknowledgement with a window size of 0, telling the sender to wait. Latter, the receiver updates the window, but the packet with the update is lost. Now both the sender and the receiver are waiting for each other to do something. When the persistence timer goes off, the sender transmits a probe to the receiver. The response to the probe gives the window size. If it is still zero, the persistence timer is set again and the cycle repeats. If it is nonzero, data can now be sent.A third timer that some implementations use is the keep alive timer. When a connection has been idle for a long time, the keep alive, timer may go off to cause one side to check whether the other side is still there. If it fails to respond, the connection is terminated. This feature is controversial because it adds overhead and may terminate an otherwise healthy connection due to a transient network partition. The last timer used on each TCP connection is the one used in the TIMED WAIT state while closing. It runs for twice the maximum packet lifetime to make sure that when a connection is closed; all packets created by it have died off.

2.8 Wireless TCP and UDP

In theory, transport protocols should be independent of the technology of the underlying network layer. In particular, TCP should not care whether IP is running over fiber or over radio [6]. In practice, it does matter because most TCP implementations have been carefully optimized based on assumptions that are true for wired networks but that fail for wireless networks. Ignoring the properties of wireless transmission can lead to a TCP implementation that is logically correct but has horrendous performance. The principal problem is the congestion control algorithm. Nearly all TCP implementations nowadays assume that timeouts are caused by congestion, not by lost packets. Consequently, when a timer goes off, TCP shows down and sends less vigorously (e.g.., Jacobson’s slow start algorithm). The idea behind this approach is to reduce the network load and thus alleviate the congestion. Unfortunately, wireless transmission links are highly unreliable. They lose packets all the time. The proper approach to dealing with lost packets is to send them again, and as quickly as possible. Slowing down just makes matters worse, if, say, 20 percent of all packets are lost, then when the sender transmits 100 packets/sec, the throughput is 80 packets/sec. If the sender slows down to 50 packets/sec, the throughput drops to 40 packets/sec. In effect, when a packet is lost on a wired network, the sender should slow down. When one is lost on a wireless network, the sender should try harder. When the sender does not know what the network is, it is difficult to make the correct decision. Frequently, the path from sender to receiver is heterogeneous. The first 1000 km might be over a wired network, but the last 1 km might be wireless. Now making the correct decision on a timeout is even harder, since it matters where the problem occurred. A solution proposed by Bakne and Badrinath (1995), indirect TCP, is to split the TCP connection into two separate connections, as shown in Fig. the first connection goes from the sender to the base station. The second one goes from the base station to the receiver. The base station simply copies packets between the connections in both directions. The advantage of this scheme is that both connections are now homogeneous. Timeouts on the first connection can slow the sender down, whereas timeouts on the second one can speed it up. Other parameters can also be tuned separately for the two connections.

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

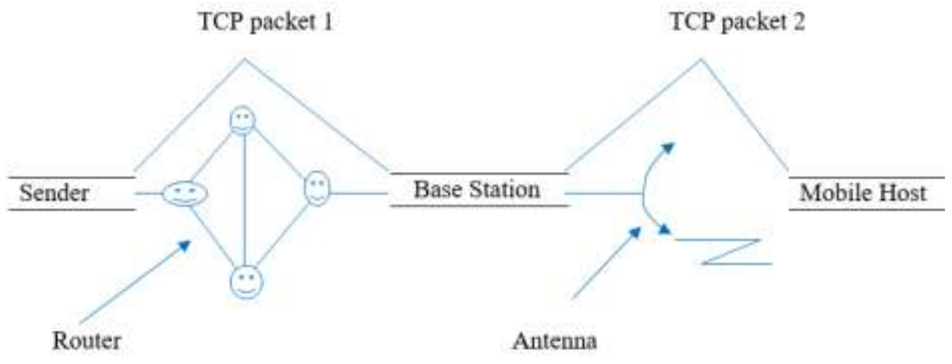


Figure 2.11: Splitting a TCP connection into two connections [6].

There are two connections. The disadvantage of the scheme is that it violates the semantics of TCP. Since each part of the connection is a full TCP connection, the base station acknowledges each TCP segment in the usual way. Only now receipt of an acknowledgement by the sender does not mean that the receiver got the segment, only that the base station got it. A different solution, due to Balakrishnan et al.(1995), does not break the semantics of TCP. It works by making several small modifications to the network layer code in the base station. One of the changes is the addition of a snooping agent that observes and caches TCP segments going out to the mobile host and acknowledgement coming back from it. When the snooping agent sees a TCP segment going out to the mobile host but does not see an acknowledgement coming back before its (relatively short) timer goes off, it just retransmits that segment, without telling the source that it is doing so. It also retransmits when it sees duplicate acknowledgement from from the mobile host go by, invariably meaning that the mobile host has missed something. Duplicate acknowledgements are discarded on the spot, to avoid having the source misinterpret them as congestion. One disadvantage of this transparency, however, is that if the wireless link is very loss, the source may time out waiting for an acknowledgement and invokes the congestion control algorithm. With indirect TCP, the congestion control algorithm will never be started unless there really is congestion in the wired part of the network The Balakrishnana et al. paper also has a solution to the problem of lost segments originating at the mobile host. When the base station notices a gap in the inbound sequence numbers, it generates a request for a selective repeat of the missing bytes by using a TCP option. Using these fixes, the wireless link is made more reliable in both directions, without the source knowing about it and without changing the TCP semantics. While UDP does not suffer from the same problems as TCP, wireless communication also introduce difficulties for it. The main trouble is that programs use UDP expecting it to be highly reliable. They know that no guarantees are given, but they still expect it to be near perfect. In a wireless environment, UDP will be far from perfect. For programs that can recover from lost UDP messages but only at considerable cost, suddenly going from an environment where messages theoretically can be lost but rarely are, to one in which they are constantly being lost can result in a performance disaster.

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

Wireless communication also affects areas other than just performance. For example, how does a mobile host find a local printer to connect to, rather than use its home printer? Somewhat related to this is how to get the www page for the local cell, even if its name is not known. Also, WWW page designers tend to assume lots of bandwidth is available, Putting a large logo on every page becomes counterproductive if it is going to take 10 sec to transmit over a slow wireless link every time the page is referenced, irritating the users no end. As wireless networking becomes more common, the problem of running TCP over it become more acute.

2.9 TCP in Wireless or Cellular Mobile System

In recent years there has been a strong interest in extending the Internet access technologies to wireless (Cellular Mobile System links). There are many advantages in using these technologies. For example they enable user mobility and network access anytime from anywhere. They can have a high bit error rate (BER), often combined with fading, can be asymmetric in nature (different forward and return channel bandwidth) or have a high propagation delay. Not all of these characteristics might be present in any given link but all of them, or even a subset of them, pose difficulties to the traditional TCP protocol.

2.10 Link Characteristics that Impact TCP Performance

High Bit Error Rate

There are two main problems when using TCP over a channel that has a high BER. The first is TCP's inability to distinguish between packet losses due to congestion and due to corruption; the second is that some TCP variants cope badly with multiple packet losses per congestion window.

Losses due to Corruption: TCP has been developed for reliable terrestrial links which have a very low BER. Because of the low probability of corrupted packets, all TCP congestion control strategies take lost packets as an indication of congestion. For every lost packet the TCP sender cuts down its transmission rate at least by half. In the case of TCP Tahoe, or when the losses are so severe that a timeout occurs, the data transmission drops back to one packet per round trip time.

Multiple Losses per Congestion Window: Some TCP variants (especially Reno) cope badly when several packets per congestion window are lost. Consecutive losses lead to a consecutive halving of the congestion window which is undesirable in wireless as well as in wire line environments. If multiple packets are lost, recovery can take a considerable amount of time.

2.11 High Propagation Delay

Cellular Mobile System links can have a very high or very variable propagation delay, depending on the altitude of Mobile user and the number of hops necessary for a connection. There are three main problems that TCP faces when used over high delay links:

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

- Slow Congestion Window Growth
- Long Time to Recover from Lost Packets
- Receive Window Limitation and
- Variable Propagation Delay

`Slow Congestion Window Growth`: The round trip time is a vital parameter for the congestion window growth. During Slow Start, congestion window doubles per round trip time and during Congestion Avoidance, congestion window is increased by one per round trip time. Because of this dependency between round trip time and congestion window increase, it is obvious that the higher the round trip time the slower the congestion window growth. This is especially profound during the Slow Start phase which is meant to probe quickly for available bandwidth.

`Long Time to Recover from Lost Packets`: Since the loss detection is based on an exchange of acknowledgments it always takes at least one round trip time for the sender to detect the loss of one packet and to retransmit it. The retransmission timeout value is also based on the round trip time as well as the variation of the round trip time samples. Therefore the larger any one of these values is, the higher the retransmission timeout will be. A high timeout value means a potentially long idle time during which the connection waits for the timeout to expire and for data transmission to recommence. Finally, as already mentioned, each loss is taken as a congestion indication. After a loss the transmission rate is reduced.

`Receive Window Limitation`: For high delay (or high bandwidth) links this value is quite large and it is possible that TCP will never be able to use the available link bandwidth. This is caused by the receive window limitation. The advertised receive window is always the upper bound for the congestion window growth. Unfortunately there is only a 16 bit field in the TCP header reserved to transmit this receive window to the sender.

`Variable Propagation Delay`: Not too much research has been done up to date to investigate TCP behavior over networks with variable propagation delay.

2.12 Error Control

Error control is a necessary function in communication networks. In general, several different error-handling mechanisms co-exist in the same network. Some errors are caused by short-lived noise in a specific location of the network that causes individual bits in the data stream to be altered. This is a problem of the physical transmission of information, for example thermal noise in the electronic components can cause errors. These types of errors are called bit errors, noise-induced errors or transmission errors. A second type of error occurs on a higher layer in the network, where packets are switched through different links. If links are overloaded, packets May have to be dropped due to

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

limited buffer space in the routers or switches, this is known as congestion losses. There are two main approaches for handling errors and losses:

- Retransmission & Forward Error Correction (FEC)

2.13 Retransmission

When retransmissions are used, all packets in which errors are detected are discarded. Hence, losses are caused by packets being discarded either due to transmission errors or due to congestion. Packets that have been discarded must be retransmitted by means of automatic repeat request (ARQ) protocols. The retransmission of lost data can also be handled in different ways, one approach is that the acknowledgements specify exactly the missing packets, and only those packets are retransmitted, which is known as selective repeat.

2.14 Forward Error Correction

The second approach to handling errors and losses is forward error correction, where redundant information is proactively sent to the receiver in order to allow it to correct errors. The redundant information is generated by means of channel coding. In forward error correction (FEC) [7], a receiver can use an error correcting code. Which automatically corrects certain errors? In theory, it is possible to correct any error automatically. Error correcting codes, however, are more sophisticated than error detection codes and require more redundancy bits.The concept underlying error correction can be most easily understood by examining the simplest case: single-bit errors. Single-bit errors can be detected by the addition of a redundant (parity) bit. A single additional bit can detect single-bit errors in any sequence of bits because it must distinguish between only two conditions: error or no error. A bit has two states (0 and 1). These two states are sufficient for this level of detection. But what if we want to correct as well as detect single-bit error? Two states are enough to detect an error but not to correct it. An error occurs when the receiver reads a 1 bit as 0 or a 0 bit as 1. To correct the error, the receiver simply reverses the value of the altered bit. To do so, however, it must know which bit is in error. The secret of error correction, therefore, is to locate the invalid bit or bits.For example, to correct a single bit error in an ASCII character, the error correction code must determine which of the 7 bits has changed. In this case, we have to distinguish between eight different states: no error, error in position 1, error in position 2 and so on, up to error in position 7. To do so requires enough redundancy bits to show all eight states. Channel coding is used in many commercial systems, for example compact discs and mobile communication systems. There are several types of channel codes, and the ones used in this thesis are block codes. In this section alternative codes are shortly described.

2.15 Block Codes

A block code is applied to a block of data symbols and generates a number of redundant symbols, a.k.a. parity symbols. Each symbol consists of one or more bits, depending on the code. The error correction capability is higher for codes with more parity symbols, but the parity symbols also use

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

some of the transmission capacity. Well-known examples of block codes are Hamming codes and Reed-Solomon codes.

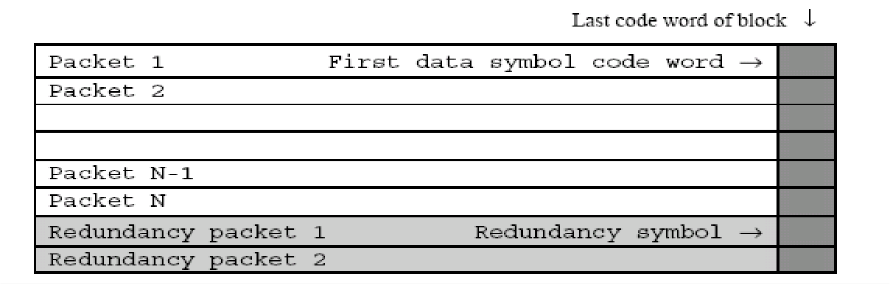


Figure 2.12: Block coding technique [7]

To correct bit errors a block of bits is encoded and transmitted over a link, then the decoder attempts to correct bit errors if any are present. If the number of bit errors within a block is lower than the code can correct, the original data block is recovered. In packet loss recovery a block of packets is encoded to generate packets of parity information, i.e. the first symbols in each data packet are encoded to produce the first symbols in each of the parity packets as shown in Figure 2.12 .

Hamming Code

Hamming code provides a practical solution. The Hamming code [7] can be applied to data units of any length and uses the relationship between data and redundancy bits that can be added to the end of the data unit or interspersed with the original data bits.

For the no of redundant bits $2^r >= m+r+1$ inequality must be satisfied. Here m=no of data bit and r=no of redundant bit. If m=7 to satisfy the inequality r=4. In figure 10.14 these bits are placed in position 1, 2, 4 and 8. For clarity in the example below, these bits are r1, r2, r4 and r8.

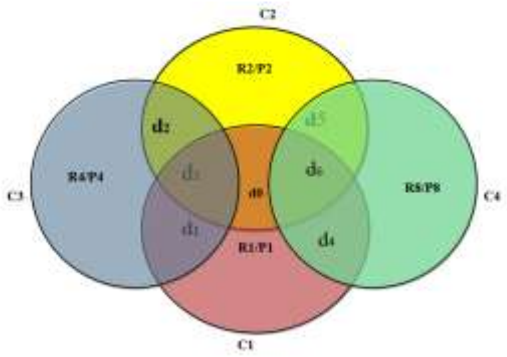


Figure 2.13: Redundancy & Check bit calculation System

1	2	3	4	5	6	7	8	9	10	11
R1	**R2**	**D0**	**R4**	**D1**	**D2**	**D3**	**R8**	**D4**	**D5**	**D6**

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

0 1 2 3 4 5 6 7 8 9 10

Table 2.1: Calculating the check bit for Hamming code.

Decimal	Binary	Check Bits(First 1 of corresponding position)	Check bits depends on(all the no contains 1 of that bit position)
1	0001	√	1,3,5,7,9,11
2	0010	√	2,3,6,7,10,11
3	0011		
4	0100	√	4,5,6,7
5	0101		
6	0110		
7	0111		
8	1000	√	8,9,10,11
9	1001		
10	1010		
11	1011		

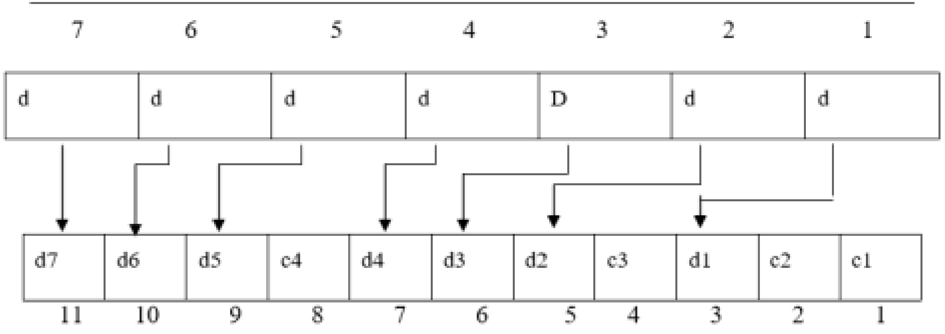


Figure 2.14: Constructing H (11, 7) from 7 bit data frame.

```
Calculation The r Values
```

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

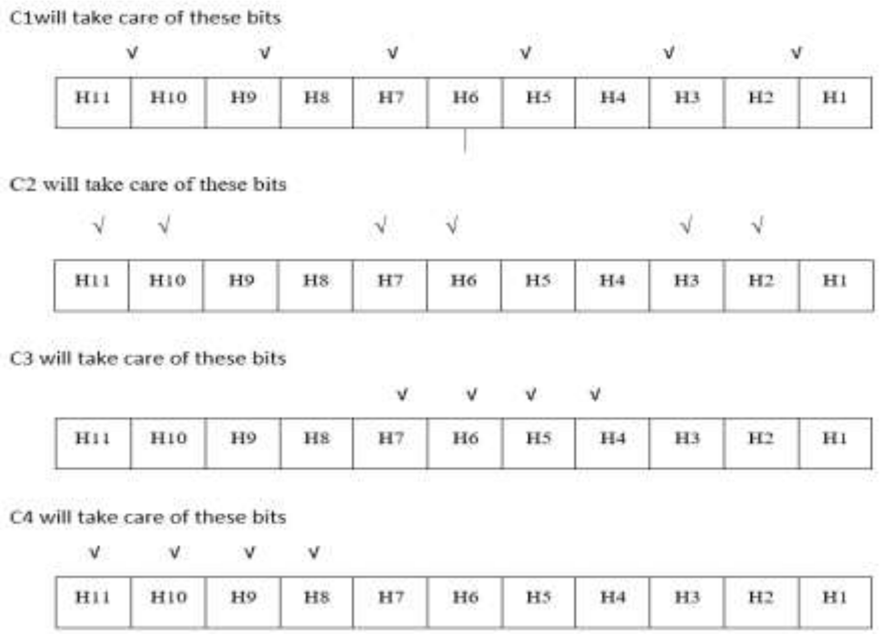

C1will take care of these bits
C2 will take care of these bits
C3 will take care of these bits
C4 will take care of these bits
H11
H10
H9
H8
H7
H6
H5
H4
H3
H2
H1

Figure 2.15: Redundancy bits calculation

The original character is in its appropriate position in the 11 bit unit frame. The even parity calculation is used form those data bit.

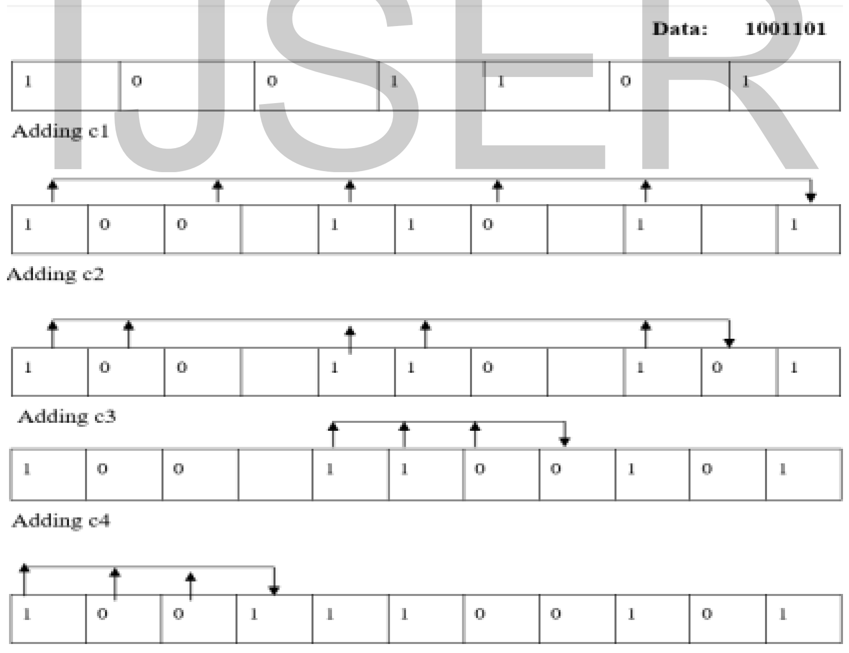


Figure 2.16: Adding check bits

The receiver takes the transmission and recalculates 4 new parity bits, using the same set of bits used by the sender plus the relevant parity r bit for each set. Then it assembles the new parity values into a binary number in order of position (r8, r4, r2, r1). In the given example, this step gives a binary number 0111 (7 decimal), which is the precise location of the bit error. 1000 is equivalent to decimal

8. So bit position 8 is in error. When the receiver identify that an error has occur then the receiver just inverse the bit and calculate the value. Error correction is shown in figure 2.19.

Error Detection and Correction

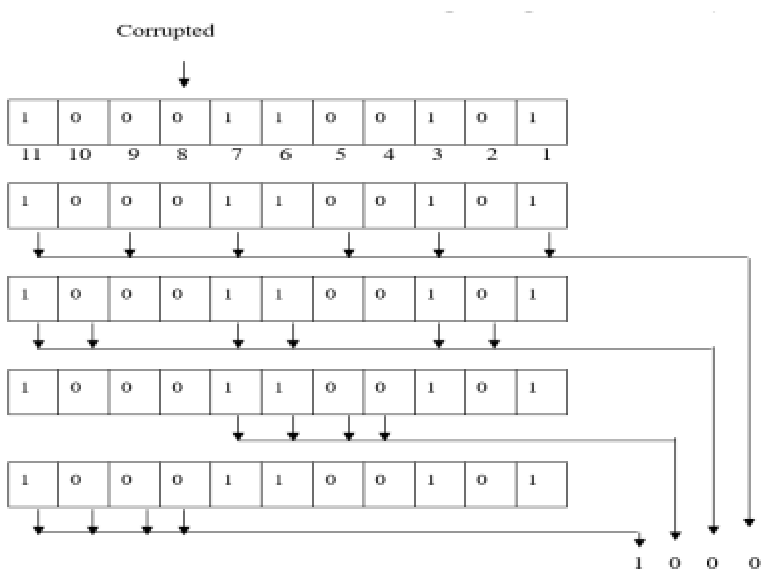


Figure 2.17: Error correction mechanism

2.16 Fountain Codes

The decoding complexity of block codes in general increases rapidly with the code length. Recent progress in the area of channel codes has lead to the development of codes with decoding complexity that only grows linearly with the code length. These are actually a form of low-density parity check codes that were initially proposed by Gallagher in 1963, but practical codes have only been developed in the 1990's. For the erasure channel these codes are known as fountain codes due to their application in the digital fountain concept for distribution of large files.

2.17 Convolution Codes

A digital finite impulse response filter performs the encoding. The data does not have to be divided into blocks before the encoding, as opposed to block codes. The decoding is made by Viterbi algorithm [8]. Where different nodes in a trellis represent the states of the encoding filter, and the transitions between different states are represented as edges.

2.18 Turbo Codes

Turbo codes are based on combining several encoders using interleaving to produce codes of very long lengths with limited decoding complexity. The basic codes of the turbo codes can either be block codes or convolution codes. Decoding is made by means of message passing between multiple decoders, similarly to the decoding of low-density parity check codes. For end-to-end FEC it does not have any clear advantages.

CHAPTER 3

TCP CONGESTION CONTROL METHODOLOGY

3.1 Introductory Description

When the load offered to any network is more than it can handle, congestion builds up. The Internet is no exception. We will discuss algorithms that have been developed over the past quarter of a century to deal with congestion. Although, the network layer also tries to manage congestions, most of the heavy lifting is done by TCP because the real solution to congestion is to slow down the data rate [6]. In theory, congestions can be dealt with by employing a principle borrowed from physics: the law of conservation of packets. The idea is to refrain from injecting a new packet into the network until an old one leaves (i.e. is delivered). TCP attempts to achieve this goal by dynamically manipulating the window size. The first step in managing congestion is detecting it. In the old days detecting congestions was difficult. A timeout caused by a lost packet could have been caused by either noise on a transmission line or packet discard at a congested router. Telling the difference was difficult. Nowadays, packet loss due to transmission errors is relatively rare because most long-haul trunks are fiber (although wireless networks are a different story). Consequently, most transmission

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

timeouts on the internet are due to congestion. All the Internet TCP algorithms assume that timeouts are caused by congestion and monitor timeouts for signs of trouble the way miners watch their carriers. Before discussing how TCP reacts to congestions, let us first describe what it does to try to prevent congestion from occurring in the first place, when a congestions is established, a suitable window size has to be chosen. The receiver can specify a window based on its buffer size. If the sender sticks to this window size, problems will not occur due to buffer overflow at the receiving end, but they may still occur due to internal congestion within the network.

In Fig 3.1: a, b .we see this problem illustrated hydraulically. In Fig 3.1: (a). We see a thick pipe leading to a small-capacity receiver. As long as the sender does not send more water than the bucket can contain, no water will be lost. In Fig 3.1: (b). The limiting factor is not the bucket capacity, but the internal carrying capacity of the network. If too much water comes in too fast, it will back up and some will be lost (in this case by over following the funnel)

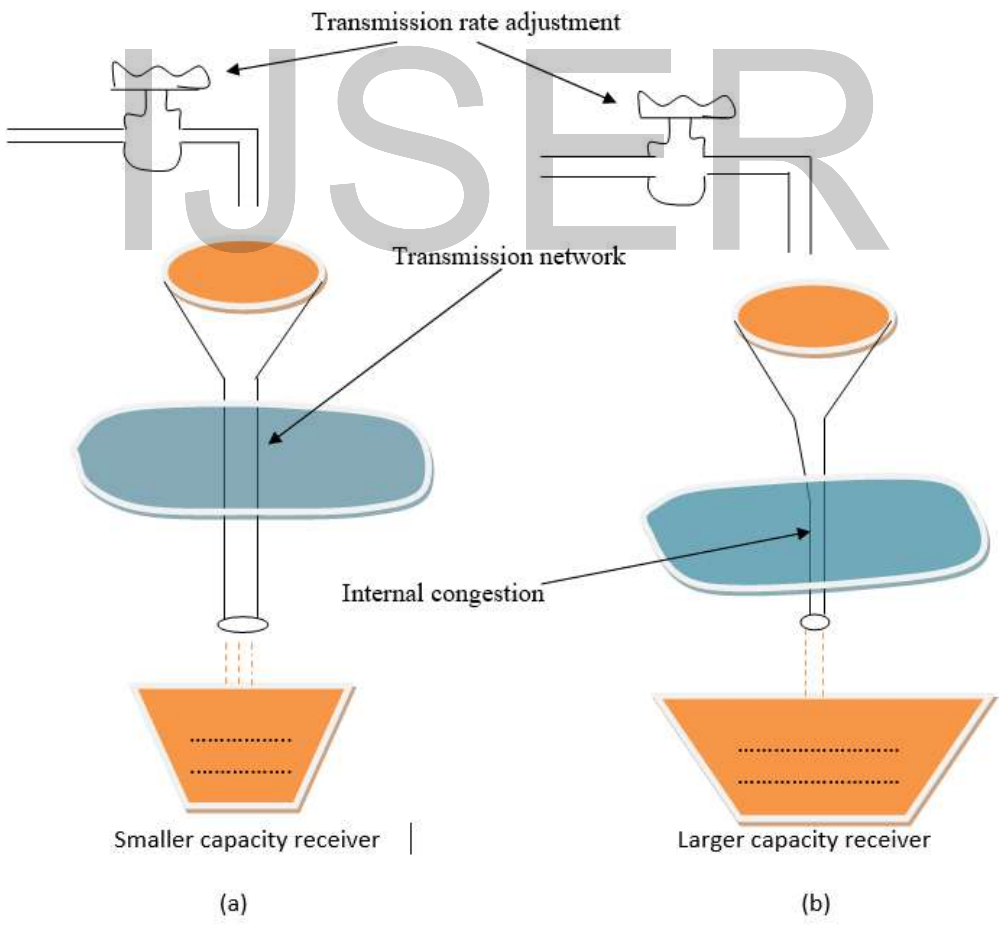


Figure 3.1: (a) A fast network feeding a low-capacity receiver (b) A slow network feeding a high-capacity receiver [6].

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

The Internet solution is to realize that two potential problems exist at network capacity and receiver capacity and to deal with each of them separately. To do so, each sender maintains two windows: the window that receiver has granted and a second window that is congestion window. Each reflects the number of bytes the sender may transmit. The number of bytes that may be sent is the minimum of the two windows. Thus, the effective window is the minimum of what the sender thinks is all right and what the receiver thinks is all right. If the receiver says a "Send 8 KB" but the sender knows that bursts of more than 4 KB clog the net-work, it sends 4 KB. On the other hand, if the receiver says "Send 8KB" and the sender knows that bursts of up to 32KB get through effortlessly, it sends the full 8 KB requested.

When a connection is established, the sender initializes the congestion window to the size of the maximum segment in use on the connection. It then sends one maximum segment. If this segment is acknowledged before the timer goes off, it adds one segment's worth of bytes to the congestion window to make it two maximum size segments and sends two segments. As each of these segments is acknowledged, the congestion window is increased by one maximum segment size. When a congestion window is n segments, if all n are acknowledged on time. The congestion window is increased by the byte count corresponding to n segments. In effect, each burst acknowledged doubles the congestion window.

The congestion window keeps growing exponentially until either a timeout occurs or the receiver's window is reached. The idea is that if bursts of size, say 1024, 2048 and 4096 bytes work fine but a burst of 8192 bytes gives a timeout, the congestion window should be set to 4096 to avoid congestion. As long as the congestion window remains at 4096 no bursts longer than that will be sent, no matter how much window space the receiver grants. This algorithm is called slow start, but it isn't slow at all (Jacobson 1988). It is exponential. All TCP implementation are required to support it.

Now let us look at the Internet congestion control algorithm. It uses a third parameter, the threshold, initially 64 KB, in addition to the receiver and congestion windows. When a timeout occurs, the threshold is set to half of the current congestion window and the congestion window is reset to one maximum segment. Slow start is then used to determine what the network can handle, except that exponential growth stops when the threshold is hit. From that point on, successful transmissions grow the congestion window linearly (by one maximum segment for each burst) instead of one per segment. In effect, this algorithm is guessing that it is probably acceptable to cut the congestion window in half, and then it gradually works its way up from there.

As an illustration of how the congestion algorithm works, see Figure 3.2: The maximum segment size here is 1024 bytes. Initially, the congestion window was 64 KB, but a timeout occurred, so the threshold is set to 32 KB and the congestion window to 1 KB for transmission 0 here. The congestion window then grows exponentially until it hits the threshold (32 KB). Starting then, it grows linearly.

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

Transmission 13 is unlucky (it should have known) and a timeout occurs. The threshold is set to half the current window (by now 40 KB, so half is 20 KB), and slow start is initiated all over again. When the acknowledgements from transmission 14 start coming in, the first four each double the congestion window, but after that, growth becomes linear again.

If no more timeout occur, the congestion window will continue to grow up to the size of the receiver's window. At that point, it will stop growing and remain constant as long as there are no more timeouts and receiver's window does not change size. As an aside, if an ICMP SOURCE QUENCH packet comes in and is passed to TCP, this event is treated the same way as a timeout

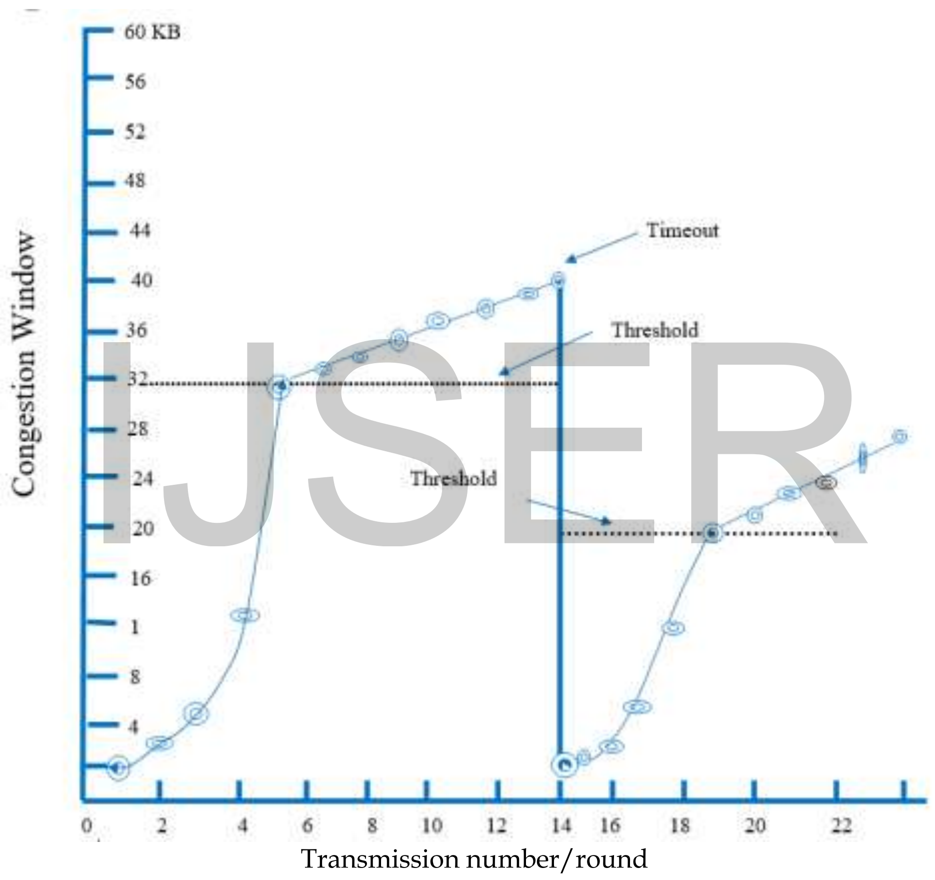


Figure 3.2: An Example of the Internet Congestion algorithm [6].

3.2 Description of Congestion Control Parameter

This section provides the definition of several terms that will be used throughout the remainder of this document.

SEGMENT: A segment is any TCP/IP data or acknowledgment packet (or both).

SENDER MAXIMUM SEGMENT SIZE (SMSS):

The SMSS is the size of the largest segment that the sender can transmit. This value can be based on the maximum transmission unit of the network, the path MTU discovery algorithm, RMSS (see next item), or other factors. The size does not include the TCP/IP headers and options**.**

RECEIVER MAXIMUM SEGMENT SIZE (RMSS):

The RMSS is the size of the largest segment the receiver is willing to accept. This is the value specified in the MSS option sent by the receiver during connection startup or, if the MSS option is not used, 536 bytes the size does not include the TCP/IP headers and options.

FULL-SIZED SEGMENT:

A segment that contains the maximum number of data bytes permitted (i.e., a segment containing SMSS bytes of data).

RECEIVER WINDOW (rwnd): The most recently advertised receiver window.

CONGESTION WINDOW (cwnd):

A TCP state variable that limits the amount of data a TCP can send. At any given time, a TCP must not send data with a sequence number higher than the sum of the highest acknowledged sequence number and the minimum of cwnd and rwnd.

INITIAL WINDOW (IW): The initial window is the size of the sender's congestion window after the three-way handshake is completed.

LOSS WINDOW (LW): The loss window is the size of the congestion window after a TCP sender detects loss using its retransmission timer.

RESTART WINDOW (RW): The restart window is the size of the congestion window after a TCP restarts transmission after an idle period.

FLIGHT SIZE: The amount of data that has been sent but not yet acknowledged.

3.3 Congestion control Technique

Tahoe TCP

Two variables are used to achieve congestion control in Tahoe one is congestion window (cwnd) and the slow start threshold (ssthresh) [9]. The congestion window governs the amount of data a connection is currently allowed to transmit, while the slow start threshold determines in which phase of the congestion control procedure the connection is currently in. Tahoe's congestion control mechanism consists of two phases: Slow Start, which is executed at the beginning of the connection and after every packet loss, and Congestion Avoidance, which is entered when cwnd reaches the value of ssthresh. In addition to the new congestion control strategy TCP Tahoe also proposes a new retransmission mechanism called Fast Retransmit. Fast Retransmit takes the reception of a threshold number of duplicate acknowledgments as loss indication and retransmits the packet immediately. A timeout is still used to recover losses; if not enough duplicate acknowledgments arrive. After a packet loss cwnd is set to 1 and Slow Start is re-entered. Ssthresh is set to half of the congestion window size when the loss occurred, which is considered as a safe transmission rate. Those who implements Tahoe follows some algorithm that are mentioned bellow shortly.

`RFC 896 – Nagle’s Algorithm`

Implementation of Nagle’s Algorithm ensures only 1 segment can be outstanding any time, ensuring that no continual streams of segments are sent. Of course, there are situations where data does have to be pushed through to the receiver TCP quickly.

`Karn’s Algorithm`: Using this algorithm, the ambiguity problem of TCP is specified and solved.

`Jacobson’s Congestion Control Algorithms`: Jacobson opinions; a host should only inject 1 packet into the network for every packet removed. If this “packet conservation principle” would have obeyed, congestion collapse would become the exception then the rule [9]. Jacobson found in general packet conservation that can be violated when

- ✓ Connections don’t get to an equilibrium state where full windows of data are exchanged.
- ✓ Sender Injects new packet before old packet has exited due to erroneous retransmits.
- ✓ Equilibrium can’t be reached because of resource limits along the path.

By implementing mechanisms in TCP to enforce packet conservation, TCP can better deal with network congestion. The 3 mechanisms implemented are [9] –

`Slow-start:`

Mechanism: A congestion window (cwnd) is added to the connection. When starting initially or restarting after segment loss, set cwnd to less than or equal to 2 segments. On receiving an ACK, increase cwnd by 1 when transmitting data, send the minimum of the receiver's advertised window and cwnd. Using slow start increases the congestion window gradually but exponentially until the sender hits the receiver's advertised window size and reaches equilibrium. This allows the connection to reach their equilibrium state while eliminating sudden influxes of data into the network that risks stability. TCP Receiver sends a duplicate ACK immediately when an out–of–order data carrying segment arrives. TCP Sender immediately retransmits what appears to be the missing segment when 3 duplicate ACKs arrive.

`Reno TCP – Fast Recovery`

To compliment Fast Retransmit, Jacobson developed a complementary Fast Recovery algorithm [14]. The inclusion of Fast Recovery into Tahoe TCP resulted in a new TCP released referred to as “Reno” TCP [16]. In Tahoe TCP, triggering of the Fast Retransmit mechanism also triggers congestion

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

avoidance (and by extension slow start), resulting in a slow recovery from a cwnd with a size of 1. Jacobson proposes that the following algorithm be followed when duplicate ACKs are received [10].

1. When the 3rd duplicate ACK is received by the sender [11],

$$\text{set ssthresh} \leq \max\left(\frac{Amount\ of\ UnACKed\ Data\ in\ Send\ Window}{2}, 2\times Sender\ Max\ Segment\ Size\right)$$

2. Fast retransmit the lost segment (as per Fast Retransmit) and set

$$\text{cwnd=old cwnd} + 3 \times \text{Sender Maximum Segment Size}$$

3. For each additional duplicate ACK received beyond the 3 which triggered Fast Retransmit, increment cwnd by Sender Maximum Segment Size and transmit another segment if allowed by cwnd and the receiver's advertised window.

4. For the next ACK arriving that acknowledges new data, set cwnd=ssthresh. This ACK should be elicited by the segment retransmitted in step 2 and should acknowledge all intermediate segments sent between the lost segment and the receipt of the 3rd duplicate ACK, providing none of these segments were lost too. Continue congestion avoidance with reduced cwnd.

Essentially, the arrival of the duplicate ACK stream indicates that the ACK "clock" (congestion control and avoidance) is preserved – there is no need to probe the network via slow start to start an ACK stream flowing to "strobe" additional packets into the network. In addition, the duplicate ACKs also indicate that the segments which triggered these ACKs have left the network and so the packet conservation principle allows the injection of additional packets into the network. A step 1–3 reflects this.

By skipping slow start, a sending TCP Layer only drops to ½ its original transfer rate instead of going to slow start, allowing it to recover faster. Note that Fast Recovery works best when only 1 segment is lost within a window – studies done by [12] found that Fast Recovery does not recover efficiently from multiple segment losses within a single window.

RFC 2018- Selective Acknowledgements, a Reno TCP extension:

Given that Fast Retransmit/Recovery only retransmits 1 segment, the loss of multiple segments in a send window leads to inefficient recovery as Reno has to go through multiple Fast Retransmits and a full retransmission timeout before recovery occurs via slow start/congestion avoidance [12, 13]. Selective Acknowledgement (SACK) is an optional Reno extension that relieves this problem [14]. In essence, the multiple retransmit problem arises because the sender transmits segments that are already have being correctly received by the source (successive fast retransmits). By allowing ACKs sent by the receiver to contain a SACK option field listing the out of order segments received, the

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

sender can deduce and retransmit only lost segments. Subsequent arrivals of missing data are acknowledged normally by the receiver.

The SACK option field is composed of sets of two 32 bit sequence numbers. Each set represents a contiguous block of received out of order data – the 1st number represents the "left edge" (i.e. 1st sequence number) whereas the 2nd number represents the "right edge" (i.e. sequence number following the last sequence number of this block). Sequence numbers below the block (i.e. smaller then Left Edge of Block sequence number) and above the block (i.e. larger then or equal to Right Edge of Block sequence number) represent missing data at the receiver.

Provided that both sender/receiver agree to use SACK, the receiver should generate an ACK with SACK option for every valid segment that arrives when the receiver has out of order data in its buffers. Also note that since SACK is an advisory option, the receiver may discard data reported in a SACK option if they run out of buffer space. For every SACK ACK, the receiver should define SACK blocks in the SACK option field such that [14] –

• The first SACK block specified must contain the segment which triggered this ACK, even if that segment is going to be discarded and the receiver has already discarded adjacent segments. The only exception is if that segment advanced the ACK Number in the header. This allows the sender to determine whether these SACK blocks represent the most recent change in the receiver's buffers.

• Except for the newest segment, all SACK blocks must not report any old data which is no longer actually held by the receiver.

• Since many distinct SACK blocks as possible in the SACK option, there may be insufficient space to specify all missing blocks.

• The SACK option is filled out by repeating the most recently reported SACK blocks (based on first SACK blocks in the previous ACK) that are not subsets of a SACK block already included in the SACK option being constructed. This assures that any segment remaining part of a non–contiguous block of data held by the data receiver is reported in at least three successive SACK options. After the first SACK block, the following SACK blocks in the SACK option may be listed in arbitrary order.

A Sender receiving an ACK with a SACK option should record the SACK blocks for future reference to allow selective segment retransmission. For example, assume that each segment in the transmission queue has a "SACKed" bit. When an ACK with a SACK option arrives, the sender will turn on the "SACKed" bit for segments SACKed by the specified SACK blocks and have their retransmission timers disabled. Any segment with an off SACKed bit and is less than the highest SACKed segment is available for retransmission when timeout occurs.

However, should a timeout occur on a segment retransmission, the sender should turn off all SACKed bits and re–enable their retransmission timers as the receiver may have discarded their out of order segments? The sender must then start retransmitting segments from the left edge of the window. Segments are not released from the retransmission queue until it is ACKed normally.

Although RFC 2018 does not specify how SACK interoperates with Reno congestion control algorithms, RFC 2581 states that the inclusion of SACK should not alter the essence of Reno's congestion control algorithms [10]. This involves [14] –

• When the first loss in a window of data is detected, the sender must set

$$\text{set ssthresh} \leq \max\left(\frac{Amount\ of\ UnACKed\ Data\ in\ Send\ Window}{2}, 2\times Sender\ Max\ Segment\ Size\right)$$

• Until all missing segments in the window are successfully retransmitted, the number of segments transmitted in each RTT MUST is no more than half the number of outstanding segments when the loss was detected.

• Once all lost segments in a window are successfully retransmitted, cwnd must be set to no more than ssthresh and congestion avoidance must be used to further increase cwnd.

• Loss in two successive windows of data, or the loss of a retransmission, should be taken as two indications of congestion and, therefore, cwnd (and ssthresh) must be lowered twice in this case.

SACK is often incorporated into RENO [15] by using SACK information from duplicate ACKs to determine what segments to retransmit during step 3 of the Fast Recovery algorithm defined above [16]. Via this method, simulations show that SACK, compared to Tahoe, Reno or New Reno, recovers the most efficiently from multiple segments being dropped in a single window.

RFC 2582 – New Reno TCP:

For TCP implementations not supporting SACK, Janey Hoe [17] proposed an alternative scheme to recover from multiple lost segments by responding to partial ACKs. When segments are lost within a single window, the first new information available to the sender comes when the sender receives an ACK for the Fast–Retransmitted packet. In the case of a single segment loss, the ACK clears the entire send window. In the case of a multiple segment drop, the ACK only acknowledges some of the segments sent before the Fast Retransmit. This is a partial ACK. Hoe suggested that the arrival of partial ACKs during Fast Recovery should indicate another lost segment which should also be retransmitted. This involves changes to steps 1 and 4 of the Fast Recovery algorithm defined previously and adds a step 5. It defines 2 new variables, "recover" and "send high". Send high's initial value is the initial send sequence number. After each retransmit timeout, the highest sequence number transmitted so far is recorded in the variable "send high". The altered steps, as defined in RFC 2582 and implemented in "New Reno" TCP [18], are –

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

1. When the third duplicate ACK is received and the sender is not already in the Fast Recovery procedure, check to see if those duplicate ACKs cover more than "send_high". If they do, set
2. set ssthresh $\leq \max\left(\frac{Amount\ of\ UnACKed\ Data\ in\ Send\ Window}{2}, 2\times Sender\ Max\ Segment\ Size\right)$ and record the highest sequence number transmitted in the variable "recover", and go to Step 2.
3. If the duplicate ACKs do not cover "send_high", then do nothing. Do not enter Fast Retransmit/Recovery, change ssthresh in anyway, carryout out step 2 or 3.
4. When an ACK arrives that ACKs new data, this ACK could be the acknowledgment elicited by the retransmission from step 2, or elicited by a later retransmission. If this ACK ACKs all data up to and including "recover", then all segments in the send window have being ACKed. Set $cwnd = ssthresh\ and\ exit\ Fast\ Recovery$. If this ACK does not ACK all data up to and including "recover", this is a partial ACK. Retransmit the first unACKed segment.

5. $Set\ cwnd\ = \ old\ cwnd - Amount\ of\ Data\ ACKed\ +\ 1\ Maximum\ Segment\ Size$
Send a new segment if permitted by the new value of cwnd. For the first partial ACK that arrives during Fast Recovery, reset the retransmit timer for all segments.
After a retransmit timeout, it records the highest sequence number transmitted in the variable "send_high" and exit the Fast Recovery procedure if applicable.

Essentially, a "recover" variable is used to determine whether the ACK returned in step 4 is a partial ACK and by extension, whether more then 1 segment was dropped. If the ACK was a partial ACK, the sender effectively stays in Fast Recovery mode and retransmits 1 segment per RTT for every Partial ACK received until all dropped segments are retransmitted or a retransmission timeout on a dropped segment forces the sender into slow start/congestion avoidance mode. The adjustments made to cwnd in step 4 also ensure that when Fast Recovery eventually ends, only ssthresh amount of data is in flight in the network. In addition, send_high is used to prevent multiple fast retransmits triggered due to duplicate ACKs arriving from a previous Fast Retransmit cycle. By recording the highest sequence number transmitted in the last Fast Retransmit cycle (step 5), incoming duplicate ACKs are checked to ensure that they did arise from out of order segments arriving at the receiver during the current segment transmission cycle, in which case Fast Retransmit/Recovery is triggered (step 1).

The algorithm proposed above is only 1 variant of New Reno. Other variants differ in whether they reset segment retransmission timers once (as in step 4 above) or reset them for every partial ACK received. The number of packets retransmitted for each partial ACK received may also vary. The variant above is conservative in that it only transmits 1 segment per partial ACK. More aggressive variants may transmit multiple segments per partial ACK received [19].

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

A study [15] shows that New Reno can recover efficiently from more then 1 missing segment in the sender window, provided that the number of missing segments was small. When large numbers of segments are lost, New Reno performs as poor as Reno since it only transmits only 1 segment per RTT. Consequently, a retransmission timeout usually occurs before all lost segments are transmitted, resulting in New Reno being forced into Slow Start like Reno.

Vegas TCP – A Proactive Approach:

TCP Vegas is an extension to TCP Reno with modified or new versions of Reno's mechanisms. Congestion Avoidance and Slow Start Mechanisms which are corrected those inefficiencies [20]. These are specified below.

Vegas' Modified Retransmission Mechanism:

Implementations of Reno TCP use a coarse–grained timer (around 500ms) that does not measure RTTs accurately or trigger segment retransmissions on time, resulting in a retransmission mechanism that does not adapt quickly to network congestion. Vegas improves the mechanism's accuracy via –

- More accurate RTT measurement – The sender records the system time for every segment sent. When ACKs arrive, the sender estimates new RTTs by subtracting the recorded system time of ACKed segments with the current system time, yielding more accurate estimates of RTT.

- Timelier Retransmission– The sender retransmits segments in 2 additional situations.

When a duplicate ACK is received, Vegas checks if the difference with current system time and the recorded sent system time for the relevant segment is greater then its RTO value, or

Non–duplicate ACKs are received and it is the 1st or 2nd non duplicate ACK after a segment retransmission, Vegas checks if the difference with current system time and the recorded sent system time for the recently retransmitted segment is greater than its RTO value.

If yes to either of the above, Vegas retransmit the relevant segment immediately without waiting for the arrival of 3 duplicate ACKs or a coarse retransmission timeout.

In essence, ACK arrivals are used as triggers to determine if timeouts have occurred. This allows Vegas to retransmit sooner and more accurately then what Reno's coarse timers allow. In the event that ACKs are lost en masse in the system, Vegas can still fall back on Reno's coarse timers for segment retransmission.

- More accurate treatment of cwnd – Under the current Reno specifications, cwnd can be reduced multiple times for multiple segment losses that occurred before the last cwnd reduction. Given that multiple segment losses that happened before the last cwnd reduction do not indicate that the network is congested for the current reduced cwnd size, cwnd should not be halved again since it has already being reduced. To correct this, Vegas only decreases cwnd on segment retransmission if the recorded timestamp for when the retransmitted segment was last transmitted was after the last cwnd reduction, preventing excessive reduction of cwnd.Vegas's New Congestion Avoidance Mechanism

Reno's Congestion Avoidance mechanism is reactive – cwnd is only adjusted when segment losses occur. In effect, Reno must create losses for it to adjust to changes in the level of network congestion. In contrast, Vegas takes a proactive approach and attempts to anticipate network congestion and adjust cwnd prior to segment losses occurring to reduce retransmissions. Vegas does this by implementing

• Congestion Detection Mechanism – To anticipate network congestion, Vegas takes constant measures of network throughput. If the measured throughput drops significantly at any point in time, network congestion has occurred. The precise algorithm is defined below.

1. Let BaseRTT = RTT when connection is not congested. This is usually the minimum RTT ever achieved on a connection.

   Expected Throughput = Data Sent During 1 RTT / BaseRTT

   Where Data Sent During 1 RTT = Send Window size at any time.

   α = A throughput threshold (in bps) determining when there is too little data in the network and cwnd needs to be increased.

   β = A throughput threshold (in bps) determining when there is too much data in the network and cwnd needs to be reduced.

2. For every RTT (except during Slow Start), Vegas calculates Actual Throughput by Recording the size of a marked segment and the system timestamp when this segment was sent and also by recording the timestamp when the first ACK acknowledging the marked segment arrives.

The RTT for the marked segment is computed using the 2 timestamps gathered. CalculateActual $Throughput = Marked\ Segment\ Size\ /\ MeasuredRTT$

3. $Every\ time\ an\ ActualThroughput\ is\ calculated, cwnd\ is\ adjusted\ by$

   3.1 Let $\Delta\,Throughput = ExpectedThroughput - ActualThroughput$

   3.2 If $\Delta\,Throughput\ is\ negative, set\ BaseRTT = MeasuredRTT$ and quit this process.

   3.3 If ΔThroughput is positive, compare ΔThroughput to α and β,

   If $\Delta\,Throughput < \alpha, increase\ cwnd\ linearly\ during\ next\ RTT.$

   $\Delta\,Throughput > \beta, increase\ cwnd\ linearly\ during\ next\ RTT.$

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

$\alpha < \Delta Throughput < \beta, leave\ cwnd\ unchanged.$

As a result, by continually measuring the actual throughput experienced by the sender and comparing it to the throughput expected when the network is not congested, Vegas can determine whether there is too much/too little data in the network and adjust cwnd downward/upward as required respectively.

Vegas' Modified Slow Start Mechanism:

Due to the reactive nature of Reno's Slow Start, it is very expensive in terms of losses when available bandwidth is high and cwnd is large. This occurs as Slow Start doubles the size of cwnd every RTT. When losses occur as cwnd oversteps the available network bandwidth, losses on the order of ½ of cwnd are expected [20]. Vegas reduce this loss by incorporating Congestion Detection into Reno's Slow Start.

Modified Slow Start Mechanism – In effect, Vegas only allows cwnd to grow exponentially for every other RTT – the additional RTT cycle is used to obtain a valid measure of actual throughput from which it can compare with the expected throughput to determine whether cwnd should continue to grow exponentially or switch to linear growth. Hence –

1. Start Slow Start normally with $cwnd = 1$.
2. Let , $BaseRTT = RTT\ of\ 1^{st}$ segment sent
   $ExpectedThroughput = Size\ of\ 1st\ Segment\ Sent\ /\ BaseRTT.$

   γ = A throughput threshold (in bps) determining when the data transmission rate as defined by cwnd is approaching the available bandwidth the network has to offer. cwnd shall be increased linearly, not exponentially, after this point.

3. For all subsequent RTT cycles, if cwnd has grown in the last RTT cycle, keep cwnd constant, transmit data normally and calculate ActualThroughput as specified in step 2 of the Congestion Detection Mechanism defined above.
4. For all subsequent RTT cycles, if cwnd has not grown in the previous RTT cycle, calculate $\Delta Throughput = ExpectedThroughput - ActualThroughput$.
   If (4.1) $\Delta Throughput > \gamma$, increase cwnd linearly from this point on. Exit Slow Start and continue adjusting cwnd using the Congestion Avoidance Mechanism defined previously.
   $\Delta Throughput < \gamma$, continue increasing cwnd exponentially. (4.2) Increase cwnd as required

   and transmit segments as allowed by cwnd.

   In essence Vegas's modified Slow Start halts the exponential growth of cwnd when its measured throughput falls below the expected throughput by a margin ofγ, indicating that the sender is nearing the bandwidth limit which the network can sustain without congestion.

By switching to linear growth at this point, any losses are limited to the size of the linear increase of cwnd, which is much lower then the ½ cwnd loss encountered by Reno for big bandwidth connections.

Simulations studies [21,22,23,24,25] of Vegas performance supports the claim made by Brakmo, O'Malley and Peterson that Vegas gives an increase throughput over other TCP implementations like Reno and Tahoe. However, the gains given by Vegas are situational – in situations when heavy congestion hits, Vegas falls back on Reno's coarse gain retransmission timer, eliminating performance gains of Vegas. In addition, Vegas also have the bonus of easy implementation. Unlike Reno extensions like SACK, Vegas mechanisms only need to be implemented in the sender– performance gains are still present even if the receiver is running a non–Vegas TCP implementation [20].

Currently, no different studies have being made on how Vegas mechanisms would interact with SACK. However, it is envisaged that due to Vegas' low amount of retransmissions, SACK implementation with Vegas would only present very limited improvements [20].

3.4 Congestion Control Algorithms

This section defines the four congestion control algorithms using these algorithm we have implemented our congestion control simulation: slow start, congestion avoidance, fast retransmit and fast recovery, developed in. In some situations it may be beneficial for a TCP sender to be more conservative than the algorithms allow, however a TCP must not be more aggressive than the following algorithms allow (that is, must not send data when the value of cwnd computed by the following algorithms would not allow the data to be sent).

Slow Start and Congestion Avoidance:

The slow start and congestion avoidance algorithms must be used by a TCP sender to control the amount of outstanding data being injected into the network [11]. To implement these algorithms, two variables are added to the TCP per-connection state. The congestion window (cwnd) is a sender-side limit on the amount of data the sender can transmit into the network before receiving an acknowledgment (ACK), while the receiver's advertised window (rwnd) is a receiver-side limit on the amount of outstanding data. The minimum of cwnd and rwnd governs data transmission. Another state variable, the slow start threshold (ssthresh), is used to determine whether the slow start or congestion avoidance algorithm is used to control data transmission, as discussed below. Beginning transmission into a network with unknown conditions requires TCP to slowly probe the network to determine the available capacity, in order to avoid congesting the network with an inappropriately large burst of data. The slow start algorithm is used for this purpose at the beginning of a transfer, or after repairing loss detected by the retransmission timer. IW, the initial value of cwnd, must be less than or equal to 2*SMSS bytes and must not be more than 2 segments. We note that a non-standard, experimental TCP extension allows that a TCP may use a larger initial window (IW), as defined in equation 1

$$IW = min\,(4 * SMSS, max\,(2 * SMSS, 4380\ bytes)) \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots Equ.\,(1)$$

With this extension, a TCP sender may use a 3 or 4 segment initial window, provided the combined size of the segments does not exceed 4380 bytes. We do not allow this change as part of the standard defined by this document. However, we include discussion of (1) in the remainder of this document as a guideline for those experimenting with the change, rather than conforming to the present standards for TCP congestion control. The initial value of ssthresh may be arbitrarily high (for example, some implementations use the size of the advertised window), but it may be reduced in response to congestion. The slow start algorithm is used when cwnd < ssthresh, while the congestion avoidance algorithm is used when cwnd > ssthresh. When cwnd and ssthresh are equal the sender may use either slow start or congestion avoidance. During slow start, a TCP increments cwnd by at most SMSS bytes for each ACK received that acknowledges new data. Slow start ends when cwnd exceeds ssthresh (or, optionally, when it reaches it, as noted above) or when congestion is observed. During congestion avoidance, cwnd is incremented by 1 full-sized segment per round-trip time (RTT). Congestion avoidance continues until congestion is detected. One formula commonly used to update cwnd during congestion avoidance is given in equation 2:

$$cwnd \mathrel{+}= SMSS * SMSS / cwnd \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots Equ.\ (2)$$

This adjustment is executed on every incoming non-duplicate ACK. Equation (2) provides an acceptable approximation to the underlying principle of increasing cwnd by 1 full-sized segment per RTT. (Note that for a connection in which the receiver acknowledges every data segment, (2) proves slightly more aggressive than 1 segment per RTT, and for a receiver acknowledging every-other packet, (2) is less aggressive.) Implementation Note: Since integer arithmetic is usually used in TCP implementations, the formula given in equation 2 can fail to increase cwnd when the congestion window is very large (larger than SMSS*SMSS). If the above formula yields 0, the result should be rounded up to 1 byte. Implementation Note: older implementations have an additional additive constant on the right-hand side of equation (2). This is incorrect and can actually lead to diminished performance. Another acceptable way to increase cwnd during congestion avoidance is to count the number of bytes that have been acknowledged by ACKs for new data. (A drawback of this implementation is that it requires maintaining an additional state variable.) When the number of bytes acknowledged reaches cwnd, then cwnd can be incremented by up to SMSS bytes. Note that during congestion avoidance, cwnd must not be increased by more than the larger of either 1 full-sized segment per RTT, or the value computed using equation 2. Implementation Note: some implementations maintain cwnd in units of bytes, while others in units of full-sized segments. The latter will find equation (2) difficult to use, and may prefer to use the counting approach discussed in the previous paragraph.When a TCP sender detects segment loss using the retransmission timer, the value of ssthresh must be set to no more than the value given in equation 3:

$$\text{ssthresh} = \max(\text{FlightSize} / 2, 2*\text{SMSS}) \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \textit{Equ. (3)}$$

As discussed above, FlightSize is the amount of outstanding data in the network. Implementation Note: an easy mistake to make is to simply use cwnd, rather than FlightSize, which in some implementations may incidentally increase well beyond rwnd. Furthermore, upon a timeout cwnd must be set to no more than the loss window, LW, which equals 1 full-sized segment (regardless of the value of IW). Therefore, after retransmitting the dropped segment the TCP sender uses the slow start algorithm to increase the window from 1 full-sized segment to the new value of ssthresh, at which point congestion avoidance again takes over.

Fast Retransmit/Fast Recovery:

A TCP receiver should send an immediate duplicate ACK when an out- of-order segment arrives. The purpose of this ACK is to inform the sender that a segment was received out-of-order and which sequence number is expected. From the sender's perspective, duplicate ACKs can be caused by a number of network problems. First, they can be caused by dropped segments. In this case, all segments after the dropped segment will trigger duplicate ACKs. Second, duplicate ACKs can be caused by the re-ordering of data segments by the network (not a rare event along some network paths. Finally, duplicate ACKs can be caused by replication of ACK or data segments by the network. In addition, a TCP receiver should send an immediate ACK when the incoming segment fills in all or part of a gap in the sequence space. This will generate more timely information for a sender recovering from a loss through a retransmission timeout, a fast retransmit, or an experimental loss recovery algorithm, such as NewReno. The TCP sender should use the "fast retransmit" algorithm to detect and repair loss, based on incoming duplicate ACKs. The fast retransmit algorithm uses the arrival of 3 duplicate ACKs (4 identical ACKs without the arrival of any other intervening packets) as an indication that a segment has been lost. After receiving 3 duplicate ACKs, TCP performs a retransmission of what appears to be the missing segment, without waiting for the retransmission timer to expire. After the fast retransmit algorithm sends what appears to be the missing segment, the "fast recovery" algorithm governs the transmission of new data until a non-duplicate ACK arrives. The reason for not performing slow start is that the receipt of the duplicate ACKs not only indicates that a segment has been lost, but also that segments are most likely leaving the network (although a massive segment duplication by the network can invalidate this conclusion). In other words, since the receiver can only generate a duplicate ACK when a segment has arrived, that segment has left the network and is in the receiver's buffer, so we know it is no longer consuming network resources. Furthermore, since the ACK "clock" is preserved, the TCP sender can continue to transmit new

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

segments (although transmission must continue using a reduced cwnd). The fast retransmit and fast recovery algorithms are usually implemented together as follows.

1. When the third duplicate ACK is received, set ssthresh to no more than the value given in equation 3.
2. Retransmit the lost segment and set cwnd to ssthresh plus 3*SMSS. This artificially "inflates" the congestion window by the number of segments (three) that have left the network and which the receiver has buffered.
3. For each additional duplicate ACK received, increment cwnd by SMSS. This artificially inflates the congestion window in order to reflect the additional segment that has left the network.
4. Transmit a segment, if allowed by the new value of cwnd and the receiver's advertised window.
5. When the next ACK arrives that acknowledges new data, set cwnd to ssthresh (the value set in step 1). This is termed "deflating" the window. This ACK should be the acknowledgment elicited by the retransmission from step 1, one RTT after the retransmission (though it may arrive sooner in the presence of significant out- of-order delivery of data segments at the receiver). Additionally, this ACK should acknowledge all the intermediate segments sent between the lost segment and the receipt of the third duplicate ACK, if none of these were lost. Note: This algorithm is known to generally not recover very efficiently from multiple losses in a single flight of packets. One proposed set of modifications to address this problem can be found in

Re-starting Idle Connections:

A known problem with the TCP congestion control algorithms described above is that they allow a potentially inappropriate burst of traffic to be transmitted after TCP has been idle for a relatively long period of time. After an idle period, TCP cannot use the ACK clock to strobe new segments into the network, as all the ACKs have drained from the network. Therefore, as specified above, TCP can potentially send a cwnd-size line-rate burst into the network after an idle period recommends that a TCP use slow start to restart transmission after a relatively long idle period. Slow start serves to restart the ACK clock, just as it does at the beginning of a transfer. This mechanism has been widely deployed in the following manner. When TCP has not received a segment for more than one retransmission timeout, cwnd is reduced to the value of the restart window (RW) before transmission begins. For the purposes of this standard, we define RW = IW. We note that the non-standard experimental extension to TCP defined in defines RW = min (IW, cwnd), with the definition of IW adjusted per equation (1) above. Using the last time a segment was received to determine whether or not to decrease cwnd fails to deflate cwnd in the common case of persistent HTTP connections. In this case, a WWW server receives a request before transmitting data to the WWW browser.

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

The reception of the request makes the test for an idle connection fail, and allows the TCP to begin transmission with a possibly inappropriately large cwnd. Therefore, a TCP should set cwnd to no more than RW before beginning transmission if the TCP has not sent data in an interval exceeding the retransmission timeout.

Loss Recovery Mechanisms:

A number of loss recovery algorithms that augment fast retransmit and fast recovery has been suggested by TCP researchers. While some of these algorithms are based on the TCP selective acknowledgment (SACK) option, such as others do not require SACKs. The non-SACK algorithms use "partial acknowledgments" (ACKs which cover new data, but not all the data outstanding when loss was detected) to trigger retransmissions. These enhanced algorithms are implicitly allowed, as long as they follow the general principles of the basic four algorithms outlined above. Therefore, when the first loss in a window of data is detected, ssthresh must be set to no more than the value given by equation (3). Second, until all lost segments in the window of data in question are repaired, the number of segments transmitted in each RTT must be no more than half the number of outstanding segments when the loss was detected. Finally, after all loss in the given window of segments has been successfully retransmitted, cwnd must be set to no more than ssthresh and congestion avoidance must be used to further increase cwnd. Loss in two successive windows of data, or the loss of a retransmission, should be taken as two indications of congestion and therefore, cwnd (and ssthresh) must be lowered twice in this case.

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

CHAPTER 4

PROBLEM FORMULATION & RESULT DISCUSSION

4.1 Congestion control in TCP

Congestion Control techniques such as Tahoe, Reno, New Reno, SACK, Vegas are used to control congestion in network level. These techniques have been developed by different scientist or network researchers. Those who research those techniques have been successful to long ago. These techniques are very old opinion. Although, there are lot of techniques to control congestion that actually is not published for security purposes, that's why we have known only aforesaid techniques. We have simulated only two techniques such as Tahoe TCP and Reno TCP by Java programming language after studying those congestion control techniques.

4.2 Optimum Forward error correction

Block codes can recover as many data packets as number R of redundancy packets are added in a block. But those redundancy packets will also steal some of the available bandwidth that could have been used to transmit data packets instead. So, for addition of more redundancy packets than it is necessary to recover losses, the throughput will be negatively affected and network load unnecessarily increased.

On the other hand, if addition of less redundancy packets than which is needed, besides that, .recovery of those packets that have been lost is not possible, by increasing the network load with some redundancy traffic it is not be able to successfully use them.

The question springs to mind how much redundancy should add when applying FEC to maximize the TCP throughput? In other words it is the "optimum" value of R that Maximizes for link-level FEC.

Block coding error correction technique use hamming code for forward error correction. For any type of block coding technique N=K+R.

Here N is the total no of bits or packets.

K is the total no of data bits or packets.

R is the total no of redundant bits or packets.

For this thesis hamming code is used for this coding $2^r >= m+r+1$.

Here m is the total no of data bits and r is the total no of redundant bits.

So to use hamming code it must satisfy this inequality. So hamming is represented by H (m+ r, m).

Hamming code can only handle single bit error. So it for large no of data bit is considered as a block and for this a no of redundant bit is added then the benefit of forward error correction can not found. To find the benefit of forward error correction hamming code is used for a small no of bits. For this thesis the block is considered of 7 bits so to satisfy the inequality the value of r is 4. Because $2^4 >= 7+4+1$ is right.

So the hamming code for this thesis is H (11, 7).

If throughput T is represented in terms of packet/s:

Then T = (No of acknowledged packet*1024*1024)/time…………………*Equ.(1)*

If FEC is used with TCP then the no of retransmission is reduced for wireless network. For wireless network the bit error probability is high so for error in packets the no of retransmission is higher. So form eq(1) the throughput degrades for wireless network. It is the main cause of using FEC with TCP.

For using FEC the encoding and decoding algorithm will be such that the encoding and decoding algorithm should be faster because TCP maintain timeout. If the encoding and decoding algorithm is slower then the performance also degrades because the no of retransmission is higher. So from eq(1) the faster encoding and decoding algorithm will improve the performance.

4.3 TCP Routing Agent

There are several approaches in conventional routing algorithm in traditional wire line networks, and some ideas from these are also used in ad-hoc networks. Among the traditional approaches the followings are used generally:

- Link State

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

- Distance Vector
- Source routing
- Flooding

Destination Sequenced Distance Vector-DSDV:

DSDV is a distance vector routing protocol. Each node has a routing table that indicates for each destination, which is the next hop and number of hops to the destination. Each node periodically broadcast routing updates. A sequence no is used to tag each route. It shows the freshness of the route: a route with higher sequence no is more favorable. In addition among two routed with the same sequence no, the one with less hops is more favorable. If a node detects that a route to a destination has broken, then its hop no is set to infinity and its sequence no updated but assigned an odd number: even numbers corresponds to sequence numbers of connected paths.

Ad-hoc On Demand Distance Vector – AODV:

AODV is a distance vector type routing. It does not require nodes to maintain routes to destination that are not actively used. As long as the endpoints of a communication connection have valid routes to each other, AODV does not play a role. The protocol uses different messages to discover and maintain links: Route Requests (RREQ), Route Replies (RREPs), and Route Errors (RERRs). These message types are received via UDP, and normal IP header processing applies. AODV does not allow handling unidirectional links.

Dynamic Source Routing – DSR:

Designed for mobile ad hoc networks with up to around two hundred nodes with possibly high mobility rate is the protocol works "on demand", i.e. without any periodic updates. Packets carry along the complete path they should take. This reduces overheads for large routing updates at the network. The nodes store in their cache all known routes. The protocol is composed of route discovery and route maintenance.

It consists of the following two steps

- Route discovery
- Route maintenance

Temporally Ordered Routing Algorithm-TORA:

This protocol is of the family of link reversal protocols. It may provide several routes between a source and a destination. TORA contains three parts: creating, maintaining and erasing routes. At each node, a separate copy of TORA is run per each destination. TORA builds a directed acyclic graph rooted at the destination. It associated a height with each node in the network. Messages flow

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

from nodes with higher height to those with lower heights. Routes are discovered using Query and Update packets.

4.4 TCP Congestion Control simulation details

Class summary

Router Class	This class is a simulation of network router
TCPReceiver Class	This is a simulation of network receiver
TCPSegment Class	This is a segment which can carry segment, ack
TCPSender Class	This is a base class for sender
TCPSenderTahoe Class	TCPTahoe implementation of sender
TCPSenderReno Class	TCPReno implementation of sender
TCPSimulator Class	The main class of Congestion control

Class Router

This class is a simulation of network router. Router routes the TCP segment that may be towards the sender or towards the receiver. Router silently enforces that no more packets let pass through than what the bottleneck capacity allow. The bottleneck resource that we have studied using this router is the buffer space which determines the maximum queue length. If more packets arrive than queue of router can hold, it will discord the access segment. If access segments arrive, router will drop only the data segment. There is no limitation of ACK segment. If more ack segment arrives, it simply routes towards the destination without discarding. In real world, all segments are subjected to the same treatment at the network level.

Constructor summary

Public Router (int mismatchRatio, int bufferSize)

This constructor function silently enforces that this router introduce a resource bottleneck in the network. The router buffer size is forced to be smaller than the mismatch ratio between the router's input and output links.

mismatchRatio: it is the transmission speed between input and output links.

bufferSize: The given buffer size for router's queue.

Method summary

public int getBottleneckCapacity()

This method is used for retrieving of the bottleneck resource of this router. This value tells us the maximum number of bytes that router can relay per unit of time. The router can transmit one

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

segment at a time and can hold up to the buffer size segments. So, the routers capacity is calculated as follows

$$(buffer\ size\ +1)\ *\ Maximum\ segment\ size.$$

Public void relay(TCPSegment[] packets_)

It keeps the segment up to buffer size and unaffected segment discards up to the mismatch ratio. I mean mismatch ratio is the transmission speed between input and output link. For example, if mismatch ratio is 10:1 of the pipe size, it means input link is 10 time faster than output link. Buffer size must be less than mismatch ratio. If arrival input 10 segments and buffer size is 7, three segments are discarded. Aging if 20 segments come, router r save1, 2……. 7number segment and 11 number segments will be transmitted.

`Class TCP Receiver`

The TCP receiver receives a segment and retransmits an acknowledgement. It is the main work of receiver. When receiver receives any segment, it has to work a lot to control congestion. Arrival segment may lost, may be duplicate acknowledgement. Receiver buffer may have previously buffered some segments. Those aforesaid problems have to be solved by receiver. Now I will describe different method and field which are essential for receiver work.

Field Summary

Protected int lastBufferedIdx It is the index of the last segment that has been buffered. -1 means there is no buffer segments.
Protected int lastByteRecvd This field make a record of a segment sequence number has been received.
Protected int nextByteExpected This is the currently expected segment from sender.
Protected int rcvWindow It is the receiver window size in bytes.
Protected TCPSegment[] rcvBuffer Receiver receives segments what goes from sender in receiver buffer.

`Constructor summary`

Public TCPReceiver() is used to initialize value in declared parameter. Here, we have initialized null value which indicates that there is no value initially in receiver buffer.

Method summary

public int receive(TCPSegment[] segments_, TCPSegment[] acks_)

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

It receives segment from sender and passes the acknowledgement segment. When receive function of TCPReceiver class is called from run function of TCPSimulator class, receiver checks all arrival segments which goes from sender before receiving by following mentioned way. TCPSegment type's Segment array and ACK array has been declared whose size 100 in TCPSimulator class. Receive function of receiver class will be called after calling send function of sender class and relay function of r in router class. When send function of sender class will be called, some segment will be kept in segment array including segment sequence number, ack in segment array. Receive function of TCPReceiver class will receive a number of segments that kept in TCPSegment array in receive buffer and TCPReceiver generate ack is kept in ack array. At first step, receive funtion will check arrival segments whether arrival segment is null or not? Null means lost segment. If segment array doesn't contain null, receive function can understand that there are some segments in segment array. After that it checks arrival segment sequence number. It itself ensures that whether arrival segment's sequence numbers are equal to expected next byte or not? If yes, it will set next expected sequence number by sum of arrival segment's sequence number plus its length. Then, it will check. Is there any segment in receiver buffer that has been previously buffered? if there is no previously buffered segments in receiver buffer, it will make a record of last byte received in sequence. On the other hand, if there are available previously buffered segments, it will check whether it has been created any gap by out of sequence or not? If yes, it will remove receiver buffer's segment one by one. To remove segment it will call check Buffered Segments function. But , if the segment is lost, this situation perform by out of Sequence Segment function.

protected TCPSegment outOfSequenceSegment(TCPSegment segment_)

This function will be called if segment is lost, receive function of receiver class also keep out of sequence segment in receive buffer. In this simulation we assume that all buffered segments are ordered in the ascending order of their sequence number. After receiving a out of sequence segment, receiver window will be decreased according to received segment's length. Then, an acknowledgement will be transmitted to sender. If a segment that comes to receiver from sender contains null, no work will be performed.

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

protected void checkBufferedSegments() It will check if the previously buffered out of sequence segment. If there is a previously buffered segment, it will remove previous buffered segments. Before removing, it will calculate next expected byte's sequence number and increase receiver window with removing segment's length. Remove system last until last buffer index's value will have larger or equal to zero. Obviously, this system only for one segment. But, if there would have more than one segment previously buffered in receiver buffer, in that situation, it had to shift remaining segments towards the beginning of array. After that it will remove another segment from receiver buffer in same way. if previously buffered segment's sequence number isn't equal to expected next byte. It won't do anything for that case. After that it will give an acknowledgement that indicates segment has been received by receiver.
public int getRcvWindow() It is a accessor for retrieving the current size of the available buffer spaces in bytes. I mean, It will return current receiver window.

`Class TCP Segment`

This is a segment. We have created segment by sequence number, length and ack. A segment can carry data or ack or both. In our simulation we have created a segment without giving data. Because of data is irrelevant. We don't have headache data. We need only send a segment. Whether data is not essential, but segment is essential. That's why we have made a segment.

Field summary

Public int seqNum = 0; It provides sequence number of segments
Public int length = 0; It keeps the length of segment.
Public boolean ack; It informs whether segment contains any ack or not?
Public boolean inError = false; It indicates whether segment is corrupted by an error or not? This also introduces an error checking mechanism for this simulation.

`Constructor summary`

Public TCPSegment(**int** seqNum_, **int** length_)

Public TCPSegment(**int** seqNum_, **int** length_, **boolean** ack_)

When sender sends a segment and receiver gives acknowledgement segment only then this constructor function is called. A new object which is actually a new segment is created by aforesaid parameter and kept in sender's segment array. Once it also transmits to receiver.

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

Class TCP Sender

TCP sender class is also interring connected to TCP Sender Tahoe and TCP Sender Reno. Tahoe class process acknowledgement segment and detect duplicate ACK, lost packet or another problem and also recover but when three duplicate acknowledgement is received by Tahoe, TCP Sender Reno class is called and recover loss segment.

Field Summary

protected int lastByteSent = -1; It is the pointer which points last byte that has been sent so far.
protected int lastByteAcked = -1; It is the pointer which points last byte that has been received so far.
protected int congWindow = TCPSimulator.*MSS*; Current congestion window size in bytes
protected int SSThresh = 65535; This is the constant parameter of Tahoe where slow start sending mode kick in.
protected static final int *SLOW_START* = 0; It is the type of the sending mode of sender initial stage.
protected static final int *CONG_AVOID* = 1; It is the type of the sending mode. It will be increased additively.
protected int sendMode = *SLOW_START*; It is the current send mode. Its default value is slow start.
protected static final int *TIMER_DEFAULT* = 3; It is a default value of timer, in our simulation its value is 3*RTT.
protected int timer = *TIMER_DEFAULT*; It is a retransmission of timer. The timer is activated at the beginning of the transmission cycle. When all outstanding segments are acknowledged, the timer is deactivated. When regular acknowledgement is received and there are still outstanding , non acknowledgement segment, the timer should be restarted.
protected int dupACKsGlobal = 0; It is the counter of duplicate acknowledgements over multiple subsequent RTT periods.

Method summary

public int getTotalBytesTransmitted() It is a accessor for retrieving total number of bytes what have been transmitted during simulation run. This also used for reporting purposes of sender.

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

public abstract void send(TCPSegment[] segments_, **int** rcvWindow_, **boolean** lostPacket_)

This is the abstract function of sender. It means to us that TCP Sender Class doesn't do any work by using send function but those who inherits this class only those class can do some work by using this send function. We have done this to use TCP Tahoe and TCP Reno class. TCP Sender gets something done by TCP Tahoe and TCP Reno class. But it itself won't do any thing. Sender sends two type segments. Obviously, Receiver will be able to distinguish between this.

public abstract int processAcks(TCPSegment[] acks_)

It is same as previous abstract meaning. It processes the acknowledgement segment what receiver has given for sender purposes.

`Class TCP Sender Tahoe`

Most of the work of sender has been done in TCP Sender Tahoe class.

`Method summary`

public int processAcks(TCPSegment[] acks_)

When process acknowledgement function is called, sender checks whether ACKs array is null or not? If yes, process ACK function is terminated. If ACKs array contain any ACK, sender has to ensure that whether this acknowledgement is for slow start or congestion avoidance. Because when any ACKs go to sender, sender doesn't know it is of which mode. Arrival ACKs may be slow start modes' or congestion modes'. So, to be ensured. Sender checks current send modes' value. If current send modes' value is equal to slow start modes' value, sender can understand that arrival ACKs is slow start modes'. After that process ACKs slow start function is called. On the other hand, If the value of current send mode is equal to the value of cong avoidance mode, sender understands that arrival ACKs is contestation avoidance modes'. After that if current send mode is in congestion avoidance mode, The function process ACKs Congestion avoidance will be called. After that global count of duplicate ACKs will be updated with local duplicate acknowledgement count. If the number of duplicate acknowledgements is greater than 2, the onThreeDuplicateACKs() function is called. If the last sending byte is equal to the last receiving byte, timer will be updated. Otherwise sender can understand that some segments are still outstanding. In this situation timer decrease and compare .if timer value is less than or equal to zero, sender can sender understand that time out has occurred. After that onExpiredTimeoutTimer() function is called.

protected boolean processAcksSlowStart(acks_[i_])

When process ACKS function is called, the following situation is created. Sender checks sequence number of arrival ACKs. If it is greater than expected sequence number, it will make equal. After that congestion window will increase with maximum segment size. When congestion window's value exceeds the SSThresh value, sender enters into the congestion avoidance and current send mode will be converted to congestion avoidance mode.

protected boolean processAcksCongestionAvoidance(acks_[i_]) When this function is called, it will check the sequence number of present acknowledgement comparing with last Byte received. If it is ok, congestion window will increase linearly. After that timer will be updated. If present arrival ACKs sequence number conflict with last Byte received ACK, it will be reported as duplicate acknowledgement.
onThreeDuplicateACKs() When this function is called, the slow start thresholds value will be half of congestion window value. Congestion window value will be equal to maximum segment size. This situation will encourage the sender to retransmit the oldest packet. But how is it possible? It is simple. Send mode has to be set to slow start mode. After that timer will be updated by calling function resetMonitoringVariables()
resetMonitoringVariables() When this function is called, it will update timer and reset the global counter of duplicate acknowledgement segment.
onExpiredTimeoutTimer() When this function is called, its work is to set, ssthesh value will be half of the congestion window. Congestion window values convert to maximum segment size.
public void send(TCPSegment[] segments_, **int** rcvWindow_, **boolean** lostPacket_) When send function of TCP Sender is called, it initializes the segment array with null because it is new transmission so before transmitting segment array should have empty. Otherwise it may contain garbage value which can stand as hindrance on the way of Congestion control. The relevant parameter of congestion control is calculated in send function. The main work of this function is to keep a segment object into the segment array. Before sending Sender checks whether previous transmission has lost or not? This is identified by lost Packet parameter. If lost Packet parameter contains true value, sender can understand that previous segment has been lost and it retransmits that segment. This is the issue of Tahoe. Otherwise, sender sends segment depend on the value of burst size. If burst size is greater than zero only then sender send full size segment. Otherwise sender sends one byte segment for retain connection.

`Class TCP Sender Reno`

This class is used for recover loss. When three duplicate acknowledgement is received by sender only then Reno class is called. It was supposed to transmit another segment after sending one segment if allowed by congestion window and receiver's window but due to simplicity we don't do it. Here only a concept has been given of TCP Reno that it recovers loss.

`Field summary`

public int uad

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

It is the amount of UnACKed data in send window.

public int s

It contains two maximum segment size.

public int m

This variable has been used to calculate maximum value between uad and s

Class TCP Simulator

This is the main class of TCP congestion control. This simulated network consists of the network elements of sender host, router and receiver host. These components are connected in a chain as follows.

SENDER <-> ROUTER <-> RECEIVER

The sender host send only data segment and receiver host generate only acknowledgement segment. In other works, we assume a unidirectional transmission, for the sake of simplicity.This simulator reports the value of congestion control parameters for each transmission by following system.

Iteration number

Congestion window

Effective window

Flight Size

SSThesh

Those are only for Sender side reports and at the end of simulation the sender utilization has been reported.

Field Summary

public static final int *REPORTING_LEVEL_1* = 1<<1;

This field reports when TCP segment loss occurs. It means loss is detected by three or more duplicate acknowledgement or timeout timer expiration. When the sender enters the congestion avoidance sending mode.

public static final int *REPORTING_LEVEL_2* = 1 << 2;

This is the TCP Simulator reporting level 2. It reports every new TCP segment that is created.

public static int *currentReportingLevel* =(*REPORTING_LEVEL_1* | *REPORTING_LEVEL_2*);

This field specifies the current reporting level of this simulator.

public static final int *MSS* = 1024;

This is the masimum segment size in bytes.

public static final int *MAX_WIN* = 100;

This is the maximum window size. In units.

public static final int *SUCCESS* = 0;

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

This is the output of previous transmission.
public static final int *DUP_ACKx3* = *SUCCESS* + 1; This is the outcome of previous transmission that may be three or more duplicate acknowledgement.
public static final int *TIMEOUT* = *DUP_ACKx3* + 1; This is the outcome of previous transmission that indicates timeout.
int actualTotalTransmitted_ It contains how many bytes were transmitted during simulation on run.
int potentialTotalTransmitted_ It contains How many bytes could have been transmitted with the given bottleneck capacity, if there were no losses due to exceeding the bottleneck capacity .
float utilization The sender utilization means actual use of sender.

Constructor Summary

public TCPSimulator(**int** mismatchRatio_, **int** bufferSize_)

This is the constructor for our simulation. Its importance a lot; where sender, router, and receiver have been activated.

Method Summary

public void run(**int** num_iter_) It runs the simulator for the given number of transmission rounds.
public static void main(String[] args) It is the main method for our simulation.

4.4.1 Flowchart of TCP Congestion Control

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

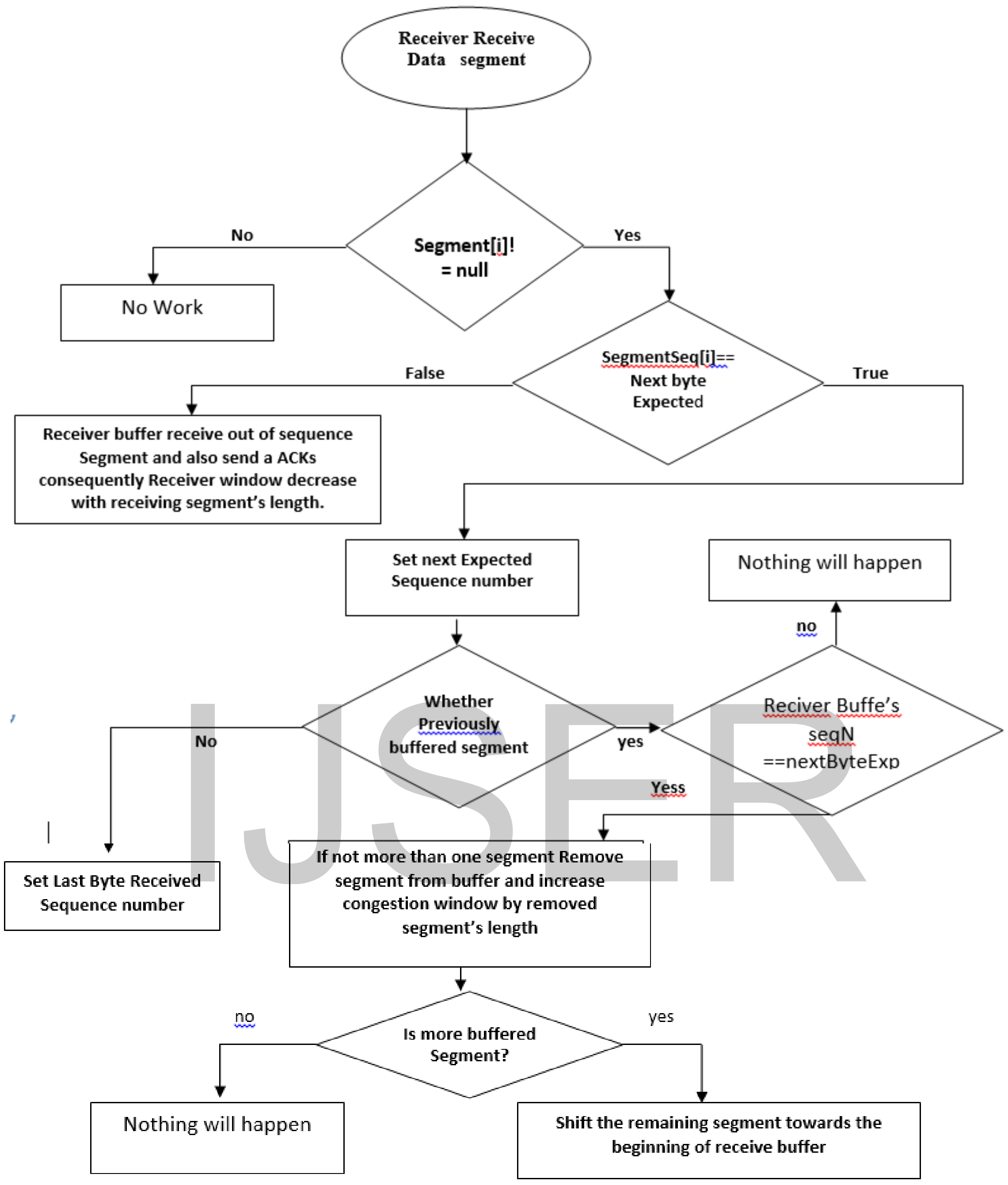

Receiver Receive Data segment
Segment[i]! = null
No
Yes
No Work
SegmentSeq[i]== Next byte Expected
False
True
Receiver buffer receive out of sequence Segment and also send a ACKs consequently Receiver window decrease with receiving segment's length.
Set next Expected Sequence number
Nothing will happen
no
Whether Previously buffered segment
No
yes
Reciver Buffe's seqN ==nextByteExp
Yess
Set Last Byte Received Sequence number
If not more than one segment Remove segment from buffer and increase congestion window by removed segment's length
Is more buffered Segment?
no
yes
Nothing will happen
Shift the remaining segment towards the beginning of receive buffer

Figure 4.1: Procedure of Receiver system

4.4.2 Sender ACK Processing Flowchart

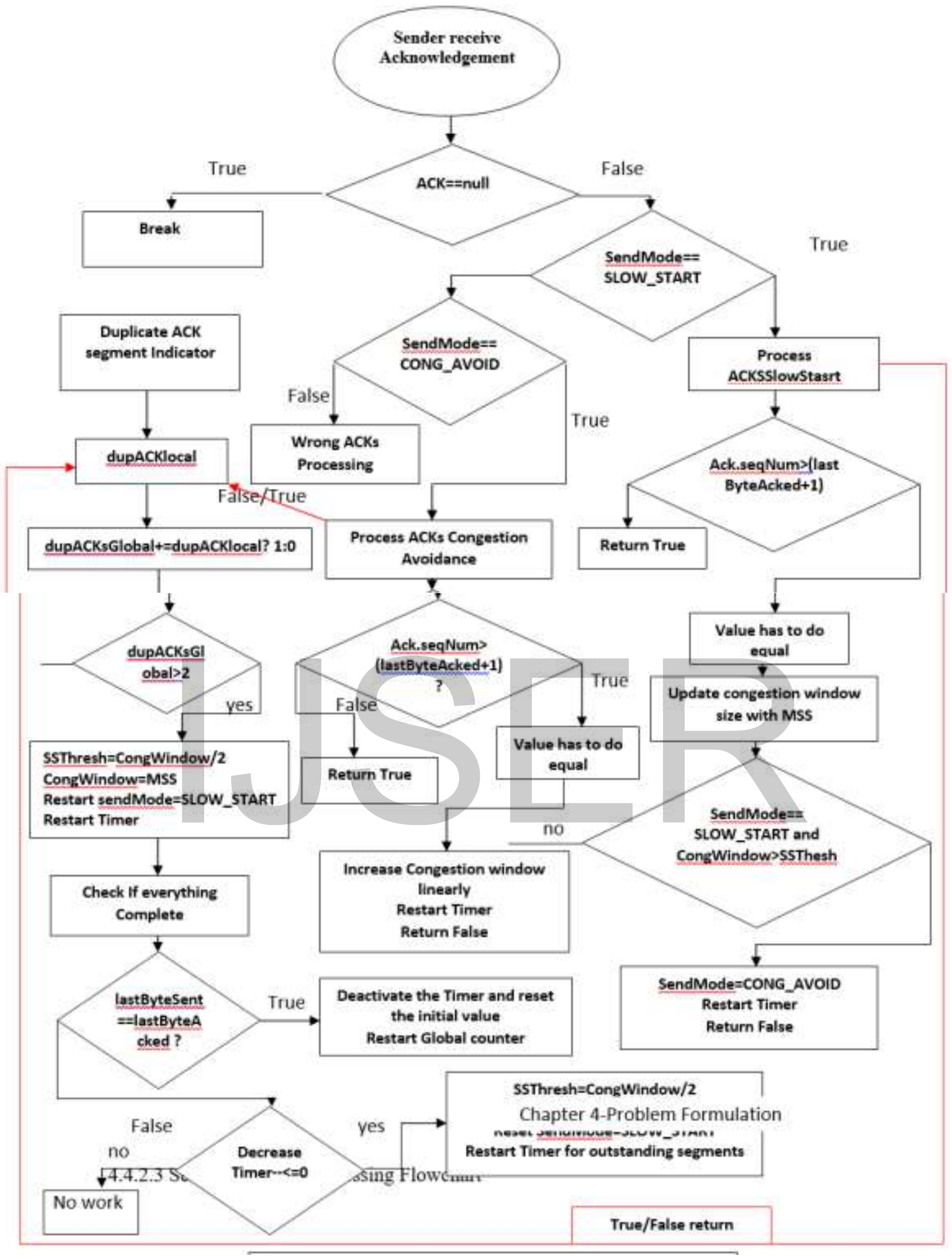

Sender receive Acknowledgement
ACK==null
True
False
Break
SendMode== SLOW_START
True
Duplicate ACK segment Indicator
SendMode== CONG_AVOID
Process ACKSSlowStasrt
False
True
dupACKlocal
Wrong ACKs Processing
Ack.seqNum>(last ByteAcked+1)
False/True
dupACKsGlobal+=dupACKlocal? 1:0
Process ACKs Congestion Avoidance
Return True
Value has to do equal
dupACKsGl obal>2
Ack.seqNum> (lastByteAcked+1) ?
True
Update congestion window size with MSS
yes
False
Value has to do equal
SSThresh=CongWindow/2
CongWindow=MSS
Restart sendMode=SLOW_START
Restart Timer
Return True
no
SendMode== SLOW_START and CongWindow>SSThesh
Check If everything Complete
Increase Congestion window linearly
Restart Timer
Return False
lastByteSent ==lastByteA cked ?
True
Deactivate the Timer and reset the initial value
Restart Global counter
SendMode=CONG_AVOID
Restart Timer
Return False
SSThresh=CongWindow/2
Chapter 4-Problem Formulation
False
yes
no
Decrease Timer--<=0
Restart Timer for outstanding segments
No work
True/False return

Figure 4.2: Sender Acknowledgement Segment

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

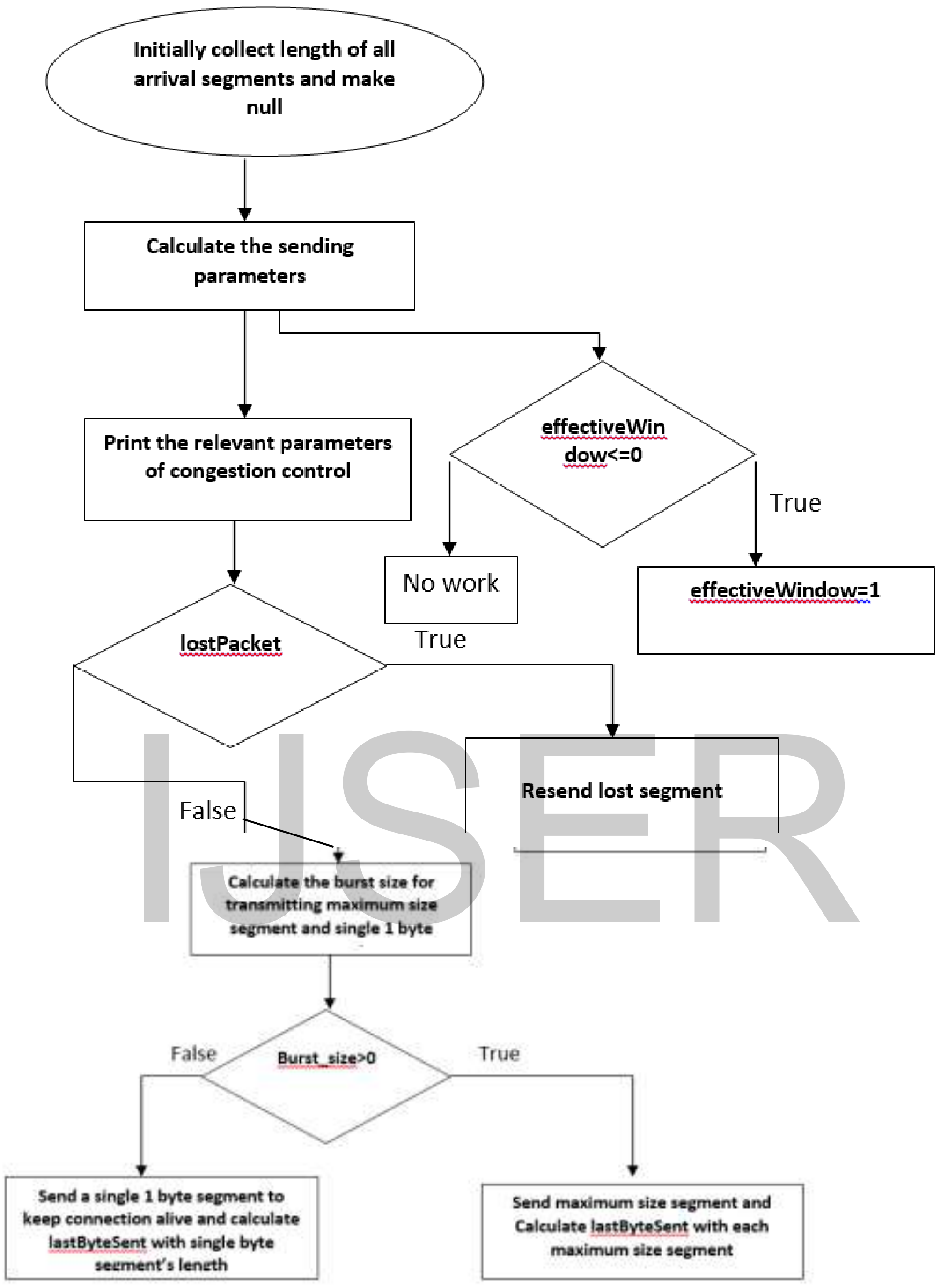

Initially collect length of all arrival segments and make null
Calculate the sending parameters
Print the relevant parameters of congestion control
effectiveWindow<=0
True
No work
effectiveWindow=1
lostPacket
True
False
Resend lost segment
Calculate the burst size for transmitting maximum size segment and single 1 byte
False
Burst_size>0
True
Send a single 1 byte segment to keep connection alive and calculate lastByteSent with single byte segment's length
Send maximum size segment and Calculate lastByteSent with each maximum size segment

Figure 4.3: Sender segment processing System

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

4.4.3 Congestion Control simulation Diagram

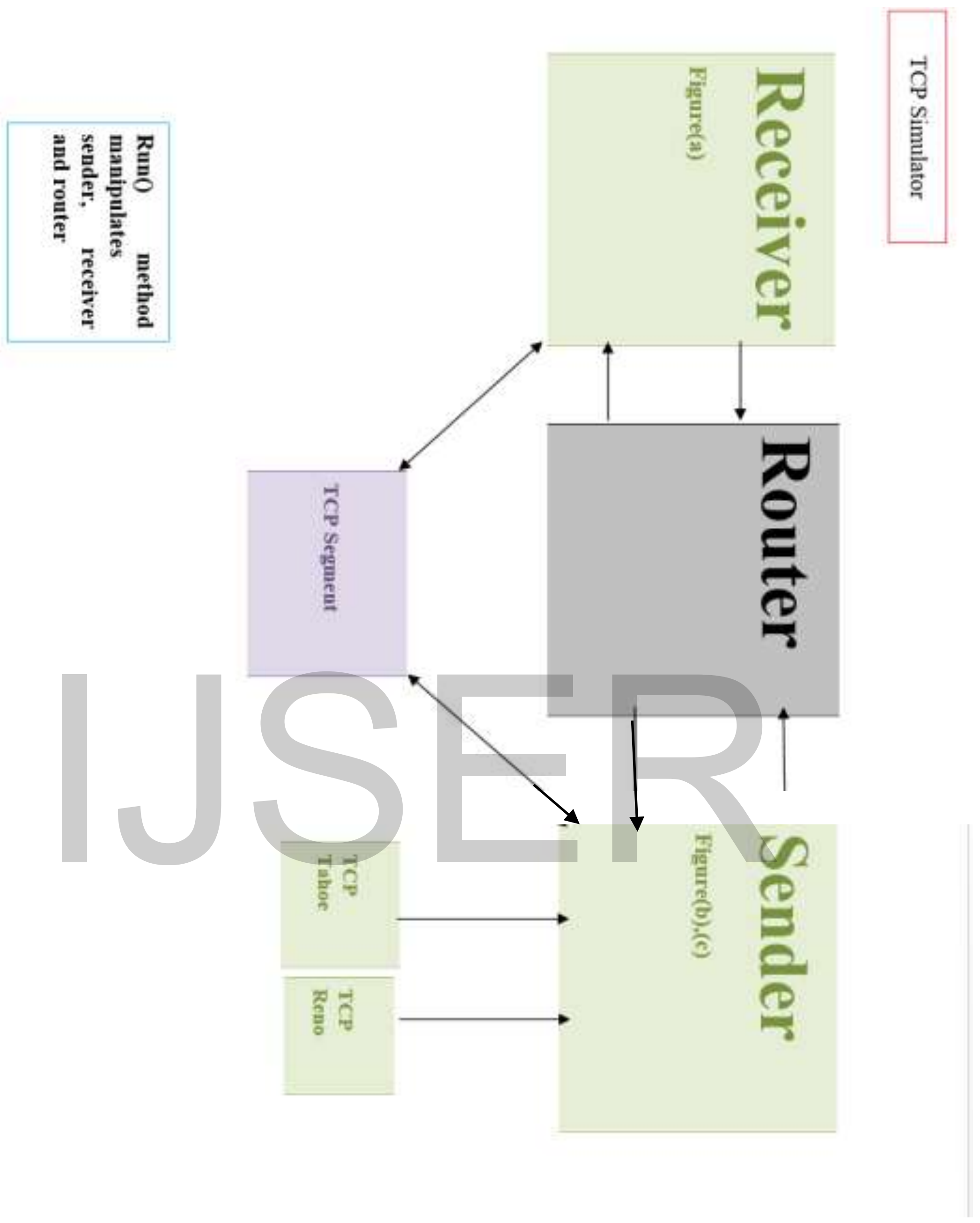


Figure 4.4: Congestion control simulation

4.4.4 Summary of Forward Error Correction simulation

`Class FORWARDERRORCORRECTION`

This class has been used to correct error what occurs during data transmission. There are lots of codes for error correction. We have simulated forward error correction with Hamming (11, 7) code. Hamming code is the single error correction technique. Error occurs when congestion occurs in different network level. Although, there are some constant variants which are used to control congestion, here we have shown only error correction technique.

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

Filed Summary

Int l; It has been used to count index of each bit in packet.
Int n; It has been used to create a number of frames; where data bit and redundancy bit will be put.
Int cwnd=1; It is the congestion window size.
Int flag =0; It has been used to detect error. If any error is occurred anywhere in code, flag will be set to 1 otherwise it remains zero.
Int t_o; It has been used to timeout purposes.
Int ack=1; It is the acknowledgement segment what sender will receive from receiver.
Int I; It has been used to redundant bit calculation.
Int simu_t; It has been used as simulation time.
Int err=0; It will report error.
Int n_out=0; It has been used to generate new output.
Int n_err=0; It has been used to report new error.
FileReader in; It is the file reader building variable type of java. This variable has been used to read data; where in is the data reader type object.

FileWriter out; This java building variable type. Here out is the writer type object.
FileWriter out_pkt; This object has been used to write a packet in file; where out_pkt is a writer type object.
int[] pack = new int[220]; This is the packet array; where all data frame and 20 byte header will be kept.
FileWriter out_ak This is the writer type object. The object out_ak will be used to write a data in file.

```
Constructor Summary
```

Public FORWARDERRORCORRECTION () {}

In this constructor function, three files have been created; where one is reader file others two are writer files. The FileReader variable has been used to read data from a file and two FileWriter variables have been used to write data in file.

```
Method Summary
```

void header() In this header function, first we have created a twenty byte header. Header design depends on user. The first 16 bits has been kept for source port. The other 16 bits has been kept for destination port. The 32 bits has been kept for sequence number. 32 bits has been kept for acknowledgement. The four bits has been kept for TCP header length. The six bits has been kept for unused. Other six bits has been kept for flag set. The 16 bits has been kept for word size. The 16 bits has been kept checksum and other 16 bits has been kept for urgent pointer.
void packet() This function has been used in this program to create a packet. The packet is a array variable. Hamming code has been used to make a packet. In packet array, the five frames which have been made by Hamming (11, 7) code have been kept. Each frame has been made with four redundancy bit and 7 data bit. The redundant bits have been calculated with data bit which is user dependent. In each frame has 11 bits code. The 7 ASCII characters have been taken as input data. Each ASCII character has been converted to binary number and those binary numbers will be kept into data frame. We have created twenty bytes header means 160 bits and 5 frames. In each frame has 11 bits. So number of bits is 215 which have been kept in array packet.
void checksum()

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

The checksum function has been used to create a checksum taking a sender side data bit. Sender sends data including checksum. Receiver also makes a checksum. The 16 bits are kept for checksum purposes that are fixed. A question may be raised by any one that's why we have divided packet length by 16. The answer is very simple. We have applied a technique to count data. We have divided the packet length with 16. After that, the length what we have gotten using that length we have maintained a loop which contains from 1 to divided length and each counting packet index will be increased with 16. So this logic will manipulate whole packet data.
For example: Suppose a packet length is 16000. If we divide it with 16, we will get output 1000. After that if we maintain a loop from 1 to 1000; where another index purposes variable l which starting point 0. Now if variable l increase 16 in each circle, and if variable l increases up to 1000 times, again we will get packet length (1000*16=16000). It means to us that generating each bit of checksum has been calculated considering whole data of packet.

void medium(int u,long t2)
The medium function has been used as media between sender and receiver. Media has also been used to slow transmission. Sender has sent a sequence number and a timer to receiver. When media will receive system timer by sender, media will perform some work. Then media count system timer. These two timer values will be subtracted and compared with given system transmission timer into media. If differencing value of timer is larger than given transmission timer, a timeout or an error will occur. Then congestion window will be changed. If it is an error, congestion window will be half of previous size and flag will set to 1. If there is no error or time out occurs, congestion window will increase twice. These are all about media working system.

int ft_out()

It will return total number of timeout packets.

currentTimeMillis()
It is a building function for java programming language. It will return current time in millisecond from system.

int ft_err()

It will return total number of error packets.

int receive(int u)
Receiver receives number of sequence numbers from sender by receive function. When receive function is called, receive function will generate a checksum from receiving data bits and checksum of sender. Receiver will compare each checksum bit with sender checksum bit. If the checksum of receiver and checksum of sender are equal, receiver can understand that there is no error in transmission that may be for an error or timeout we can find out error by checksum but there is no option to correct it. That's why Hamming code has been used because we can detect an error by hamming code and also correct it. A sequence number is considerate as segment. Segment may lose

for an error or timeout. After those work, receiver will also determine the error and correct the error by hamming code. Receiver has used Hamming (11, 7) code for error detection and correction into the receive function. For each redundancy bit, the number of fixed check bits has been used. For four redundancies, four constant check variables have been used. Each constant check variable will care some particular bits. After completing aforesaid work, the receiver will generate a window system for reporting purposes. It means numbers of logical window views will be generated for each transmission by following way.
public static void main(String[] args) This is a compulsory function of forward error correction.

4.4.5 Forward Error correction

Forward error correction depends on the sender end and receiver end, Where Checksum has been used for checking data both sender and receiver side. A diagram of Checksum is mentioned bellow.

Design of Sender

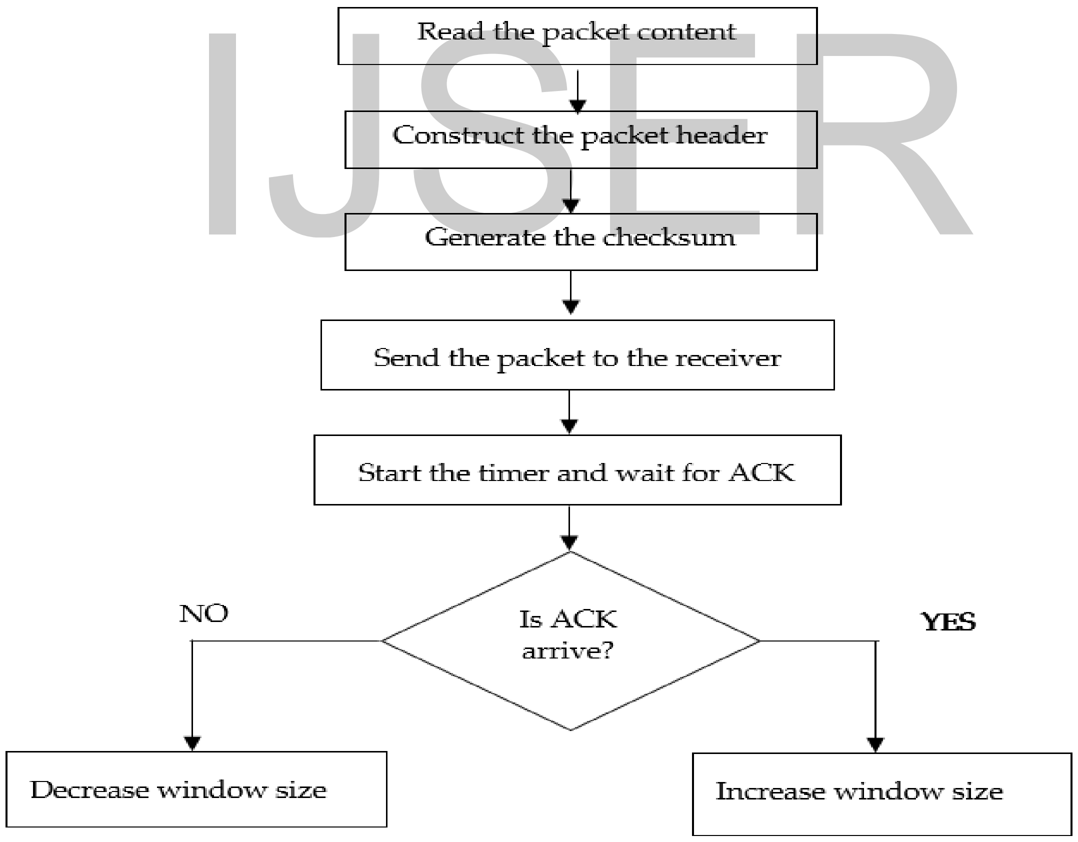


Figure 4.5: Flowchart at the sender end

Steps of sender design

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

- Read the content of packet in binary format from a file
- Construct the packet header
- Generate the checksum
- Send the packet to the receiver
- Start the timer to check the time out
- Update the window size

Design of Receiver:

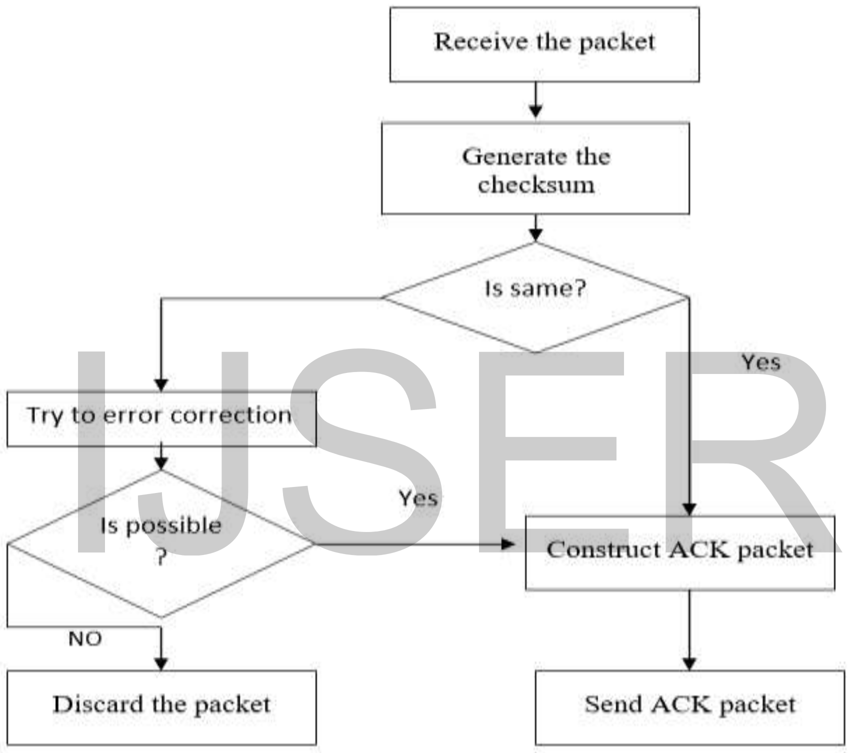


Figure 4.6: Flowchart of processing at the receiver end

Steps of receiver design

- Receive the packet
- Generate the checksum and compare with the existing checksum
- If any error is found then try to correct error
- If error correction is possible or no error occurs then send ack packet
- If error correction is not possible discard the packet and wait for a retransmitted packet.

Checksum and Checksum Generator Diagram

The third error detection method we discuss here is called the checksum. Like the parity checks and CRC, the checksum is based on the concept of redundancy [7]. In the sender, the checksum generator

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

subdivides the data unit into equal segments of n bits(usually) These segments are added using ones complement arithmetic in such a way that the total is also n bits long. That total (sum) is then complemented and appended to the end of the original data unit as redundancy bits, called the checksum field. The extended data unit is transmitted across the network. So if the sum of the data segment is T, the checksum will be –T

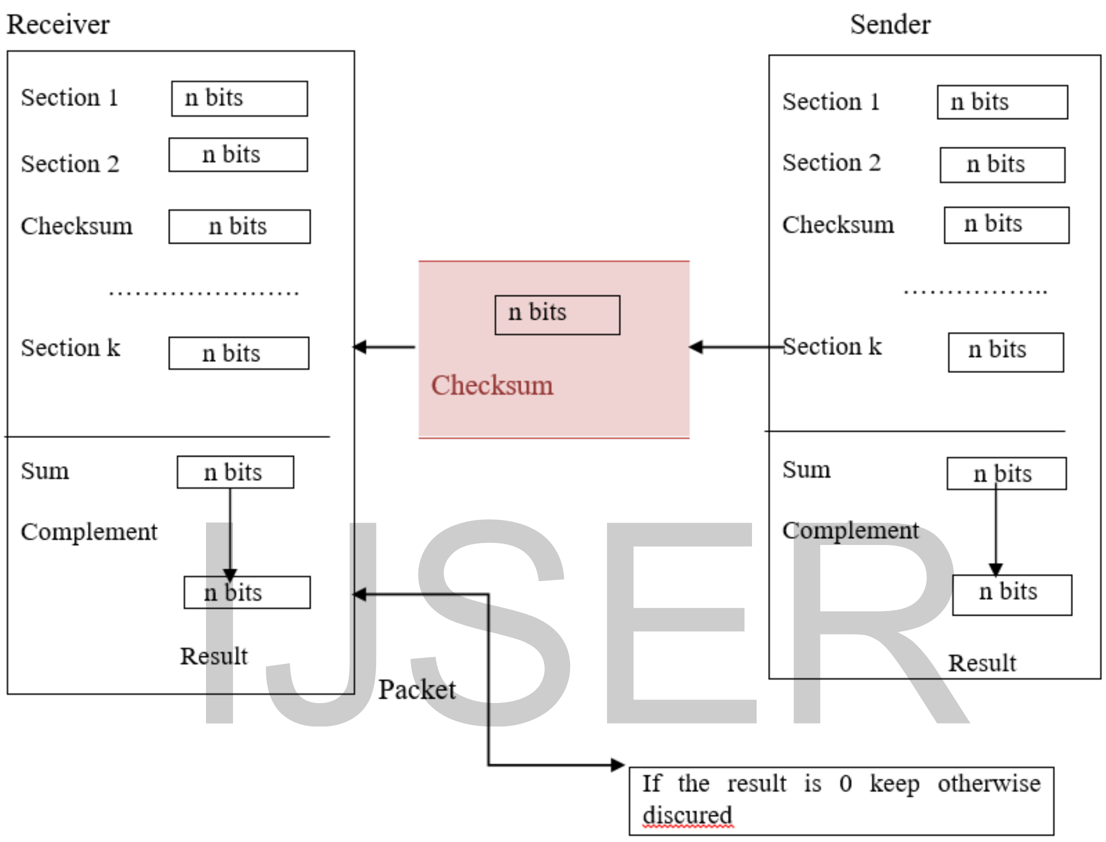


Figure 4.7: Checksum Generator

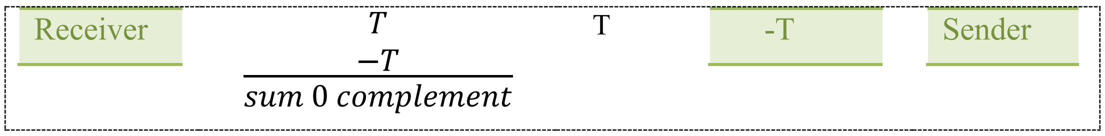


Figure 4.8: Data unit and Checksum

The sender follows these steps

- The unit is divided into k sections, each of n bits.
- All sections are added using ones complement to get the sum
- The sum is complemented and becomes the checksum
- The checksum is sent with the data

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

Checksum Checker

The receiver subdivided the data unit as above and adds all segments and complements the result. If the extended data unit is intact, the total values are found by adding the data.

- The receiver follows these steps:
- The unit is divided into k sections, each of n bits.
- All sections are added using ones complement to get the sum
- The sum is complemented.

If the result is zero, the data are accepted: otherwise, they are rejected

Simulation Model of FEC

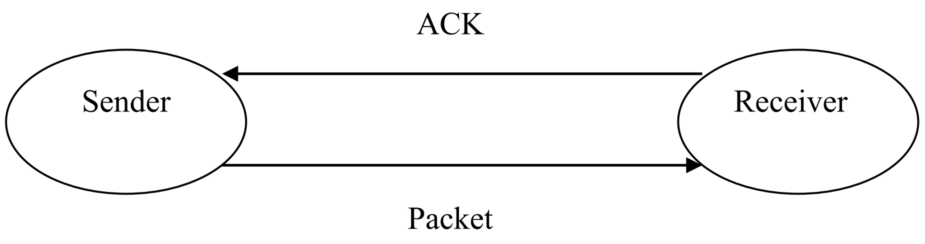


Figure 4.9: Simulation Model of FEC

Here one sender and one receiver is used. The link between the sender and the receiver is wired and bidirectional. The link is used for packet and acknowledgement transmission between the sender and the receiver.

4.4.6 Parameter of FEC

The following parameters are varied for Forward Error Correction (FEC):

- Congestion Window
- Time out
- Retransmission if error correction is not possible
- Sequence no
- Ack processing
- Checksum generation
- Block coding
- Error generation

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

4.5 Simulation Result of TCP Congestion Control

Table 4. 1: The following mentioned result is for sender performance of 5 second simulation on run

Iteration No	CongWindow	EffctWindow	FlightSize	SSThresh
0 Round 0: 1024 ack	1024	1024	0	65535
1 Round 1: 1024 1024 ack ack	2048	2048	0	65535
2 Round 2: 1024 1024 1024 1024 ack ack ack ack	4096	4096	0	65535
3 Round 3: 1024 1024 1024 1024 1024 1024 1024 1024 ack ack ack ack ack ack ack	8192	8192	0	65535

4 Round 4: 1024 1024 1024 1024 1024 1024 1024 1024 1024 1024 1024 1024 1024 1024 ack ack ack ack ack ack ack ack	15360	14336	1024	65535

Sender utilization: 35 %

Table 4.2: The following mentioned result is for sender performance of 10 second simulation run.

Iteration No	CongWindow	EffctWindow	FlightSize	SSThresh
0 Round 0: 1024 ack	1024	1024	0	65535
1 Round 1: 1024 1024 ack ack	2048	2048	0	65535
2 Round 2: 1024 1024 1024 1024 ack ack ack ack	4096	4096	0	65535

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

3 Round 3: 1024 1024 1024 1024 1024 1024 1024 1024 ack ack ack ack ack ack ack	8192	8192	0	65535
4 Round 4: 1024 1024 1024 1024 1024 1024 1024 1024 1024 1024 1024 1024 1024 1024 ack ack ack ack ack ack ack ack	15360	14336	1024	65535
iter No: 5 Number of duplicateACK Received:3 5 Round 5: 1024 Ack	1024	1	15360	7680
6 Round 6: (1-byte) Ack	2048	1	7168	7680
7 Round 7: (1-byte) Ack	2048	1	7169	7680

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

iter = 8 ****************** Timeout occurred!********************************				
8 Round 8: 1024 ack	1024	1	7170	2048
9 Round 9: (1-byte) ack	2048	1	6146	2048

Sender utilization: 29 %

Table 4.3: The following mentioned result is for sender performance of 30 second simulation run.

Iteration No	CongWindow	EffctWindow	FlightSize	SSThresh
0 Round 0: 1024 ack	1024	1024	0	65535
Round 1: 1024 1024 ack ack	2048	2048	0	65535
2 Round 2: 1024 1024 1024 1024 ack ack ack ack	4096	4096	0	65535
3 Round 3: 1024 1024 1024 1024 1024 1024 1024 1024 ack ack ack ack ack ack ack	8192	8192	0	65535
4 Round 4: 1024				

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

1024 1024 1024 1024 1024 1024 1024 1024 1024 1024 1024 1024 1024 ack ack ack ack ack ack ack ack	15360	14336	1024	65535
iter No: 5 Number of duplicateACK Received:3 5 Round 5: 1024 ack	 1024	 1	 15360	 7680
6 Round 6: (1-byte) ack	2048	1	7168	7680
7 Round 7: (1-byte) Ack	2048	1	7169	7680
iter = 8 *******************Timeout occured! ****************************** 8 Round 8: 1024 Ack	 1024	 1	 7170	 2048
9 Round 9: (1-byte) Ack	2048	1	6146	2048
10 Round 10: (1-byte) Ack	2048	1	6147	2048
iter = 11 ********************Timeout occured!**************************** 11 Round 11: 1024 ack	 1024	 1	 6148	 2048
12				

Round 12: (1-byte) ack	2048	1	5124	2048
13 Round 13: (1-byte) ack	2048	1	5125	2048
iter = 14 **********************Timeout occured!************************** 14 Round 14: 1024 ack	1024	1	5126	2048
15 Round 15: (1-byte) Ack	2048	1	4102	2048
16 Round 16: (1-byte) ack	2048	1	4103	2048
iter = 17 **************************Timeout occured!********************** 17 Round 17: 1024 ack	1024	1	4104	2048
18 Round 18: (1-byte) ack	2048	1	2056	2048
19 Round 19: (1-byte) ack	2048	1	2057	2048
iter = 20 *********************Timeout occured!*************************** 20 Round 20: 1024 ack	1024	1	2058	2048
21 Round 21: (1-byte) Ack	2048	1014	1034	2048
22 Round 22: (1-byte) ack	2048	1013	1035	2048
iter = 23 ***********************Timeout occured!************************ 23 Round 23: 1024 ack	1024	1	1036	2048
24				

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

Round 24: 1024 1024 ack ack	2048	2048	0	2048
#################Sender entering congestion avoidance#############				
25	3541	3541	0	2048
Round 25: 1024 1024 1024 ack ack ack				
26 Round 26: 1024 1024 1024 1024 ack ack ack ack	4725	4725	0	2048
27 Round 27: 1024 1024 1024 1024 1024 ack ack ack ack ack	6040	6040	0	2048
28 Round 28: 1024 1024 1024 1024 1024 1024 1024 ack ack ack ack ack ack ack	7472	7472	0	2048

29 Round 29: 1024 1024 1024 1024 1024 1024 1024 1024 1024 ack ack ack ack ack ack ack	9258	9258	0	2048

Sender utilization: 21 %

Table 4.4: The following mentioned result is for sender performance of 50 second simulation run.

Iteration No	CongWindow	EffctWindow	FlightSize	SSThresh
0 Round 0: 1024 ack	1024	1024	0	65535
1 Round 1: 1024 1024 ack ack	2048	2048	0	65535
2 Round 2: 1024 1024 1024 1024 ack ack ack ack	4096	4096	0	65535
3 Round 3: 1024 1024 1024 1024 1024 1024 1024	8192	8192	0	65535

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

1024 ack ack ack ack ack ack ack				
4 Round 4: 1024 1024 1024 1024 1024 1024 1024 1024 1024 1024 1024 1024 1024 1024 ack ack ack ack ack ack ack ack	15360	14336	1024	65535
iter No: 5 Number of duplicate ACK Received:3 5 Round 5: 1024 Ack	1024	1	15360	7680
6 Round 6: (1-byte) ack	2048	1	7168	7680
7 Round 7: (1-byte) ack	2048	1	7169	7680
iter = 8 ************************Timeout occurred!**************************** 8 Round 8: 1024 Ack	1024	1	7170	2048
9 Round 9:	2048	1	6146	2048

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

(1-byte) ack				
10 Round 10: (1-byte) ack	2048	1	6147	2048
iter = 11 ***********************Timeout occurred!************************** 11 Round 11: 1024 Ack	1024	1	6148	2048
12 Round 12: (1-byte) ack	2048	1	5124	2048
13 Round 13: (1-byte) ack	2048	1	5125	2048
iter = 14 **********************Timeout occurred!********************************* 14 Round 14: 1024 Ack	1024	1	5126	2048
15 Round 15: (1-byte) ack	2048	1	4102	2048
16 Round 16: (1-byte) ack	2048	1	4103	2048
iter = 17 **************************Timeout occurred!*********************** 17 Round 17: 1024 Ack	1024	1	4104	2048
18 Round 18: (1-byte) ack	2048	1	2056	2048
19 Round 19: (1-byte) ack	2048	1	2057	2048
iter = 20 **************************Timeout occured!**************************** 20 Round 20: 1024 Ack	1024	1	2058	2048
21 Round 21: (1-byte) ack	2048	1014	1034	2048

22 Round 22: (1-byte) ack	2048	1013	1035	2048
iter = 23 *********************Timeout occurred!**************************** 23 Round 23: 1024 Ack	1024	1	1036	2048
24 Round 24: 1024 1024 ack ack	2048	2048	0	2048
####################### Sender entering congestion avoidance############ 25 Round 25: 1024 1024 1024 ack ack ack	3541	3541	0	2048
26 Round 26: 1024 1024 1024 1024 ack ack ack ack	4725	4725	0	2048
27 Round 27: 1024 1024 1024 1024 1024 ack ack ack ack ack	6040	6040	0	2048
28 Round 28: 1024 1024 1024 1024 1024 1024				

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

1024 ack ack ack ack ack ack ack	7472	7472	0	2048
29 Round 29: 1024 1024 1024 1024 1024 1024 1024 1024 1024 ack ack ack ack ack ack ack	9258	9258	0	2048
30 Round 30: 1024 1024 1024 1024 1024 1024 1024 1024 ack ack ack ack ack ack ack	10889	8841	2048	2048
iter No: 31 Number of duplicate ACK Received:4 31 Round 31: 1024 Ack	1024	1	10240	5444
32 Round 32: (1-byte) ack	2048	1	9216	5444
33 Round 33: (1-byte)	2048	1	9217	5444

ack				
iter = 34 *********************Timeout occurred!**************************** 34 Round 34: 1024 Ack	1024	1	9218	2048
35 Round 35: (1-byte) ack	2048	1022	1026	2048
36 Round 36: (1-byte) ack	2048	1021	1027	2048
iter = 37 ************************Timeout occurred!************************ 37 Round 37: 1024 Ack	1024	1	1028	2048
38 Round 38: 1024 1024 ack ack	2048	2048	0	2048
################### Sender entering congestion avoidance############### 39 Round 39: 1024 1024 1024 ack ack ack	3541	3541	0	2048
40 Round 40: 1024 1024 1024 1024 ack ack ack ack	4725	4725	0	2048
41 Round 41: 1024 1024 1024 1024 1024 ack ack ack	6040	6040	0	2048

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

ack ack				
42 Round 42: 1024 1024 1024 1024 1024 1024 1024 ack ack ack ack ack ack ack	7472	7472	0	2048
43 Round 43: 1024 1024 1024 1024 1024 1024 1024 1024 1024 ack ack ack ack ack ack ack	9258	9258	0	2048
44 Round 44: 1024 1024 1024 1024 1024 1024 1024 1024 ack ack ack ack ack ack ack	10889	8841	2048	2048
iter No: 45 Number of duplicateACK Received:5				

45 Round 45: 1024 Ack	1024	1	10240	5444
46 Round 46: (1-byte) Ack	2048	1	9216	5444
47 Round 47: (1-byte) Ack	2048	1	9217	5444
iter = 48 **********************Timeout occurred!****************************				
48 Round 48: 1024 Ack	1024	1	9218	2048
49 Round 49: (1-byte) ack	2048	1022	1026	2048

Sender utilization: 26 %

Analysis Result

Table1 is the result of 5 second simulation on run. We have gotten five output results in each iteration number. In each output, we have printed iteration number, Congestion window, Effective window, and Flight Size and Slow start threshold values. Among those; for each iteration number, a sender has transmitted number of segments to receiver and also receives number of acknowledgements from receiver. As we have given five input iteration number, we have gotten five outputs pattern from sender. We have shown output result only for Sender performance. In first iteration, sender sends one segment to receiver and receiver also sends an acknowledgement to sender after receiving a segment. In this situation we can see that there is no change in congestion window (1024), effective window (1024), flight size (0) and SSThresh(65536). Because of this, there is no change will occur in first iteration. The value of these four parameter will change depend on number of transmission round. In second iteration, sender sends two segments to receiver and receiver has also received that two arrival segments successfully, then receiver sends two acknowledgement segments to sender. Consequently, Congestion window increases with one maximum segment size (2048). Sender will increase Congestion window after getting an acknowledgement segment. Effective window also increases (2048). Effective window is such window that sender and receiver both can use flexibly. Effective window will increase depend on minimum value between congestion window and receiver window minas flight size. In second iteration, we can see that flight size zero because there is no segment in outstanding. Flight size means some segments have been sent but acknowledgement isn't gotten yet. SSThresh value will be changed depend on timeout or duplicate acknowledgement. In second iteration, since there is no timeout and duplicate acknowledgement, SSThresh value remains unchanged. In third iteration, there is no change. In Forth iteration, we can see that an acknowledgement has been lost. Because, sender has sent eight segment to receiver but receiver has sent 7 acknowledgements to sender. So flight size should have been changed in forth iteration. But simulation has shown it in next iteration that means in fifth iteration, simulation has shown outstanding segment (1024) what is the loss of iteration number 4. In fifth iteration, Sender has sent 14 segments to receiver and receiver has transmitted 8 acknowledgements to sender. Others 6 acknowledgement doesn't go to sender. So, 6 segments have been lost. Due to loss, simulation doesn't show the result of flight size of iteration number 5. Simulation has terminated before reporting of flight size value. Even though, sender doesn't take any initiatives. Because of this, when sender gets three or more duplicate acknowledgment only then sender takes some initiatives to control or to recover loss. On the other hand, sender also takes some initiatives when time out occurs. This is issue of our simulation. Here we can see that simulation doesn't control congestion. Because, simulation run time has terminated before control congestion and recover loss. So, we should give much transmission round or iteration before running simulation. Sender utilization has been shown after completing 5 second simulation run. Sender utilization means actual use of sender or actual performance. Sender utilization will increase depend on its actual transmission. In 5 second simulation run, we have gotten sender utilization 35%. This is the at most utilization in our simulation. The reasons to get such utilization, there is no timeout or duplicate acknowledgement reception. These are all about 5 second simulation run summary.

Table 2 is the result of 10 second simulation on run. We have gotten ten output results with 10 input iteration numbers. First 5 output results are same as Table1's output results. Our objective to understand other 5 output result. In iteration number 6, we have seen that there are three duplicate acknowledgement received by sender. After that congestion window has changed with maximum segment size1024, SSThresh value has been half of the pervious congestion window. Previous congestion window was 15360. Due to duplicate acknowledgement report, SSThresh has been half (7680). Effective window has been 1 because effective window won't be 0 or less. If the value of effective window is equal to zero or less, it will be converted to 1. Flight size has been 15360. Because of this, an ACKs was lost in fourth iteration. In 5th iteration, 6 ACKs was lost. So number of ACKs was lost 7 before 6th iteration. In iteration 6th, sender has shown three duplicate ACKs although there was lost another 8 ACKs that hasn't shown. So total lost segment is 15.The calculation has been made in back end side as (15*1024=15360). Sender has shown the result after counting 6th iteration. That's why we are seeing the flight size as 15360. In iteration 7th, sender hasn't sent any data segment. Because, during iteration 7th, Receiver didn't have enough capacity that's why sender has sent only single 1 byte segment for retaining connection. In iteration 8th, sender also has sent single 1 byte segment. In iteration 9th, a timeout has occurred. Due to timeout, Congestion window's value has been changed with maximum segment size 1024. SSThresh value has also been changed by half of congestion window. So here question may be raised that congestion window was 2048 in that case due to timeout SSThresh value should have supposed to 1024 but without being it has been 2048. Because of this, SSThresh value must be double than congestion window when new transmission has made or timeout occurred or duplicate acknowledgement received by sender. Due to timeout, sender has sent a segment to the receiver and receiver has also sent an acknowledgement after receiving a segment. In transmission round 10th, Sender hasn't sent data segment to receiver because during iteration 10th receiver didn't have capacity to receive a data segment. That's why sender has sent a single 1 byte segment for retaining connection to receiver and receiver has also received it. After that, receiver has sent an acknowledgment to sender to retain connection. We can see that sender has not entered into the congestion avoidance in 10 second simulation run. We should justify this simulation by setting more iteration number before running simulation. So that, we can see that whether simulation can control congestion or no? Sender utilization has been reported after 10 second transmission between sender and receiver. Here we can see that, sender utilization is 29% which is lower than First 5 second simulation run. Because of this, number of Actual transmission between sender and receiver has been lower compare to potential transmission. These are all about 10 second simulation run summary.

Table 3 is the result of 30 second simulation on run. We have gotten thirty output results with 30 input iteration numbers. First 24 outputs are same as Table2. Our objective is to understand another 26 output results. In iteration number 25th, Sender enters into the congestion avoidance. When sender enters into the congestion avoidance, the value of flight size is zero because in this situation no packet is outstanding. When the value of congestion window exceeds the value of SSThresh, only then sender enters into the congestion avoidance. Sender last in congestion avoidance until it again gets three or more duplicate acknowledgement. When it will get again three or more duplicate acknowledgement, sender again starts its activities with slow start mode. In 30 seconds simulation, we get only 21% utilization. Table 4 is the result of 50 second simulation on run. We have gotten fifty output results with 50 iteration number. These results are all about same as Table 3's output result. We have shown these results only for utilization comparison. We have gotten 26% utilization. On the contrary, in 30 second simulation run, we have gotten 21% utilization. We have gotten much utilization in 5th second simulation run because there was no problem.

4.6 Simulation Result of Forward Error Correction

A snapshot of forward error correction result has been taken on receiver side that has been mentioned bellow.

packet's header length is 160

packet length is 215

The Trsamission round up to 5

1s 2s 3s 4s 5s

total no of send packet 455

total no of error packet is 45

total no of time out packet is 3

Table 4.5: The following result is for Forward error correction of receiver side performance

Reciver ACK.doc file	Receiver OUT.doc file	Receiver Info.doc file
For seq no 1 Ack received	Sequence no 1 successfully received Window size 2	Sequence no 1 6096 6096 6096 6096 6096
For seq no 2 Ack received	Sequence no 2 successfully received Window size 4	Sequence no 2 6096 6096 6096 6096 6096
. . . .	. . . .	. . . .
For seq no 9 Ack received	Sequence no 9 successfully received Window size 512	Sequence no 9 6096 6096 6096 6096 6096
For seq no 10 Ack received	Sequence no 10 is lost for an error Window size 256	10 6096 6096 6096 6096 6096
For seq no 11 Ack received	Sequence no 11 is lost for an error Window size 257	11 6096 6096 6096

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

		6096 6096
. . .	. . .	. . .
For seq no 19 Ack received	Sequence no 19 is lost for an error Window size 265	19 6096 6096 6096 6096 6096
For seq no 20 Ack received	Sequence no 20 is lost for an error Window size 132	20 6096 6096 6096 6096 6096
For seq no 21 Ack received	Sequence no 21 is lost for an error Window size 133	21 6096 6096 6096 6096 6096
. . . .	. . . .	. . . .
For seq no 29 Ack received	Sequence no 29 is lost for an error Window size 141	29 6096 6096 6096 6096 6096
For seq no 30 Ack received	Sequence no 30 is lost for an error Window size 70	30 6096 6096 6096 6096 6096
For seq no 31 Ack received	Sequence no 31 is lost for an error Window size 71	31 6096 6096 6096 6096 6096
. . . .	. . . .	. . . .
For seq no 50 Ack received	Sequence no 50 is lost for an error Window size 24	50 6096 6096 6096

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

		6096 5712
For seq no 51 Ack received	Sequence no 51 is lost for an error Window size 25	51 6096 6096 6096 6096 5712
. . . .	. . . .	. . . .
For seq no 74 Ack received	Sequence no 74 is lost for an error Window size 16 error at position 171 error at position 182 error at position 193	74 5904 5904 5900 6096 5712
. . .	. . .	. . .
For seq no 100 Ack received	Sequence no 100 is lost for time out Window size 9 error at position 160 error at position 182 error at position 193	100 5900 5904 5900 6096 5712
For seq no 101 Ack received	Sequence no 101 is lost for an error Window size 10 error at position 160 error at position 171 error at position 193	101 5900 5904 5904 6096 5712
For seq no 102 Ack received	Sequence no 102 is lost for an error Window size 11 error at position 171 error at position 182 error at position 193	102 5904 5904 5900 6096 5712
For seq no 103 Ack received	Sequence no 103 is lost for an error Window size 12 error at position 160 error at position 193	103 5900 5904 5904 6096 5712
For seq no 104 Ack received	Sequence no 104 is lost for an error Window size 13 error at position 160 error at position 182 error at position 193	104 5900 5904 5900 6096 5712
. .	. .	. .

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

. . .	. . .	. . .
For seq no 200 Ack received	Sequence no 200 is lost for time out Window size 7 error at position 160 error at position 182 error at position 204	200 5900 5904 5900 6096 5712
For seq no 201 Ack received	Sequence no 201 is lost for an error Window size 8 error at position 160 error at position 171 error at position 182 error at position 204	201 5900 5904 5900 3024 5712
For seq no 202 Ack received	Sequence no 202 is lost for an error Window size 9 error at position 182 error at position 193 error at position 204	202 5904 5904 5900 2960 5712
For seq no 203 Ack received	Sequence no 203 is lost for an error Window size 10 error at position 171 error at position 182 error at position 204	103 5904 5904 5900 3024 5712
. . .	. . .	. . .
For seq no 284 Ack received	Sequence no 284 is lost for an error Window size 12 error at position 160 error at position 171 error at position 182 error at position 193 error at position 204	284 2764 5808 5900 2960 4176
For seq no 285 Ack received	Sequence no 285 is lost for an error Window size 13 error at position 160 error at position 171 error at position 182 error at position 204	285 2768 5806 5900 3024 4176
For seq no 286 Ack received	Sequence no 286 is lost for an error Window size 14 error at position 171 error at position 182 error at position 193	286 2832 5806 5900 2960 4176

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

	error at position 204	

Analysis Result:

We have gotten aforesaid results by 5 second simulation on run. At first header length has been printed out then packet length. Total number of packets has been sent 455 and error packets 45 and timeout packets are 3 that we can see at beginning level. Then receiver has used three files for reporting purposes. One is ACK.doc file where receiver will generate acknowledgement segment after receiving a segment. One is OUT.doc file where receiver will write a message if receiver can receive a segment successfully or los for timeout or los for any other error. Another is info.doc file; where receiver will generate possible logical window for all fames. Obviously, after detecting and correcting error. That information will be written in aforesaid file by receiver that has been elucidated bellow. In Table Forward error correction result the header length is in fixed format. Packet length is 215 bits including header. After that our simulation transmission round was 5 second what has been printed. Then simulation has show total number of send packet(455) and error packet is 45.After that, the information what has been written in ACK.doc file those have been gotten from receiving function called. Whenever, receive function has been called, receive function has return an ack that value is 1. In file ack.doc, we can see that Receiver has given an ACK for each calling and that information has Witten in ack.doc file including sequence number. Whenever, receive function has been called of receiver, at first receive function has checked out that whether any error or timeout has or not? If there is no error, receive function writes information in OUT.doc file and also if receive function gets any error or timeout packet those information has written in OUT.doc file where window size has increased for each reception. In file OUT.doc, we see that Receiver has received 9 sequence number means 9 packets successfully. But in sequence number 10, an error has reported. Even though, window size is not decrease. But it was supposed to half of previous window size. Without being window size increases. Because of this, compiler still into receive function of receiver. Window size will decrease when again medium function will be called. Medium function will be called after completing receives function of receiver. In sequence no 20, we can see that window size has been half of previous window because medium function has called. After completing the checksum, receiver will detect error and correct it by hamming code. Receiver will detect error position by Hamming (11, 7) code and also write in OUT.doc file. Such as, we can see that sequence no 74 three error position has been detected by Hamming (11, 7) code. In sequence no 103, three error positions has been detected. Although, after detecting error receiver has corrected that error by Hamming (11, 7) code, we can't see this result. No message is printout in OUT.doc file for error correction. In info.doc file, the number of congestion windows size has been printed. The new

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

possible logical Congestion window size has been printed after error detection and correction by hamming (11, 7) code. In info.doc file, we can see that five window has been printed for each sequence number. Because of this, we have taken 5 input frames. 5 fames have been created by Hamming code. In Each frame has 11 bit code; where 7 data bit and 4 redundant bit. A window has made for each frame. In info.doc file, we can see the sequence number 50 where an error occurred at 5th fame that's why window size has decreased because when an error occur congestion window becomes half of previous window which is actually single window. In sequence no 286, we can also that in each fames among 5 frames error has been detected that's why window size of each frame has been decreased. Each window of info.doc file has been made calculating all window of transmission. Such as in initial stage window size was 64KB after that timeout occurs and due to this SSThreh point to 32KB that means congestion size 32KB. In this way, during second timeout congestion window will be 16KB, then 8KB, then 4KB, then 2KB, then 1KB. It is actually difficult to understand but it is easy seeing following graphical view.

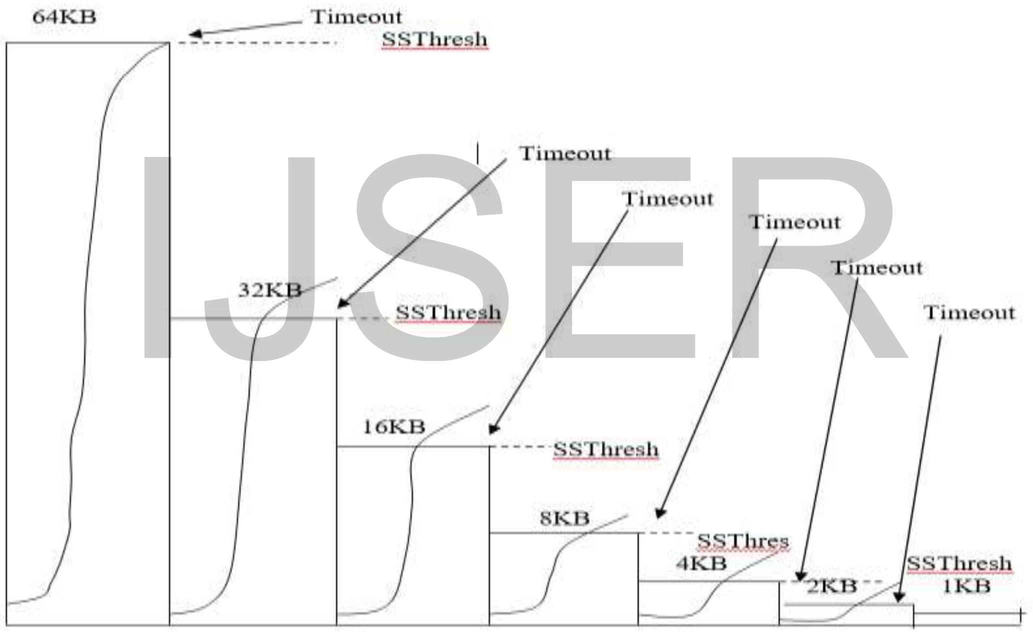


Figure 5.1: Logical Congestion window view

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

We had a topic that how can we control congestion in network level. We have studied about TCP Congestion Control and also have written our simulation study thesis taking Congestion Control concept we have studied five congestion control technique (e.g. Tahoe, Reno, New Reno, SACK, Vegas) and also been able to simulate two techniques especially one is Tahoe TCP and another is Reno TCP. Tahoe TCP takes responsibility of manipulation data at sender level. If any error or timeout occurs during transmission in network level, Tahoe retransmits that segment with restarting timer. If three or more duplicate acknowledgement received by sender, the Reno TCP is called for recovering loss in network level. In result analysis part, we have done analysis taking different simulation result which is for Congestion Control and forward error correction simulation on run. We can't recover any error during transmission in Congestion control simulation. If there was any error generated, those error packets were retransmitted. For this reasons, we have developed an error correction simulation which can detect an error and also can correct error. Although, there are many error correction mechanism what we have also studied but we have simulated only one error correction mechanism due to shortage of time and also for simplicity. We do believe that if anyone want can do but there is no sufficient materials to visualize real world work in our country. We have successfully been able to complete our simulation studies with Java Programming Language. After completing congestion analysis of TCP/IP and its simulation study, we can understand how to control congestion in network level. We also have known how to detect error in network level during transmission and how to correct error in network level. We have known how to analysis simulation for thesis and Research work. We have been confident enough after completing our research work on TCP. Our English language skills and scientific literature understanding skills enhanced to great extent while preparation this documents.

5.2 Author Contribution

I have simulated congestion control system in network level using JAVA Eclipse Framework in order to see how we can control congestion. I have shown that our proposed congestion control system work fine.

5.3 Recommendation for Further Study

We have seen that all congestion control mechanisms are used in sender side and sometimes receiver side. Again if any error occurs what may be single error or burst errors, these errors are recovered with error correction mechanism. But it is a matter of great sorrow that, these error correction mechanisms are used only sender side or receiver side or both sides. It's time consuming and cost effective. For this reasons, client or user may victim or lose patience. So we are proposing new technique what can save clients' time and also get pleasure. If we can develop an algorithm or procedure of congestion control and error correction, that will be bound in each transmitting segment. If each segment including congestion control and error correction algorithm can be sent in network level, that congestion control and error correction algorithm takes care about sending segment. When any error will be generated due to congestion control or error or timeout in network level, our suggested undeveloped algorithm will recover those problems simultaneously in network level. This is our recommendation of proposing thesis and also some idea has been generated by following way……..

- At first comparison between various TCP versions for Two Congestion Control technique especially TCP Tahoe and Reno are implemented, but for wireless handoff is a major issue. To implement handoff more than one base stations is necessary. If handoff is implemented to compare between various TCP versions then their relative performance for wireless link will be clearer.

- Hamming code is used for forward error correction but hamming code can only recover single bit error. Other forward error correction algorithm can also be implemented and their relative performance can be evaluated.

- In this thesis Congestion Control and forward error correction has been implemented using java so all the parameters of wireless network can not implemented here, if forward error correction can be implemented using particular network simulation visual software, then the necessity of Congestion Control and FEC for wireless network will be clearer

REFERENCES

[1] Tanja Lang, "Evaluation of different TCP versions in non-wireline environments", The University of South Australia, Institute for Telecommunications Research, 31 August 2002.

[2] http:// www.pdamd.com/vertical/features/wireless_3.xml (Accessed: July, 2008)

[3] Ling-Jyh Chen, Tony Sun, M. Y. Sanadidi, Mario Gerla, "Improving Wireless Link Throughput via Interleaved FEC", UCLA Computer Science Department, Los Angeles, CA 90095, USA

[4] Henrik Lundqvist and Gunnar Karlsson, "TCP with end-to-end FEC", In Proceedings of International Zürich Seminar on Communications, pages 152 – 155, Zürich, Switzerland, February 2004.

[5] Nayama Islam, Sumyea Helal, "Performance Analysis of TCP Tahoe, Reno, New Reno and SACK over Cellular Mobile System", 8th ICCIT 2005,page 786 to 790, Department of Information and Communication Technology, University of Rajshahi, Rajshahi-6205, Bangladesh

[6] Computer Networks, Andrew S. Tanenbaum 4th edition, Pearson Education Publication.

[7] Data communication and Networking, Behrouz A. Forouzan, 4th edition, Tata McGraw-Hill Publishing Company Limited.

[8]http://www.comp.leeds.ac.uk/roger/HiddenMarkovModels/html_dev/viterbi_algorithm/s1_pg 1.html

[9] V. Jacobson. “Congestion avoidance and control” In Proceedings of ACM SIGCOMM '88, pages 314-329, August 1988. Stanford, CA

[10] M. Allman, V. Paxson, W. Stevens “TCP Congestion Control” RFC 2581 April 1999

[11] http://www.faqs.org/rfcs/rfc2581.html, October-10-2008.

[12] K. Fall and S. Floyd, "Simulation–based Comparisons of Tahoe, Reno, and SACK TCP", Computer Communications Review, 26(3), pp. 5–21, July 1996

[13] S. Floyd. TCP and successive fast retransmits. ftp://ftp.ee.lbl.gov/papers/fastretrans.ps, 1995

[14] M. Mathis, J. Mahdavi, S. Floyd, A. Romanov, “TCP Selective Acknowledgement Options” RFC 2018 October 1996

[15] McDonald, C.S. "Network Simulation Using User–level Context Switching,", Proc. of the Australian UNIX Users' Group Conference '93 , Sydney, Sept 1993, pp1–10

[16] E. L. Yan, X. Yan “Empirical Analyses of SACK TCP Reno and Modified TCP Vegas”, http://citeseer.nj.nec.com/246505.html , November 20, 2008

[17] J. Hoe, "Startup Dynamics of TCP's Congestion Control and Avoidance Schemes", Master's Thesis, MIT, 199, November 21,2008

[18] S. Floyd, T. Henderson, “The NewReno Modification to TCP's Fast Recovery Algorithm” RFC2582 April 1999

[19] Changhee Joo and Saewoong Bahk, "Analysis of Start–up Transition Dynamics of TCP NewReno", Computer Networks, Vol. 36, No. 2, pp. 237–250, 2001

[20] L. Brakmo, S. O'Malley, and L. Peterson, “TCP Vegas: New techniques for congestion detection and avoidance” In Proceedings of the SIGCOMM '94 Symposium (Aug. 1994) pages 24–35

[21] J.–S. Ahn, P. Danzig, Z. Liu, and L. Yan. “An evaluation of TCP Vegas: Emulation and experiment” Computer Communications Review, 25(4):185–195, Oct. 1995

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

[22] L. Brakmo and L. Peterson. “TCP Vegas: End to End Congestion Avoidance on a Global Internet” IEEE Journal on Selected Areas in Communication, Vol 13, No. 8 (October 1995) pages 1465–1480

[23] U. Hengartner, J. Bolliger, and T. Gross. “TCP Vegas revisited” In Proceedings of IEEE Infocom, March 2000

[24] Steven Low, Larry Peterson, and Limin Wang. “Understanding TCP Vegas: theory and practice. Submitted for publication”, Feb. 2000, http://www.ee.mu.oz.au/staff/slow/research/

[25] O. Ait–Hellal, and E. Altman, "Analysis of TCP Vegas and TCP Reno," Proc. IEEE ICC'97, 1997

[26] http://www.iaeng.org/IJCS/issues_v34/issue_1/IJCS_34_1_7.pdf

Appendix A

This section contains simple TCP Sender and TCP Receiver program. We have done this using Java socket programming language. We have not included any error-checking or error handling mechanism. For the shake of simplicity,

```
package receiver.test.expected;
import java.net.*;
import java.io.*;

public class Receiver
{
        ServerSocket ss;
        Socket sc;
        PrintWriter pr;
        BufferedReader br, sout;
        String sendersay;

        public Receiver()
        {
                try {
                        ss = new ServerSocket(2000);
                        sc = ss.accept();
                        br=newBufferedReader(new InputStreamReader(sc.getInputStream()));
                      sout =  new BufferedReader(new InputStreamReader(System.in));
                        pr = new PrintWriter(sc.getOutputStream());
                        System.out.println("THIS IS Receiver");

                         while(true)
                         {
                           String str = br.readLine();
                           System.out.println("Sender >> " + str);
                           pr.flush();
                           System.out.print("Receiver >> ");
```

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

```
                                    sendersay = sout.readLine();
                                    pr.println(sendersay);
                                    pr.flush();
                                   }
                           }
                            catch (IOException ex){System.err.print(ex);}}

        public static void main(String[] args)
        {
                Receiver r= new Receiver();
        }}


package sender.expected;
import java.net.*;
import java.io.*;

public class Sender {
        BufferedReader br, input;
          String cinput;
          PrintWriter pr;

        public Sender()
        {
                try {
                   Socket s = new Socket("127.0.0.1", 2000);
                   br = new BufferedReader(new InputStreamReader(s.getInputStream()));
                   input = new BufferedReader(new InputStreamReader(System.in));
                   pr = new PrintWriter(s.getOutputStream());
                   System.out.println("THIS IS Senderr");
                   while (true)
                   {
                    System.out.print("Sender >> ");
                    cinput = input.readLine();
                    pr.println(cinput);
                    pr.flush();
                    String str = br.readLine();
                    System.out.println("Receiver>> " + str);
                   }

            }
                catch (Exception ex) { }
        }


        public static void main(String[] args)
        {
                Sender sn = new Sender();
        }

}
```

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

Appendix B

This section contains a TCP Congestion Control simulator. We have done this using Java Programming Language; where three components are used. TCP Sender, TCP Router, and TCP Receiver have been used. Although there are five techniques for congestion control, we have used only two techniques especially Tahoe TCP and Reno TCP to control congestion in our simulation for simplicity.

```
package simulator.congestionControl.TCP;

public class Router
{
	private int mismatchRatio;
	private int bufferSize;

	public Router(int mismatchRatio_, int bufferSize_)
	{
		mismatchRatio = mismatchRatio_;
		bufferSize = bufferSize_;

		if (bufferSize >= mismatchRatio)
		{
			bufferSize = mismatchRatio - 1;
		}
	}

	public int getBottleneckCapacity()
	{

		return (bufferSize + 1) * TCPSimulator.MSS;
	}


	public void relay(TCPSegment[] packets_)
	{
		for (int i = bufferSize; i < mismatchRatio; i++)
```

```
            {
                  packets_[i] = null;
            }

            int idx_let_pass_ = 1;

            for (int i = mismatchRatio; i < packets_.length; i++)
            {
                  if ((i % mismatchRatio) != idx_let_pass_)
                  {

                        packets_[i] = null;
                  }
             }
        }
  }
```

```
package simulator.congestionControl.TCP;
public class TCPReceiver{
protected TCPSegment[] rcvBuffer =new TCPSegment[TCPSimulator.MAX_WIN];
protected int lastByteRecvd = -1;
protected int nextByteExpected = 0;
protected int rcvWindow = 65536;
protected int lastBufferedIdx = -1;

public TCPReceiver()
{
      for (int i_ = 0; i_ < rcvBuffer.length; i_++)
      {
            rcvBuffer[i_]= null;
      }
      lastBufferedIdx = -1;
}


public int getRcvWindow()
{
      return rcvWindow;
}


public int receive(TCPSegment[] segments_, TCPSegment[] acks_)
{
   for (int i_ = 0; i_ < acks_.length; i_++)
   {
            acks_[i_] = null;
      }

      for (int i_ = 0; i_ < segments_.length; i_++)
      {
        if (segments_[i_] != null)
          {
        if (segments_[i_].seqNum == nextByteExpected)
          {
```

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

```
            nextByteExpected = segments_[i_].seqNum + segments_[i_].length;

        if(lastBufferedIdx == -1)
         {
          lastByteRecvd =segments_[i_].seqNum + segments_[i_].length - 1;

          }
        else
            {
              checkBufferedSegments();
             }
               acks_[i_] = new TCPSegment(nextByteExpected, 1, true);
             }

            else
               {
                acks_[i_] = outOfSequenceSegment(segments_[i_]);

               }
          }
        }
        return rcvWindow; }


protected TCPSegment outOfSequenceSegment(TCPSegment segment_)
{

      lastBufferedIdx++;
      rcvBuffer[lastBufferedIdx] = segment_;
      lastByteRecvd = segment_.seqNum + segment_.length - 1;
      rcvWindow -= segment_.length;
      int ackSeqNum_ = nextByteExpected;
      return new TCPSegment(ackSeqNum_, 1, true);
}

protected void checkBufferedSegments()
{
  while (lastBufferedIdx >= 0)
        {
   if (rcvBuffer[0].seqNum == nextByteExpected)
              {
      nextByteExpected = rcvBuffer[0].seqNum + rcvBuffer[0].length;
            rcvWindow += rcvBuffer[0].length;
          rcvBuffer[0] = null;
            lastBufferedIdx--;

   if (lastBufferedIdx == -1){break;}

       TCPSegment[] temp_ = new TCPSegment[lastBufferedIdx + 1];
       System.arraycopy(rcvBuffer, 1, temp_, 0, lastBufferedIdx + 1);
       System.arraycopy(temp_, 0, rcvBuffer, 0, lastBufferedIdx + 1);

       rcvBuffer[lastBufferedIdx + 1] = null;
 }
else
     {
      break;
       }
    }
  }
```

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

```
}

package simulator.congestionControl.TCP;

public class TCPSegment {

        public int seqNum = 0;
        public int length = 0;
        public boolean ack;
        public boolean inError = false;

        public TCPSegment(int seqNum_, int length_)
        {
                this(seqNum_, length_, false);
        }

        public TCPSegment(int seqNum_, int length_, boolean ack_)
        {
                this.seqNum = seqNum_;
                this.length = length_;
                this.ack = ack_;



if((TCPSimulator.currentReportingLevel&TCPSimulator.REPORTING_LEVEL_2)!= 0)
                {

System.out.println((ack?"\tack":((length==1)?"(1-byte)":"\t"+Integer.
toString(length))));
                }
        }
}

package simulator.congestionControl.TCP;

public abstract class TCPSender {
        protected int lastByteSent = -1;
        protected int lastByteAcked = -1;
        protected int congWindow = TCPSimulator.MSS;
        protected int SSThresh = 65535;
        protected static final int SLOW_START = 0;
        protected static final int CONG_AVOID = 1;
        protected int sendMode = SLOW_START;
        protected static final int TIMER_DEFAULT = 3;
        protected int timer = TIMER_DEFAULT;
        protected int dupACKsGlobal = 0;


        public int getTotalBytesTransmitted()
        {
                        return (lastByteAcked + 1);
        }

        public abstract int processAcks(TCPSegment[] acks_);

        public abstract void send(TCPSegment[] segments_, int rcvWindow_,
        boolean lostPacket_);


}
```

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

```
package simulator.congestionControl.TCP;

public class TCPSenderTahoe extends TCPSender
{
        int f=0;
       public int processAcks(TCPSegment[] acks_)
       {

              int retVal_ = TCPSimulator.SUCCESS;

              for (int i_ = 0; i_ < acks_.length; i_++)
              {

                     if (acks_[i_] == null)
                     {
                            break;
                     }


                     boolean dupACKlocal_ = false;

                     if (sendMode == SLOW_START)
                     {
                            dupACKlocal_ = processAcksSlowStart(acks_[i_]);

                     }
                     else if (sendMode == CONG_AVOID)
                     {

                     dupACKlocal_ = processAcksCongestionAvoidance(acks_[i_]);

                     }

                     else
                       {
System.out.println("TCPSenderTahoe.processAcks(): Wrong sending mode.");
                 }

                            dupACKsGlobal += dupACKlocal_ ? 1 : 0;

                            if (dupACKsGlobal > 2)
                            {
                              onThreeDuplicateACKs();
                               retVal_ = TCPSimulator.DUP_ACKx3;
                                break;
                            }
                 }


              if (lastByteSent == lastByteAcked)
              {
                     resetMonitoringVariables();

              }

              else
                  {
                            timer--;
```

```
                        if (timer <=0)
                        {
                        onExpiredTimeoutTimer();

                        retVal_ = TCPSimulator.TIMEOUT;
                      }
                 }

            return retVal_;
      }

  protected boolean processAcksSlowStart(TCPSegment ack_)
            {

            if (ack_.seqNum > (lastByteAcked + 1))
            {
                  lastByteAcked = ack_.seqNum - 1;

                  congWindow += TCPSimulator.MSS;

                  if ((sendMode == SLOW_START)&&(congWindow > SSThresh))
                    {
                        sendMode = CONG_AVOID;

                        if ((TCPSimulator.currentReportingLevel &
TCPSimulator.REPORTING_LEVEL_1) != 0)
                        {
System.out.println("\##nSender entering congestion avoidance ##"+"\n");
                        }
                    }
                  resetMonitoringVariables();

                      return false;

            }

            else
               {
                      return true;
              }

      }

  protected boolean processAcksCongestionAvoidance(TCPSegment ack_)
  {
            if (ack_.seqNum > (lastByteAcked + 1))
            {
                  lastByteAcked = ack_.seqNum - 1;
                  congWindow +=(TCPSimulator.MSS * TCPSimulator.MSS) /
congWindow + TCPSimulator.MSS / 8;
                  resetMonitoringVariables();

                  return false;}

            else
            {
                        return true;
                }
   }
  protected void onThreeDuplicateACKs()
  {
            if (dupACKsGlobal > 2)
            {
            SSThresh = congWindow / 2;
                  SSThresh = Math.max(SSThresh, 2*TCPSimulator.MSS);

                  congWindow = TCPSimulator.MSS;
                  sendMode = SLOW_START;

                  resetMonitoringVariables();
            }
      }

  protected void onExpiredTimeoutTimer()
      {
            if (timer <= 0)
            {
                  SSThresh = congWindow / 2;
                  SSThresh = Math.max(SSThresh, 2*TCPSimulator.MSS);


                  congWindow = TCPSimulator.MSS;

                  sendMode = SLOW_START;

                  resetMonitoringVariables();
            }
      }

  protected void resetMonitoringVariables()
  {
            dupACKsGlobal = 0;
            timer = TIMER_DEFAULT;
  }

  public void send(TCPSegment[] segments_, int rcvWindow_, boolean
lostPacket_)
  {
            for (int i_ = 0; i_ < segments_.length; i_++)
            {
                  segments_[i_] = null;
            }

          int flightSize_ = lastByteSent - lastByteAcked;
          f=flightSize_;
            int effectiveWindow_ =Math.min(congWindow, rcvWindow_) -
flightSize_;

            if (effectiveWindow_ <= 0)
            {
                  effectiveWindow_ = 1;
            }

System.out.println("\t\t" +congWindow + "\t\t\t  " + effectiveWindow_ +
      "\t\t\t" + flightSize_ + "\t\t\t" + SSThresh);


            System.out.println("\nRound " +TCPSimulator.r+":");
            if (lostPacket_)
            {
                  segments_[0] =new TCPSegment(lastByteAcked + 1,
TCPSimulator.MSS);
                  return;
            }

            int burst_size_ = effectiveWindow_ / TCPSimulator.MSS;

            if (burst_size_ > 0)
            {
                  for (int seg_ = 0; seg_ < burst_size_; seg_++)
                  {
                        segments_[seg_] = new TCPSegment(lastByteSent + 1,
TCPSimulator.MSS);
                        lastByteSent += segments_[seg_].length;
                  }

            }

            else
              {

             segments_[0] = new TCPSegment(lastByteSent + 1, 1);
               lastByteSent += segments_[0].length;
            }}}

package simulator.congestionControl.TCP;

public class TCPSenderReno extends TCPSenderTahoe
{

      public int uad;
      public int s,m;

public void Recover(TCPSegment[] segments_, int rcvWindow_,     boolean
lostPacket_)
 {
      uad=f/2;
      s=2*TCPSimulator.MSS;
      m=Math.max(uad,s);
      if(SSThresh<=m)
      {
            if(lostPacket_)
segments_[0]=new
TCPSegment(lastByteAcked+1,TCPSimulator.MSS);
            congWindow=congWindow+3*TCPSimulator.MSS;

      }}}

package simulator.congestionControl.TCP;

public class TCPSimulator {

public static final int REPORTING_LEVEL_1 = 1<<1;
public static final int REPORTING_LEVEL_2 = 1 << 2;
public static int currentReportingLevel =(REPORTING_LEVEL_1   |
REPORTING_LEVEL_2);

      public static final int MSS = 1024;
```

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

```
      public static final int MAX_WIN = 100;
      public static final int SUCCESS = 0;
      public static final int DUP_ACKx3 = SUCCESS + 1;
      public static final int TIMEOUT = DUP_ACKx3 + 1;
      private TCPSender sender=null;
      private TCPSenderReno sender1=null;
      private TCPReceiver receiver = null;
      private Router router = null;
      private int c=2;
     public static int r=0;

      public TCPSimulator(int mismatchRatio_, int bufferSize_)
      {
            sender1=new TCPSenderReno();
            sender = new TCPSenderTahoe();
            receiver = new TCPReceiver();
            router = new Router(mismatchRatio_, bufferSize_);
      }

      public void run(int num_iter_)
      {
            TCPSegment[] segments_ = new TCPSegment[MAX_WIN];
            TCPSegment[] acks_ = new TCPSegment[MAX_WIN];

            for (int i_ = 0; i_ < MAX_WIN; i_++)
            {
                  segments_[i_] = null;
                  acks_[i_] = null;
            }

System.out.println("Iter\t\t\t CongWindow\t\t"
+"EffctWindow\t\tFlightSize\t\tSSThresh");
System.out.println("=========================================");
            int rcvWindow = receiver.getRcvWindow();

            for (int i_ = 1; i_ <= num_iter_; i_++)
            {
                  int outcome_ = SUCCESS;
                  if (i_ != 1)
                  {
                        outcome_ = sender.processAcks(acks_);
                  }
                  if((outcome_ == DUP_ACKx3) &&((currentReportingLevel  &
REPORTING_LEVEL_1) != 0))
                  {
                c++;

System.out.println("iter No: " + (i_-1)+ " "+"Number of duplicateACK
Received:"+c);


      sender1.Recover(segments_, rcvWindow, outcome_ != SUCCESS);

                  }
                  else if((outcome_ == TIMEOUT)&&((currentReportingLevel  &
REPORTING_LEVEL_1) != 0))
                  {
System.out.println("iter = " + (i_-1) + "** Timeout occured! ");
                  }
```

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

```
                    System.out.print((i_-1) + "\t");

                    sender.send(segments_, rcvWindow, outcome_ != SUCCESS);
                    router.relay(segments_);
                    rcvWindow = receiver.receive(segments_, acks_);
                    System.out.print("\n");
                    r++;

            }

System.out.println=========================================");
            int actualTotalTransmitted_ =
sender.getTotalBytesTransmitted();
            int potentialTotalTransmitted_ =router.getBottleneckCapacity()
* num_iter_;

            float utilization_ =(float) actualTotalTransmitted_ / (float)
potentialTotalTransmitted_;

            System.out.println("Sender utilization: " +
Math.round(utilization_*100.0f) + " %");
      }

      public static void main(String[] args)
      {
            int x=args.length;
            if(x< 1)
                  x=100;
            if (x< 1) {

System.err.println("Please enter the number of iterations!");
                  System.exit(1);
            }

            int mismatch_ratio_ = 10;
            int buffer_size_ = 7;

            TCPSimulator simulator =
                  new TCPSimulator(mismatch_ratio_, buffer_size_);

            Integer numIter_ = new Integer(x);

            simulator.run(numIter_.intValue());


      }

}
```

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

Appendix C

This section contains simple TCP Sender and TCP Receiver program. We have done this using Java Programming Language; Where Forward Error Correction has been done. There are many Error Correction mechanisms. But, we have included only Hamming code [H (11, 7)] for Forward Error correction in our simulation study.

```
package forward.error.correction.test;
import java.io.*;
import java.util.Random;

public class FORWARDERRORCORRECTION
{
     int l,cwnd=1,n=5,flag=0,t_o=0;
     int ack=1,i=0;
     int err=0;
     int[] pack = new int[220];
     int n_out=0,n_err=0;
     FileReader in ;
     FileWriter out,out_pkt;


      public FORWARDERRORCORRECTION()
      {
         try
            {
    in = new FileReader("E:\\Thesis\\code\\CHAR_OUT.txt");
    out= new FileWriter("E:\\Thesis\\code\\OUT.doc");
    out_pkt=new FileWriter("E:\\Thesis\\code\\info.doc");
                }
  catch(java.io.FileNotFoundException ss)
          {}
  catch(java.io.IOException ss1)
                      {}
      }
```

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

```
  void header()
    {
              for( l=0;l<16;l++)        pack[l]=0;  //for sp[16]
              for( l=16;l<32;l++)       pack[l]=1;  //for dp[16]
              for( l=32;l<64;l++)       pack[l]=0;  //for sn[32]
              for( l=64;l<96;l++)       pack[l]=1;  //for an[32]
              for( l=96;l<100;l++)      pack[l]=0;  //tcp_h_l[4]
              for( l=100;l<106;l++)     pack[l]=1;   //un_use[6]
              for( l=106;l<112;l++)     pack[l]=0;  //for b6[6]
              for( l=112;l<128;l++)     pack[l]=1;  //for ws[16]
           for( l=128;l<144;l++)        pack[l]=0;//check_sum[16]
           for( l=144;l<160;l++)        pack[l]=1;  // for up[16]
      }
void packet()
    {
      System.out.println("\npacket's header length is "+l);

         int j=0; int[]d=new int[8];

         String input="abcdefg";
     try
         {

           for(int ig=1;ig<=n;ig++)
           {
            char c[] = new char[(char)input.length()];
             in.read(c);


             for(j=0;j<7;j++)
             d[j]=c[j]-48;    //finding the data

             for( j=1;j<12;j++)
              {
                i=0;
                if(j==1)
                 {
                   i=d[0]+d[1]+d[3]+d[4]+d[6];
                   if(i%2==0)
                   pack[l+j-1]=0;
                   else pack[l+j-1]=1;
                 }
               else if(j==2)
                  {
                    i=d[0]+d[2]+d[3]+d[5]+d[6];
                    if(i%2==0)
                    pack[l+j-1]=0;
                    else pack[l+j-1]=1;
                  }
               else if(j==3)
                 {
                  pack[l+j-1]=d[0];
                 }
               else if(j==4)
                {
                      i=d[1]+d[2]+d[3];
                      if(i%2==0)
                      pack[l+j-1]=0;
                      else
                      pack[l+j-1]=1;
```

Publication Partner:
International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

```
                   }

           else if(j==5)
             {
               pack[l+j-1]=d[1];
             }

          else if(j==6)
            {
              pack[l+j-1]=d[2];
            }
          else if(j==7)
            {
              pack[l+j-1]=d[3];
            }
         else if(j==8)
          {
          i=d[4]+d[5]+d[6];
          if(i%2==0)
          pack[l+j-1]=0;
          else pack[l+j-1]=1;
         }

          else if(j==9)
            {
             pack[l+j-1]=d[4];
          }
          else if(j==10)
            {
              pack[l+j-1]=d[5];
            }
         else if(j==11)
          {
             pack[l+j-1]=d[6];
          }
      }
          l=l+11;
    }
     in.close();
    }

    catch(java.io.IOException e)
    {
          System.out.println("Cannot access the input file");
    }
    System.out.println();
    System.out.println("packet length is "+l);

      System.out.print("\nThe Trsamission round up to 5\n\n");
}


void checksum()
    { for(int u=0;u<16;u++)
        {
            int c=0,l=u;
            for( int v=1;v<=pack.length/16;v++)
            {
                  if(l>126 && l<144)
                        c=c+0;
```

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

```
                    else
                      c=c+pack[l];
                          l=l+16;
              }

              pack[u+127]=c%2;
          }
      }


void medium(int u,long t2)
      {
        int rtt=10;        //set the round trip time here
        try{Thread.sleep(rtt);
      }

      catch(java.lang.InterruptedException gg)
      {   }

Random r=new Random();
      err=0;

   if (u%10==0)
   {
        err=r.nextInt(215);
        if(pack[err]==0)
        pack[err]=1;
        else pack[err]=0;
        err=1;
        n_err=n_err+1;
  }

       t_o=0;
       if(u%100==0)
         try
         {
           Thread.sleep(rtt);
         }
         catch(java.lang.InterruptedException gg)
         {   }

       long t3 = System.currentTimeMillis();
       if(t3-t2>=20)
       {
         err =1;     t_o=1;       }
         if (cwnd==0)
          cwnd=1;
         if (err==1)
         {
           cwnd=cwnd/2;
            flag=1;
          }

else if(err==0 && flag==0)
      cwnd=cwnd*2;

else if(err==0 && flag==1)
      cwnd=cwnd+1;
}
int receive(int u)
```

```
    {
      int[] check = new int[20];
      int err=0;
      for(int b=0;b<16;b++)
         {
           int c=0,l=b;
            for( int v=1;v<=pack.length/16;v++)
            {
              if(l>126&&l<144)
               c=c+0;

              else
                  c=c+pack[l];
                  l=l+16;
            }
              check[b]=c%2;}


            for(int b=0;b<16;b++)
            if(check[b]!=pack[b+127])
            err=1;
            if (err==1 && t_o==1)
                {
                  try
                    {
out.write("\n\nSequence no "+u+" is lost for time out"+"\nWindow size
"+cwnd);
                    }
                    catch(java.io.IOException ss1)
                    {}
                    n_out=n_out+1;
                  }
             else if (err==1)
                      {
                        try
                           {
out.write("\n\nSequence no "+u+" is lost for an error"+"\nWindow size
"+cwnd);
                           }
                           catch(java.io.IOException ss1)
                           {}
                         }
                      else
                      {
                        try
                           {
out.write("\n\nSequence no "+u+" successfully received"+"\nWindow size
"+cwnd);
                           }
                           catch(java.io.IOException ss3)
                           {}
                  }
              err=0;
            try
                {
                  out_pkt.write("\n"+u);
                }
                catch(java.io.IOException ss)
                {}
```

```
                    int l1=160;
      int []e = new int[5];
      for(int j1=1;j1<=n;j1++)
       {
         int
e1=((pack[l1+0])+(pack[l1+2])+(pack[l1+4])+(pack[l1+6])+(pack[l1+8])+(pack[
l1+10]))%2;
         int
e2=((pack[l1+1])+(pack[l1+2])+(pack[l1+5])+(pack[l1+6])+(pack[l1+10])+(pack
[l1+9]))%2;
         int e3=((pack[l1+3])+(pack[l1+4])+(pack[l1+5])+(pack[l1+6]))%2;
         int e4=((pack[l1+7])+(pack[l1+8])+(pack[l1+9])+(pack[l1+10]))%2;
         e[1]=e1;e[2]=e2;e[3]=e3;e[4]=e4;
          int b=0,y;


if(e1==0&&e2==0&&e3==0&&e4==0)
   b=0;
else
 {
   b=e4*8+e3*4+e2*2+e1*1;
if(pack[l1+b-1]==0)
     pack[l1+b-1]=1;
      else pack[l1+b-1]=0;
         try
         {
           out.write("\nerror at position "+l1);
         }
         catch(java.io.IOException ss1)
         {}
       }
  int  b1=((pack[l1+2]))*64;
       b1=b1+(pack[l1+4])*32;
       b1=b1+(pack[l1+5])*16;
       b1=b1+(pack[l1+6])*8;
       b1=b1+(pack[l1+8])*4;
       b1=b1+(pack[l1+9])*2;
       b1=b1+(pack[l1+10])*1;

  try
       {
          if(b1<0)
      b1=b1*-1;
         out_pkt.write("\n"+b1);
       }
     catch(java.io.IOException ss)
       {}
       l1=l1+11;
    }
     ack=1;
   return (ack);
 }

int ft_out()
 {
   return n_out;
 }
int ft_err()
 {
   return n_err;
 }
      public static void main(String[] args)
            {
              try
           {

          FileWriter out_ak = null;
          try {
            out_ak = new FileWriter("E:\\Thesis\\code\\Ack.doc");
          }
          catch (IOException ex) {
          }


          int simu_t = 5; //set the simulation time here
           int seq, s = 1;

FORWARDERRORCORRECTION ob = new FORWARDERRORCORRECTION();
           ob.header();
           ob.packet();

           ob.checksum();
 long time = System.currentTimeMillis(),time1,t1,t2;//t3;
           t1 = time;
           System.out.print(s + "s  ");
           s = 2;

           for (seq = 1; ; seq++) {
             try {
               t2 = System.currentTimeMillis();
               ob.medium(seq, t2);

               if (ob.receive(seq) == 1)
                 try {
                   out_ak.write("\nFor seq no " + seq + " Ack received");
                 }
                 catch (IOException ex1) {
                 }

               else
                 try {
                   out_ak.write("\nFor seq no " + seq +
                                " Ack not received");
                 }
                 catch (IOException ex2) {
                 }

             }

             catch (java.lang.NullPointerException xx) {
               System.out.println("EXception");
             }

             time1 = System.currentTimeMillis();
             if (time1 - time >= (simu_t + 0) * 1000)
               break;

             if (t1 + 1000 == time1) {
               System.out.print(s + "s  ");
               t1 = time1;
```

Publication Partner:

International Journal of Scientific & Engineering Research-IJSER (ISSN: 2229-5518)

```
                s = s + 1;
              }
            }

            System.out.println("\n\ntotal no of send packet " + seq);
            System.out.println("\ntotal no of error packet is " +
ob.ft_err());
            System.out.println("\ntotal no of time out packet is " +
ob.ft_out());
          }

          catch (java.lang.NullPointerException fxgf)
           {System.out.print("Exception");}
           System.out.print("\nThe End of the simulation");
        }

}
```