TCP performance and the effect of queues in routes

Master of Technology
in
Computer Science

By
DEVENDRA KUMAR

School of Computer and Information Sciences
University of Hyderabad, Gachibowli
Hyderabad - 500046, India

Month, Year

March 28, 2015

Abstract. In this paper, we understand TCP performance and the effect of queues in routes. Reliable transport protocol such as TCP are tuned to perform well in traditional networks where packet losses accur mostly because of congestion. Here we are discuss on how congestion behave with queue size, number of queue and different type of packet in different queues(small and big packet behaviour with respect to queues). We also discuss the one of the famous congestion control protocol reno in this paper.

1 Introduction

1.1 Mininet - Network topology emulator

Mininet is a network emulator. It runs a collection of end-hosts, switches, routers, and links on a single Linux kernel. A Mininet host behaves just like a real machine; you can ssh into it (if you start up sshd and bridge the network to your host) and run arbitrary programs. The programs you run can send packets through what seems like a real Ethernet interface, with a given link speed and delay. Packets get processed by what looks like a real Ethernet switch, router, or middlebox, with a given amount of queuing.

Why MININET?

1. **Custom topologies:** a single switch, huge Internet-like topologies, the Stanford backbone, a data center, or anything else can be create.
2. **Can run real programs:** anything that runs on Linux is available to run, from web servers, to TCP window monitoring tools, to Wireshark.
3. **Customize packet forwarding:** Mininet switches are programmable using the OpenFlow protocol.
4. **Can share and replicate results:** anyone with a computer can run any other code once they packaged it up.
5. **You can use it easily:** you can create and run Mininet experiments by writing simple (or complex if necessary) Python scripts.

Compared to simulators, Mininet runs real, unmodified code including application code, OS kernel code, and control plane code (both OpenFlow controller code and Open vSwitch code) and easily connects to real networks.

Limitations of mininet: Mininet-based networks cannot (currently) exceed the CPU or bandwidth available on a single server. It cannot run non-Linux-compatible OpenFlow switches or applications; this has not been a major issue in practice.

1.2 TCP performance issues

1. **Connect-heavy Applications**
   Some applications instantiate a new TCP connection for each transaction. TCP connection establishment takes time, contributes extra RTTs, and is subject to slow-start. In addition, the closed connections are subject to TIME-WAIT, which consumes system resources.

2. **Head-of-line blocking**
   The TCP subflows of the Multipath TCP connection may go through paths with different characteristics. For example, a subflow going over the smartphones WiFi interface experiences a much lower RTT than a subflow sent over the phones 3G interface. As packets are multiplexed across the different subflows, assuming each subflow goes on a different path, the paths delay difference might cause out-of-order delivery at the receiver. As Multipath TCP ensures in-order delivery, the packets that are scheduled on the low-delay subflow have to wait for the high-delay subflows packets to arrive in the out-of-order queue of the receiver. This phenomenon is known as head-of-line blocking.
   Head-of-line blocking causes burstiness in the data stream by delaying the data delivery to the application, which is undesirable especially for interactive or streaming traffic. Interactive applications will become less reactive, resulting in poor user experience. Streaming applications will need to add a high amount of application-level buffering, stressing the end systems to cope with burstiness and provide a continuous streaming experience to the end user.

1.3 QUEUING AND SCHEDULING

In packet-switched networks, packets contend for access to an outgoing transmission link, since the instantaneous rate at which packets arrive for transmission to that link may exceed the links capacity. This is why packet-switching nodes use buffers, where packets arriving at a rate greater than the link capacity are queued and wait for transmission.
Packet queuing and scheduling is the mechanism that dictates which of the packets waiting in a links buffer will be selected for transmission. Obviously, the simplest scheduling mechanism (or queuing discipline) is first-in-first-out (FIFO) queuing, in which the packets are queued in a single queue in the order that they arrive and are transmitted in that order. This simple discipline guarantees ordering of packets belonging to the same flow, but it suffers from two weaknesses: it permits misbehaving senders to exhaust network resources, and it does not permit differentiation of performance levels.
In order to alleviate the above weaknesses of FIFO queuing, other scheduling disciplines have been developed.
Round-Robin (RR): The round-robin scheduler selects one subflow after the other in round-robin fashion. Such an approach might guarantee that the capacity of each path is fully utilized as the distribution across all subflows is

equal. However, in case of bulk data transmission, the scheduling is not really round-robin, since the application is able to fill the congestion window of all subflows and then packets are scheduled as soon as space is again available in each subflows congestion window. This effect is commonly known as ack-clock.
Lowest-RTT-First (LowRTT): In heterogeneous networks, scheduling data to the subflow based on the lowest round-trip-time (RTT) is beneficial, since it improves the user-experience. It reduces the application delay, which is critical for interactive applications. In other words, the RTT-based scheduler first sends data on the subflow with the lowest RTT estimation, until it has filled its congestion window. Then, data is sent on the subflow with the next higher RTT.
In the same way as the round-robin scheduler, as soon as all congestion windows are filled, the scheduling becomes ack-clocked. The acknowledgements on the individual subflows open space in the congestion window, and thus allow the scheduler to transmit data on this subflow.
Fair Queuing (FQ): Packets of each flow are queued in a separate queue. These queues are then serviced in a round robin mode. Fair queuing extends round-robin queuing by taking into account the count of bytes serviced from each flow rather than the count of packets, so as to implement fairness among flows with large and small packet sizes.
Weighted Fair Queuing (WFQ): This scheme is based on the same principles as fair queuing, with the difference that each queue is assigned a weight and the number of bytes serviced from each queue during each round is proportional to this weight.

2 Experiment Setup

Directions

1. **Environment-Setup**
   `https://github.com/mininet/mininet/wiki/Environment-Setup`

2. Clone the project starter code from Github into the Mininet virtual machine.
   git clone https://bitbucket.org/huangty/cs144bufferbloat.git

3. Enter in cs144bufferbloat folder
   cd cs144bufferbloat/

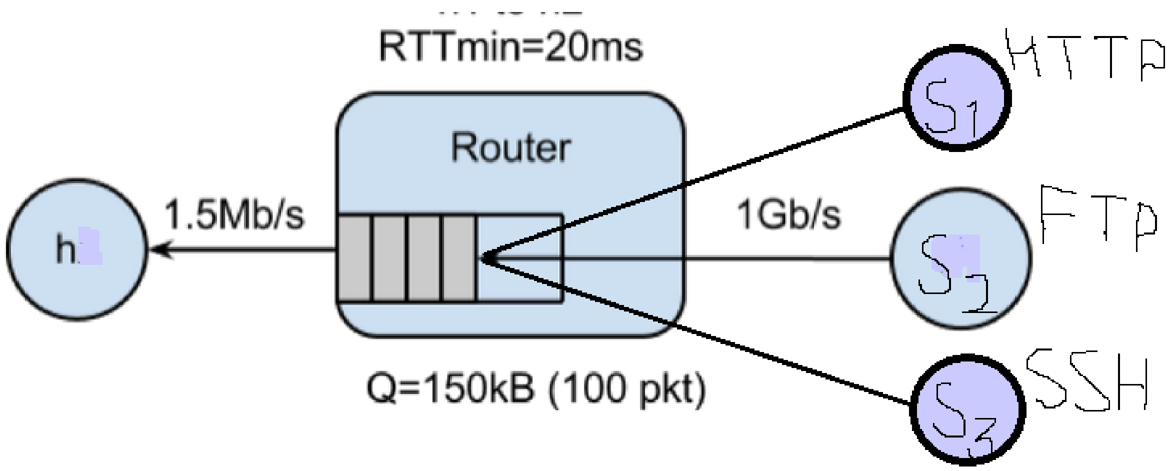
4. Update tc-cmd-diff.sh file(as a requirement)
   In 2-queue problem: build 2-queue, assign 20 and 1500 port number one of the queue and assign 80 and 22 port number another queue.
   In 4-queue problem: build 4-queue and assign each port number to each queue.

5. Update run-diff.sh file(as a requirement)
   In 2-queue problem: set maxq 200(maximum queue size)
   In 4-queue problem: set maxq 100(maximum queue size)
   In both cases to set n 4(number of nodes) one of them client and others are servers.

6. Update monitor.py file like that for 2-queue problem only generate three txt occupancy queue file(two virtual queues and one main queue) and for 4-queue problem generate five txt occupancy queue file(four virtual queues and one main queue).

7. Similarly, update plot-figures.sh file for generating the images(as a requirement) and also update maxy 200(y-axis value).

8. All traffic generated by the different servers shown in below figure except iperf. It is define it self in assingment folder(cs144 bufferbloat) by file name iperf.txt which call by server s1(send iperf traffic at host h).



3 Experiment Result and Analysis

This experiment performed in mininet emulator. In first experiment one queue divided in to two different virtual queues(subflows) of size 200 packets each. In which four TCP traffic directed, ftp and iperf in one of the virtual queue and http and ssh in another virtual queue. After performing the operation result shown in figure 3 and figure 4 where as red graph show its queue accupancy and green graph show its TCP congestion. In second experiment one queue diveded in to four different virtual queues(subflows) of size 100 packets each and each TCP traffic go thought the one of the subflow. After performing the operation result shown in figure 1 and figure 2 where as red graph show queue accupancy and green graph show TCP congestion. Figure 5 shows the final queue accupancy of first experiment and second experiment respectively.

Simulation Parameters	
Host BW	1 Gbps
Network BW	1.5 Mbps
Delay	10 ms
RTT	20 ms
Mim cwnd size	14.5 KB
2-queue size	200 pkt
4-queue size	100 pkt

In the table the average connections one-way propagation time is 0.01 seconds. A 1 gegabit link can forward R packets per second(transmistion delay(in second) * link bandwidth), so a sender can transmit R*p packets(transmistion delay(in second) * link bandwidth * propogation delay(in second)) before the receiver seen the first packet.

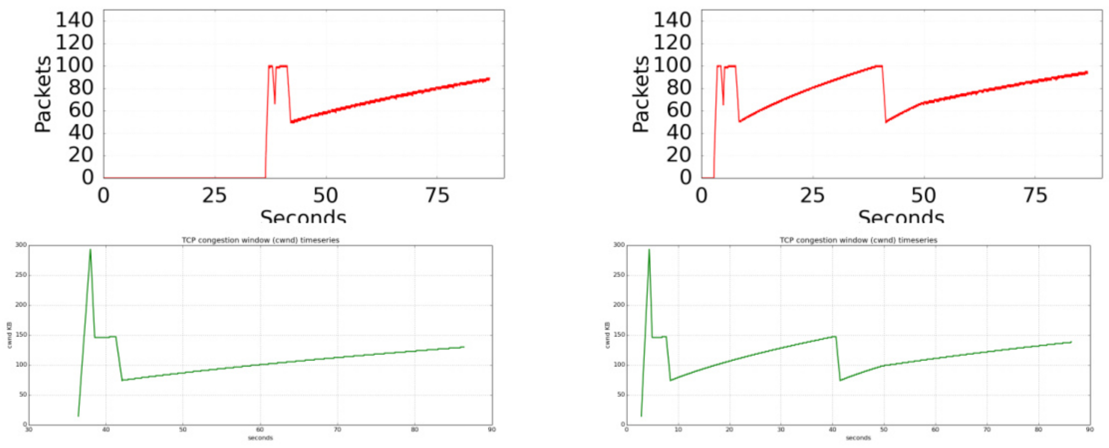


Figure 1: ftp and iperf in different queues

As a shown in figure 1, In ftp TCP congestion window(cwnd) timeseries the slow start phase started at 36 second and on each succesful ack increase cwnd and also at each RTT cwnd becomes 2*cwnd (called exponential growth of cwnd). After 2 second queue was full due to some packet may get dropped and sender not able to get any acknowledgement of dropped packet with in the define timeout intervel and sender suppose to packet get lossed due to congestion, so sender decrease the congestion window size by ssthresh and it goes in congestion avoidance phase. At the same time queue scheduler forword the queuing packet

to correspoing host. Now sender re-transmite the loss packet and increase the value of RTO(retransmission timeout)(as define algorithm) and wait for ack, one again queue become full and it dropped some packet by which sender not able to get any ack of dropped packet and it suppose to that packet get lossed due to congestion than sender agian decrease the congestion window size by current window size by 2 (cwnd/2) and it goes in congestion avoidance phase. Here, at each successful ack cwnd is cwnd+1/cwnd and for each RTT cwnd is cwnd+1 (is called Linear growth of cwnd). Congestion avoidance phase gone upto end of the graph due to this situation queue was not able to get full again and no other packets dropped by queue. So congestion not happend again in the network.
Same thing happend in iperf TCP congestion window(cwnd) timeseries upto 39 second (iperf packet size bigger then the ftp packet size so that queue was full very soon as compare to ftp queue) but at 40th second queue was full in congestion avoidance phase. So, it must queue dropped some packets and sender was not able to get ack of dropped packets and sender suppose to packet got lossed due to congestion in the network, so sender once agian decrease the congestion window size by current window size by 2 (cwnd/2) and agian it goes in congestion avoidance phase(at each successful ack cwnd is cwnd+1/cwnd and for each RTT cwnd is cwnd+1) and continue until agian congestion not occur.

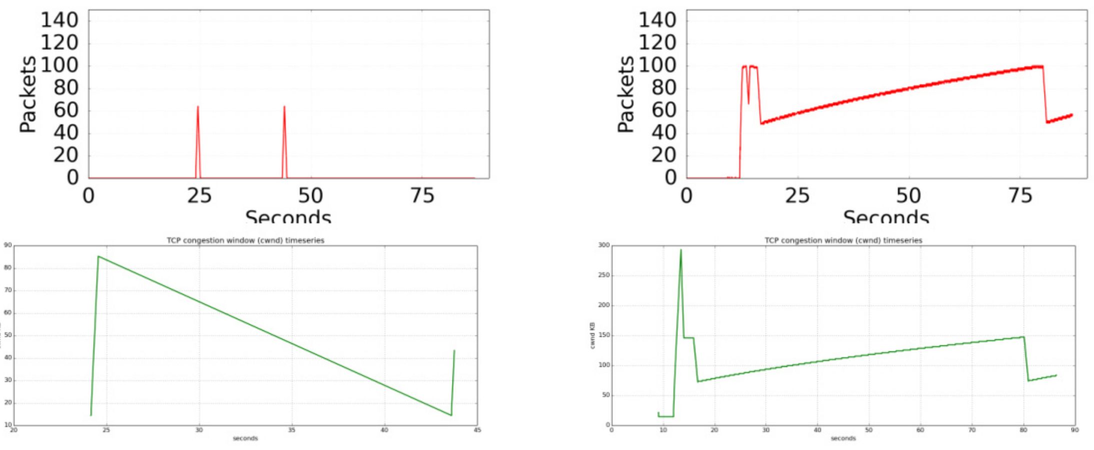


Figure 2: http and ssh in different queues

As a shown in figure 2, In http TCP congestion window(cwnd) timeseries the slow start phase started at 24 sec. because http traffic generated at 24th sec. of experiment started. it goes exponentially(both cwnd and number of packet in queue) and http request finished (less than 1 sec.) before queue get full. Packets not get to dropped and congstion was not happend and agian we generated http traffic at 44 sec. which behaviour was similar to 1st http request. (At each new http request tcp get new connection so slow start phase always

comes in picture).
The ssh TCP congestion window(cwnd) timeseries bahaviour similar to the iperf TCP congestion window(cwnd) timeseries and also queueing bahaviour is same but difference in traffic time generation. we know that ssh packets are smaller than other TCP packet size but in this experiment we are generating lots of the small ssh packets by running the shell command in infinite loop as a result packets generation is very high due to that its congestion is very high and its similar to iperf TCP congestion window(cwnd) timeseries which is dscribe in figure 1.Here the difference between iperf and ssh TCP congestion window(cwnd) timeseries is that due to big packets in iperf queue was filled very soon as compare to ssh queue so packets dropped start (when queue get full) before ssh packet get dropped (because queue not full at this time, it may full later). In ssh after long time period (after 85 sec.) the second congastion avoidance phase came but in iperf it came with in the 30 second.

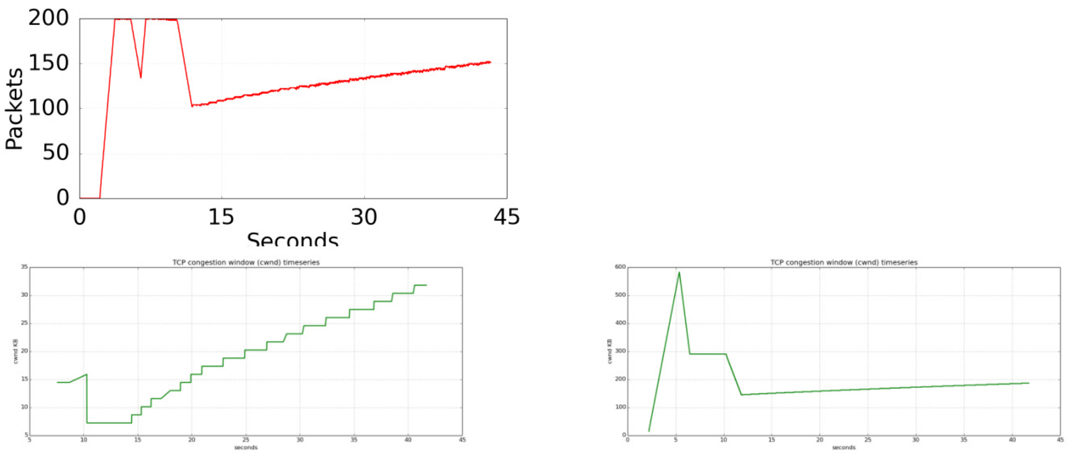


Figure 3: ftp and iperf in same queue

As a shown in figure 3, there was only one comman queue for both ftp and iperf traffic. In experiment iperf get executed few second before the ftp. In iperf TCP congestion window(cwnd) timeseries the slow start phase started at 3 sec. and its graph does exponentially(for each successful RTT cwnd becomes 2*cwnd) at some point of time queue get full(at 5th sec.) due to high traffic of iperf packet in the network. If sender sends one more packet to the network, queue will drop that packet so sender not able to get ack to that dropped packet within the timeout intervel and its suppose to packet get lossed due to congestion on the network. Re-send the lossed packet and increase the timeout intervel also sender decrease the congestion window size by ssthresh and it enter in congestion avoidance phase. At the same time (when sender wait for ack of droped packet) queue scheduler schedule the queueing packet to its corresponding destination so at 6.5th sec. queue has some space which is shown in queueing diagram and on 7th sec.(approximatly) on words congastion avoidence phase started for iperf but on the other side ftp get started (slow start phase)and send the packet at the same time, because we have comman queue for both, queue once again get full due to ftp packet and iperf packect not got a chance to enter in queue(congestion avoidance phase),so iperf packet get droped and also ftp packet get start to dropping due to queue become full. Both senders(ftp and iperf) wait for ack but both was not able to get ack of dropped packet within the predefine timeout interval. So at 10th sec. both congastion windows get dropped by its current window size by 2 (cwnd/2) and here to start congestion avoidance phase in both cases(for each RTT cwnd is cwnd+1) and both types of packet goes in the queue and scheduler has schedule the queuing packet this process will continue untill queue will not get full.

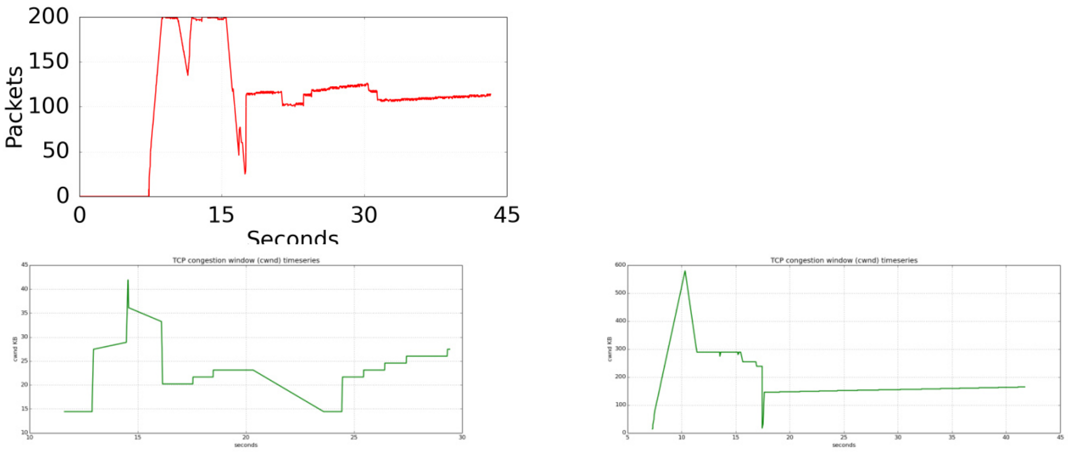


Figure 4: http and ssh in same queue

As a shown in figure 4,similar to above there is a comman queue for both http and ssh traffic. In experiment ssh get executed few second before the http. In ssh TCP congestion window(cwnd) timeseries the slow start phase started at 6.5th sec. and its graph does exponentially(for each successful RTT cwnd becomes 2*cwnd) after some point of time queue get full(at 10th sec.) due to high traffic of ssh packet in the network because queue get full then it must be dropped any incomming packet in the queue. If sender not able to get ack to that dropped packet within the timeout intervel and its suppose to packet get lossed due to congestion on the network. Re-send the lossed packet and increase the timeout intervel and also sender decrease the congestion window size by ssthresh and it enter in congestion avoidance phase. In pallal (when sender wait for ack of dropped packet) queue scheduler schedule the queueing packet to its corresponding destination so at 11th sec. queue has some space which is shown in queueing diagram. At 11th sec.(approximatly) onwords congastion avoidence phase started for ssh and on the other side http get started with slow

start phase and send the packet at the same time both of them, because we have comman queue for both, queue once again get full due to http packet and ssh packect not got a chance to enter in queue(fewer packet in a queue got a chance due to congastion avoidence phase), many of ssh packet get dropped and also http packet get start to dropping due to queue become full. Now sender wait for ack but it was not able to get ack of dropped packet within the predefine timeout interval. So at 14.5th sec. http congastion windows get dropped by its current window size by 2 (cwnd/2) and start congestion avoidance phase (for each RTT cwnd is cwnd+1) at 16th second and on 20 second the first http request has completed now at 24th second on words the second http request started with new TCP connection so its get started with slow start phase.
By observing ssh diagram here we can notice that at 15th sec. cwnd get dropped due to 3 duplicate ack(use duplicate ACKs to signal lost packet) because it use fast retransmission(three duplicate ACKs, the TCP Sender retransmits the lost packet) and fast recovery(after Fast Retransmit, set ssthresh to cwnd/2, set cwnd=1 and begin the exponential slow-start process until cwnd=ssthresh, and then increase cwnd linearly.) mechanism at 17.5th sec. onwords.

As a shown in figure 5, it shown the behaviour of the 4-queue and 2-queue

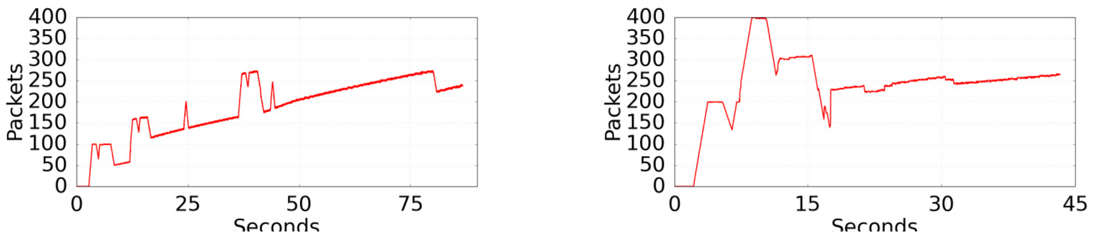


Figure 5: final queues(4q,2q)

respectivity. Here we notice that in 2-queue mechanism congestion occurrence very frequently as compare to 4-queue but disadvantages with 4-queue is not utilized very well.

4 Conclusion

After performing experiment and analyse the experiment result we observed that to assign individual queue for each TCP traffic to assign one queue some similar kind of data when many discipline are in network and we also shown that when two different type of transmit mechanism apply over the queue with different congestion protocol like one is slow-start and another is congestion avoidance than number of packet which are generated in slow-start (one of the TCP packet)greater then other type of packet(which is generated in congestion avaidance phase) in comman queue. We seen the reno protocol behaviour in different situations(in both timeout and three duplicate ack).

5 Acknowledgement

I would like to express my deepest appreciation to all those who provided me the possibility to complete this report. One of my dear friend Mr. Gaurav Jhosi whos helped me to complete this experiment successfully because without experiment there are no way to write term paper.

References

1. `https://github.com/mininet/mininet/wiki/Bufferbloat`.
2. `http://ftp.isi.edu/in-notes/rfc2581.txt`.
3. `https://tools.ietf.org/html/rfc2582`.
4. TCP/IP illustrated volume 1 the protocols by W. Richard Stevens