

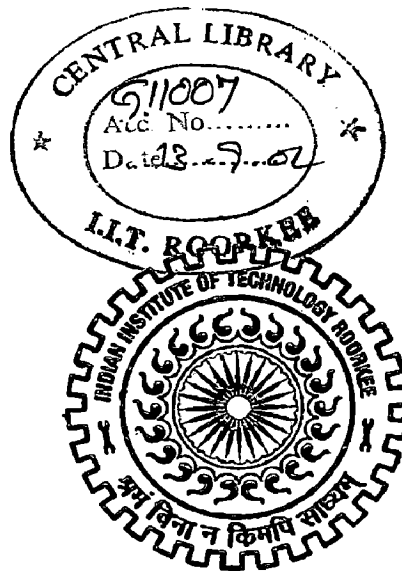
CONGESTION CONTROL AND AVOIDANCE IN COMPUTER NETWORKS

A DISSERTATION

*Submitted in partial fulfilment of the
requirements for the award of the degree
of*
MASTER OF COMPUTER APPLICATIONS

By

RAJNEESH KUMAR



DEPARTMENT OF MATHEMATICS
INDIAN INSTITUTE OF TECHNOLOGY ROORKEE
ROORKEE-247 667 (INDIA)

JUNE, 2002

CANDIDATE'S DECLARATION

I here by declare that the work presented in the dissertation titled "**CONGESTION CONTROL AND AVOIDANCE IN COMPUTER NETWORKS.**" In partial fulfillment of the requirements for award of the degree of **Master of Computer Applications**, submitted in the Department of Mathematics, Indian Institute of Technology, Roorkee is an authentic record of my own work carried out during the period from 1st February to 31st May. Under the guidance of **Mr. M. K. Sharma**, Senior System Analysis in **DENSO HARYANA PVT. LTD.** Gurgaon, Haryana and **Prof. Vinod Kumar**, Department of **Electrical Engineering**, Indian Institute of Technology, Roorkee.

The matter embodied in this dissertation has not been submitted by me for the award of any other degree of diploma.

Date : June 9 ,2002


(Rajneesh Kumar)

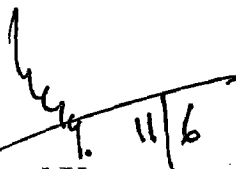
Place : Gurgaon

*****CERTIFICATE*****

This is to certify that the above statement made by the candidate is correct to the best of my knowledge and belief.



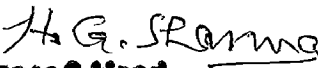
Mr. M. K. Sharma
(System Analyst)
Denso Haryana Pvt. Ltd.
Sector # 3, Gurgaon



Dr. Vinod Kumar
(Professor)
Deptt. of Elect. Engg.
Indian Institute of Technology,
Roorkee

Date : 9/6/02
Place : Gurgaon

Date :
Place : Roorkee

forwarded

Professor & Head
Department of Mathematics
I.I.T. Roorkee-247 667 12.6.02

ACKNOWLEDGEMENT

It is my great pleasure to take this opportunity to express my appreciation and thanks to my guides **Prof. VINOD KUMAR**, Department of Electrical Engineering, IIT-Roorkee, Roorkee and **Mr. M.K. Sharma**, Senior System Analyst in DENSO HARYANA Pvt. Ltd. Gurgaon(HARYANA). They cheerfully spared their valuable time and effort to Complete this work successfully.

I am also thankful to **Prof. R.C.Mittal**, Department of Mathematics , IIT-Roorkee, Roorkee for their valuable suggestions .I am very

Thankful to the Lab incharge of BSNL-N. Delhi for their constant support. It was a unique experience to work under their guidance because of endless sources of information.

I am also thankful to **Dr. H. G. Sharma**, Prof. and Head Department of Mathematics, IIT-Roorkee for his timely support in completion of my work.

Lastly, I am thankful to all those who helped me directly or indirectly in the successful completion of this work.


(RAJNEESH KUMAR)

ABSTRACT

Computer networks form an essential substrate for a variety of distributed applications, but they are expensive to build and operate. This makes it important to optimize their performance so that users can derive the most benefit at the least cost. Though most networks perform well when lightly used, problems can appear when the network load increases. Loosely speaking, congestion refers to a loss of network performance when a network is heavily loaded. Since congestive phenomena can cause data loss, large delays in data transmission, and a large variance in these delays, controlling or avoiding congestion is a critical problem in network management and design. This dissertation work presenting the implementation details for congestion control in computer networks.

Since these networks carry traffic of a single type, and the traffic behavior is well known, it is possible to avoid congestion simply by reserving enough resources at the start of each call. By limiting the total number of users, each admitted call can be guaranteed to have enough resources to achieve its performance target, and so there is no congestion. However, resources can be severely underutilized, since the resources blocked by a call, even if idle, are not available to other calls.

CONTENTS

Candidate's declarations	i
Acknowledgement	ii
Abstract	iii
Contents	iv
1. Chapter 1	
Introduction	
1.0 Introduction to congestion control	1.
1.1 Some reasons to congestion control	2.
1.2 New Definition	3.
1.3 Genrel Principal to congestion control	5.
2. Chapter 2	
Congestion control schemes	8.
For TCP/IP Networks.	
2.1 Introduction	8.
2.2 Compatiblity with Technology	8.
2.3 Complexity	8.
2.4 Slow start Algo..	9.
2.5 Algorithm	9.
2.6 Performance	11.
2.8 DUAL	12.
2.9 Algorithm	12.
2.10 Performance	12.
2.11 Discussion	13.
2.12 TCP Vegas	13.
2.13 Algorithm	14.

2.15 Gateway based schemes	15.
2.16 Random Early Detection	16.
2.17 Algorithm	16
2.19 Performance	17.
2.20 discussion	17.
Chapter 3	18.
3.1 Introduction	18.
3.2 TCP review	20.
3.4 ACK Division	22.
3.5 Dup ACK spoofing	24.
3.6 Optimistic ACKing	27.
3.7 Desiging Robost protocol	30.
3.8 ACK division	30.
3.9 Dup ACK spoofing	31.
3.10 Optimistic ACKing	32.
3.13 About Misbehaving.....	33.
Chapter 4	36.
4.1 Introduction	36.
4.2 Environment of	37.
4.3 What is congestion	39.
4.4 New definition	40.
4.5 Congestion Control	41.
4.5.1 Proactive And reactive ...	42.
4.5.2 Time scales of congestion	44.
4.5.3 Session	46.
4.5.7 Need for congestion.....	47.
4.6 Fundamental assumptions	48.

4.6.1 Administrative Control	48.
4.6.2 Source complexity	49.
4.6.3 Gateway Complexity	49.
4.6.4 Bargaining Power	50.
4.6.5 Responsibility for congestion control	50.
4.6.7 Traffic Model	51.
4.7.1 Reservation Network	52.
4.7.2 Congestion Detection	52.
4.7.3 communication	52.
4.7.5 Flow control	53.
4.7.7 Reservation – Oriented Network	53.
4.7.9 Delay	54.
4.7.10 Underutilization	54.
4.7.12 Quality of service	55.
Programming Part	56.
REFERENCES	85.

INTRODUCTION

1.0 Introduction To Congestion control :

When too many packets are present in the subnet the performance of subnet degrades. This situation is called **congestion**, Fig 1 depicts the symptoms. when no. of Packets dumped into the subnet by the hosts is within its carrying capacity, they are delivered (except for a few that are afflicted with transmission errors) and the no. of delivered packets is proportional to the no. of packets sent. However, as traffic increases too far, the routers are no longer able to cope, and they begin losing packets. This tends to make matters worse. At very high traffic, performance collapses completely, and almost no packets are delivered.

Fig 1.

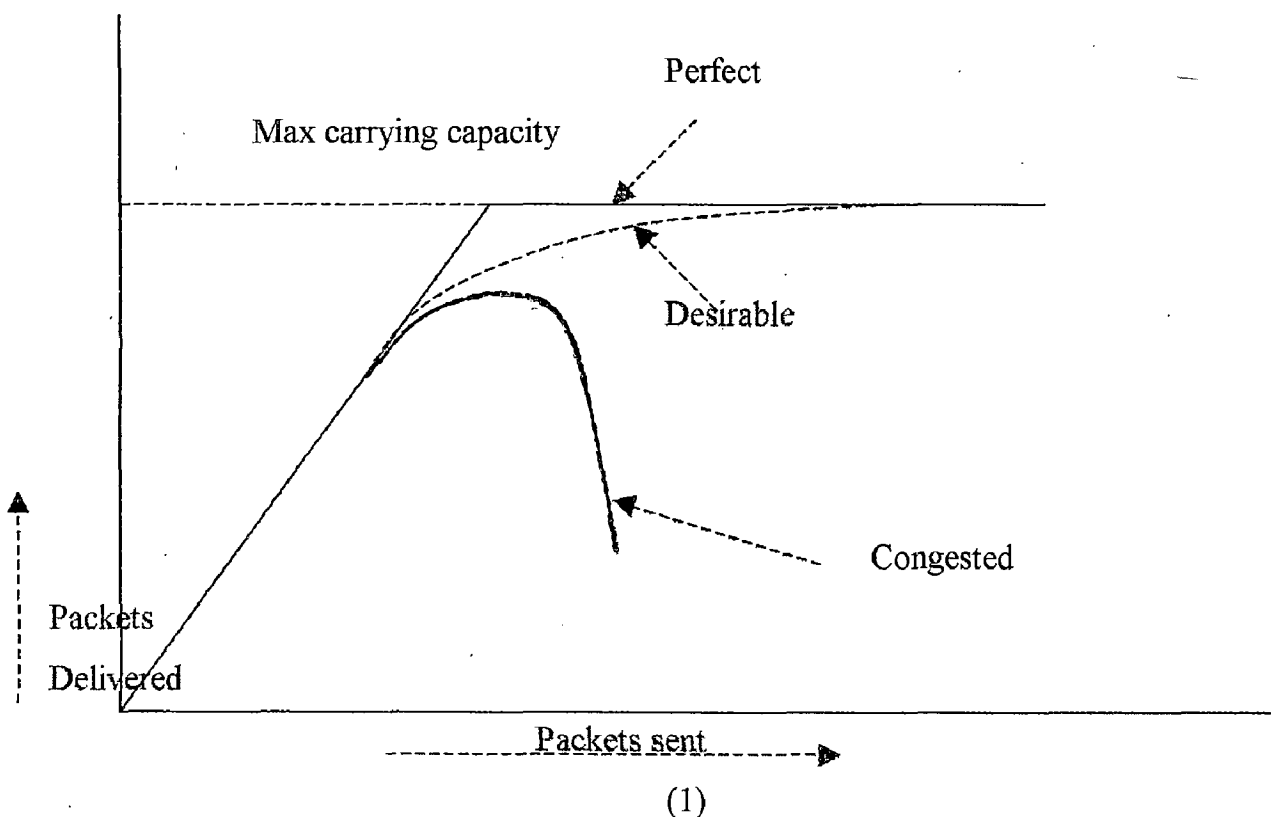


Fig. 1 : When too much traffic is offered ,congestion sets in and performance degrades sharply. congestion can be brought by several factors. If all of a sudden streams of packets begin arriving on three or four input lines and all need the same output line ,a queue will build up . If there is insufficient memory to hold all of them , packets will be lost. Adding more memory may help up to a point, but important point is takes place here is that routers have an infinite amount of memory,congestion get worse,not better , because by the time packets get to the front of the queue,they have already timed out (repeatedly), and duplicates have been sent . All the packets will dutifully forwarded to the next router , increasing the load all the way to the destination

Since congestion occurs at high network loads, definitions of congestion focus on some aspect of network behavior under high load. We first discuss a scenario that leads to network

1.1 Some reasons to congestion.....

Slow processors can also cause congestion .If the routers CPU's are slow at performing the book keeping tasks required of them (queuing buffers ,updating tables etc.) queues can build up ,even though there is excess line capacity. Similarly low bandwidth lines can also cause congestion. upgrading the lines but not changing the processors, or vice versa ,often helps a little but frequently just shift the bottleneck ,also upgrading part but not all of the system ,often just moves the bottleneck somewhere else. The real problem will persist until all the components are in balance.

The standard definitions of congestion are thus of the form: "A network is congested if, due to overload, condition X occurs", where X is excessive queueing delay, packet loss or decrease in effective throughput.

These definitions are not satisfactory for several reasons. First, delays and losses are indices of performance that are being improperly used as indices of congestion, since the change in the indices may be due to symptoms of phenomena other than congestion. Second, the definitions do not specify the exact point at which the network can be said to be congested

(except in a deterministic network, where the knee of the load-delay curve, and hence congestion, is well defined, but that is the trivial case). For example, while a network that has mean queueing delays in each switch of the order of 1 to 10 service times is certainly not congested, it is not clear whether a network that has a queueing delay of 1000 service times is congested or not. It does not seem possible to come up with any reasonable threshold value to determine congestion!

Third, a network that is congested from the perspective of one user is not necessarily congested from the perspective of another. For example, if user A can tolerate a packet loss rate of 1 in 1000, and user B can tolerate a packet loss rate of 1 in 100, and the actual loss rate is 1 in 500, then A will claim that the network is congested, whereas B will not. A network should be called uncongested only if all the users agree that it is.

1.2 New definition

From the discussion above, it is clear that network congestion depends on a user's perspective. A user who demands little from the network can tolerate a loss in performance much better than a more demanding user. For example, a user who uses a network only to send and receive electronic mail will be happy with a delivery delay of a day, while this performance is unacceptable for a user who uses a network for real-time audio communication. The key point is the notion of the *utility* that a user gets from the network, and how this utility degrades with network loading.

The concept of 'utility' used here is borrowed from economic theory. It is used to refer to a user's preference for a resource, or a set of resources (often called a resource bundle).

strictly speaking, the utility of a user is a number that represents the relative preference of that user for a resource (or performance) bundle, so that, if a user prefers bundle A to bundle B, the utility of A is greater than the utility of B. For example, if A is {end-to-end delay of 1 second, average throughput 200 pkts/second}, and B is {end-to-end delay of 100 seconds, average throughput 20000 pkts/second}, a user may prefer A to B, and we would assign a utility to A that is greater than the utility of B, while another user may do the opposite.

In classic microeconomic theory, utilities are represented by a function over the resources. Since utilities express only a preference ordering, utility functions are insensitive to monotonic translations, and the utilities of two users cannot be compared; the function can only be used to relatively rank two resource bundles from the point of view of a single user.

An example of a utility function is

$$\alpha T - (1-\alpha)RTT, \quad \text{Ref-no.6}$$

where α is a weighting constant,

T is the average throughput over some interval,

and RTT is the average round-trip-time

delay over the same interval. As the throughput (T) increases, the utility increases, and as delays increase, the utility decreases. The choice of α determines the relative weight a user gives to throughput and delay.

A delay-sensitive user will choose $\alpha \rightarrow 0$, whereas a delay-insensitive user's $\alpha \rightarrow 1$.

Ref-no.5

In practice, a utility function may depend on a threshold. For example, a user may state that he or she is indifferent to delay, as long as it is less than 0.1 seconds. Thus, if the user gets a delay of 0.05 seconds during some interval of time, and 0.06 seconds in a later period, as far as the user is concerned, there has been no loss of utility. However, if some user's utility *does* decrease as a result of an increase in the network load, that user will perceive the network to be congested. This motivates our definition.

Congestion tends to feed upon itself and become worse. If a router has no free buffers, it

must ignore newly arriving packets .When a packets is discarded ,the sending router (a neighbor)may time out and retransmit it perhaps ultimately many times . Since it can not discard the packet until it has been acknowledged , congestion at the receiver's end forces the sender to refrain from releasing the buffer it would have normally freed . in this manner , congestion backs up , likes cars approaching a toll booth.

It is worth explicitly pointing out the difference between congestion control and flow control , as the relationship between is subtle . congestion has to do with making sure the subnet is able to carry the offered traffic . It is global issue ,involving the behavior of all the hosts ,all the routers, the store and forwarding processing within the routers ,all the other factors that tends to diminish the carrying capacity of the subnet.

Flow control ,in contrast ,relates to the point to point traffic between a given sender and the receiver .Its job is to make sure that a fast sender can not continually transmit data faster than the receiver can absorb it. Flow control nearly always involves some direct feedback from the receiver to the sender to tell the sender how things are doing at the other end.

The reason congestion control and flow control are often confused is that some congestion control algorithms operate by sending messages back to the various sources telling them to slow down when the network gets into the trouble . thus host can get a “ slow down” message either because receiver can not handle the load , or because the network cannot handle it .

We will come back to this point later.

1.3 Genrel principles of congestion control

Many problems in complex systems , such as computer networks , can be viewed from a control theory point of view . This approach leads to dividing all solutions into two groups :

- (1) open loop.
- (2) Closed loop.

- (5)

Open loop solutions attempt to solve the problem by good design, in essence, to make sure it does not occur in the first place. Once the system is up and running, midcourse corrections are not made.

Tools for doing open loop control include deciding when to accept new traffic, deciding when to discard packets and which ones, and making scheduling decisions at various points in the network. All of these have in common the fact that they make decisions without regard to the current state of the network.

In contrast, closed loop solutions are based on the concept of a feedback loop. This approach has three parts when applied to congestion control:

1. Monitor the system to detect when and where congestion occurs.
2. Pass this information to places where action can be taken.
3. Adjust system operation to correct the problem.

Various metrics can be used to monitor the subnet for congestion.

Ref-no. 10

Among these are the percentage of all packets discarded for lack of buffer space, the average queue lengths, the number of packets that time out and are retransmitted, the average packet delay, and the standard deviation of packet delay. In all cases, rising numbers indicate growing congestion.

The second step in the feedback loop is to transfer the information about the congestion from the point where it is detected to the point where something can be done about it. The obvious way is for the router detecting the congestion to send the packet to the traffic source or sources announcing the problem. Of course, these extra packets increase the load at precisely the moment that more load is not needed, namely, when the subnet is congested.

The presence of congestion means that the load is (temporarily) greater than the

resources (in the part of the system) can handle. Two solutions come to the mind : increases the resources or decreases the load. for example the subnet may start using dial-up telephone lines to temporarily increase the bandwidth at between certain points .In systems like SMDS ,(Switched Multimegabit Data Service)it may ask the carrier for additional bandwidth for a while.

On satellite systems ,increasing transmission power often gives higher bandwidth .Splitting traffic over multiple routes instead of always using the best one may also effectively increase the bandwidth .finally, spare routers that are normally used only as backups (to make the systems fault tolerant) can be put on-line to give more capacity when serious congestion appears.

Ref-no.2

However , sometimes it is not possible to increase the capacity,or it has already been increased to the limit. The only way than to beat back the congestion is to decreases the load . several ways exist to reduce the load , including denying service to some users schedule their demands in a more predictable way.

2.1 Introduction:

Here we consider few methods for controlling congestion for TCP connections. The first three are Slow Start algorithm , DUAL , and TCP Vegas Treat the network as a black box, in that the only way to detect congestion is through packet loss and changes in round trip time, or throughput. The last two, Random Early Detection and Explicit Congestion Notification depend on the gateways to provide indications of congestion.

The algorithms are judged in several categories:

Performance.

The algorithm is judged by increases in throughput and decreases in retransmission as compared to other algorithms.

The algorithm is judged by how well connections share the resources with other connections and whether there are any biases towards connections with certain characteristics such as burstiness.

2.2 Compatibility with current technology.

The algorithm is judged by how well it interacts with current technology. Are the gains of this algorithm at the cost of other connections not using this algorithm? How will the algorithm perform in the presence of non-compliant sources?

2.3 Complexity.

The algorithm is judged by the complexity of implementation. Algorithms with lower overhead are preferred. In the first section we will compare algorithms which use changes

in performance, i.e., packet loss, increase in round trip time, change in throughput, to detect congestion. These algorithms have the advantage that they require only a new implementation of TCP and do not involve changing the network, as opposed to the algorithms discussed in the second section which require changing gateways and possibly adding fields to IP packets.

2.4 Slow Start algorithm:

Ref-no.3

Introduction

Jacobson and Karels developed a congestion control mechanism for TCP following a congestion collapse on the internet. Prior to this no congestion control mechanism was specified for TCP. Their method is based on ensuring the 'conservation of packets,' i.e., that the packets are entering the network at the same rate that they are exiting with a full window of packets in transit. A connection in this state is said to be in equilibrium. If all connections are in equilibrium, congestion collapse is unlikely. The authors identified three ways for packet conservation to be violated:

1. The connection never reaches equilibrium.
2. A source sends a new packet before an old one exits.
3. Congestion in the network prevents a connection from reaching equilibrium.
4. TCP is 'self clocking,' i.e., the source sends a new packet only when it receives an ack for an old one and the rate at which the source receives acks is the same rate at which the destination receives packets.

So the rate at which the source sends matches the rate of transmission over the slowest part of the connection.

2.5 Algorithm

To ensure that the connection reaches equilibrium, i.e., to avoid failure (1), a *slow-start* algorithm was developed. This algorithm added a congestion window. The minimum of the *congestion window* and the destination window is used when sending packets. Upon starting a connection, or

restarting after a packet loss, the congestion window size is set to one packet. The congestion window is then increased by one packet upon the receipt of an ack. This would bring the size of the congestion window to that of the destination window in $RTT \log_2 W$ time, where RTT is the round-trip-time and W is the destination window size in packets. Without the slow start mechanism an 8 packet burst from a 10 Mbps LAN through a 56 Kbps link could put the connection into a persistent failure mode.

if the retransmit time is too short, making the source retransmit a packet that has not been received and is not lost. What is needed is a good way to estimate the round trip time:

$$Err = Latest_RTT_Sample - RTT_Estimate$$

$$RTT_Estimate = RTT_Estimate + g*Err$$

Ref-no.6

where g is a 'gain' ($0 < g < 1$) which is related to the variance. This can be done quickly with integer arithmetic. This is an improvement over the previous method which used a constant to account for variance. The authors also added exponential backoff for retransmitting packets that needed to be retransmitted more than once. This provides exponential dampening in the case where the round trip time increases faster than the RTT estimator can accommodate, and packets, which are not lost, are retransmitted. Lost packets are a good indication of congestion on the network. The authors state that the probability of a packet being lost due to transit is very rare. Furthermore, because of the improved round trip timer, it is a safe assumption that a timeout is due to network congestion. A additive increase / multiplicative decrease policy was used to avoid congestion.

Upon notification of network congestion, i.e., a timeout, the congestion window is set to half the current window size. Then for each ack for a new packet results in increasing the window by $1/congestion_window_size$. Note that if a packet times out, it is most likely that the source window is empty and nothing is being transmitted and a slow-start is required. In that case, the slow-start algorithm will increase the window (by one packet

per ack) to half the previous window size, at which point the congestion avoidance algorithm takes over.

2.6 Performance

To test the effectiveness of their congestion control scheme, they compared their implementation of TCP to the previous implementation. They had four TCP conversations going between eight computers on two 10 Mbps LANs with a 230.4 Kbps link over the internet. They saw a 200% increase in effective bandwidth with the slow-start algorithm alone. The original implementation used only 35% of the available bandwidth due to retransmits. In another experiment, using the same network setup, the TCP implementation without congestion avoidance resulted in 4,000 of 11,000 packets sent were retransmitted packets as opposed to 89 of 8,281 with the new implementation.

Ref-no. 6

2.7 Discussion

According to this scheme suffers from oscillations when the network is overloaded. Because the window size is increased until a packet is dropped to indicate congestion, the bottleneck node is kept at maximum capacity. The window size oscillates between the maximum window size allowable by the bottleneck and half that size on timeouts. This leads to long queuing delays and high delay variation.

Wang, et. al., also point out that this scheme is biased toward connections with fewer hops. However, this would take many iterations, in which time, the shorter connections, with shorter round trip times, would increase faster.

2.8 DUAL

Introduction

In Wang et. al. propose a method called DUAL to correct the oscillation problem associated with the Slow Start algorithm. This is similar to the algorithm described in Ref-no.9 in that it uses the round trip time to detect congestion as well as packet loss as in the Slow Start algorithm.

The round trip time of a packet consists of the propagation delay and the queuing delay. The minimum round trip time would be equal to the propagation delay:

$$RTT_{min} = D_p$$

The maximum round trip time would be the sum of the propagation delay and the delay for the bottleneck node to process a full queue:

$$RTT_{max} = D_p + \text{Max_Queue_Size}/\text{Processing_Rate}$$

The Slow Start algorithm detects congestion when a packet is lost due to a queue overflow. The solution used in DUAL is to estimate RTT_{min} and RTT_{max} and using a threshold, avoid overflowing the queue at the bottleneck node. They defined this threshold as follows:

$$RTT_i = (1-\alpha)RTT_{min} + \alpha RTT_{max} \quad \text{Ref-no.9}$$

for some $\alpha < 1$. They chose $\alpha = 0.5$ to stay well away from the maximum queue capacity.

2.9 Algorithm

DUAL uses the same slow-start algorithm for initiating and restarting a connection. The algorithm differs from Slow Start in that every two round trip times DUAL checks if the current RTT is greater than RTT_i and reduces the congestion window by 7/8. It also recomputes RTT_{max} and RTT_{min} with every new RTT measurement. On a timeout, in addition to reducing the congestion window to one packet and restarting slowly as in the Slow Start algorithm, it resets RTT_{max} and RTT_{min} to 0 and infinity respectively.

2.10 Performance

DUAL was simulated in under three scenarios:

- 1) single connection on a path,
- 2) two connections share a path but one starts before the other and
- 3) two way traffic.

In the first scenario the DUAL showed an almost identical slow start phase, it did show a substantial reduction of oscillations as compared with the Slow Start algorithm.

The second scenario tests the ability of the algorithm to adjust to the addition of connections. The results are similar to that observed in scenario one, except that as the algorithms progress, the window sizes of the connections converge. This is exactly what is expected.

The last scenario tests the effect of rapid queue fluctuation on the round trip time based algorithm. The algorithm performed well despite the fluctuations. Wang, et. al. attribute this to the fact that many of the packets in the queue are ack packets which are small and therefore the actual queuing delay change is smooth.

Ref-no.3

2.11 Discussion

Because the window size is based on RTTmin and RTTmax, their accurate estimation is important. The Wang, et. al. noted that RTTmax is fairly accurate when calculated just before a buffer overflow. However if RTTmin is estimated when there are multiple connections on the path the RTTi threshold would be too high. When different flows obtain different RTTmin values the bandwidth is shared unevenly. Wang et. al. claim that in their simulations, during a slow start the load is relatively light so queuing time is low and that the RTTmin estimate was a reasonable approximation of the propagation delay.

2.12 TCP Vegas

Ref-no.3

Introduction

TCP Vegas is a new implementation of TCP proposed by Brakmo et. al. in . The authors claim an 40 to 70% increase in throughput and one fifth to one half of the packet losses as compared to the current implementation of TCP (which implements the Slow Start algorithm with the addition of Fast Retransmit and Fast Recovery). Vegas compares the measured throughput rate to the expected, or ideal, throughput rate.

2.13 Algorithm

Vegas uses a new retransmission mechanism. This is an improvement over the Fast Retransmit mechanism. In the original Fast Retransmit mechanism, three duplicate acks indicate the loss of a packet, so a packet can be retransmitted before it times out. Vegas uses a timestamp for each packet sent to calculate the round trip time on each ack received. When a duplicate ack is received Vegas checks to see if the difference between the timestamp for that packet and the current time is greater than the timeout value. If it is, Vegas retransmits the packet without having to wait for the third duplicate message. This is an improvement in many cases the window may be so small that the source will not receive three duplicate acks, or the acks may be lost in the network.

Upon receiving a non-duplicate ack, if it is the first or second ack since a retransmission, Vegas checks to see if the time interval since the packet was sent is larger than the timeout value and retransmits the packet if so. If there are any packets that have been lost since the retransmission they will be retransmitted without having to wait for duplicate acks.

To avoid congestion, Vegas compares the *actual* throughput to the *expected* throughput. The expected throughput is defined as the minimum of all measured throughputs. The actual throughput is the number of bytes transmitted between the time a packet is transmitted and its ack is received divided by the round trip time of that packet.

Vegas then compares the difference of the expected and the actual throughputs to thresholds a and b . When the difference is smaller than a , the window size is increased linearly and when the difference is greater than b the window size is decreased linearly. when the bottleneck is finally overloaded, the expected losses are half the current window. As network bandwidth increases the number of packets lost in this manner will also increase. Brakmo et. al. propose a modified slow start mechanism where the window size is doubled only every other round trip time. So every other round trip time the window is not changed which allows for an accurate comparison of the expected

and actual throughput. The difference is compared to a new threshold called c at which point the algorithm switches to the linear increase / decrease mode described above.

2.14 Performance

Vegas was simulated to test its performance when used in the same environment as Reno. In this experiment, a 1MB file was transferred using one implementation with a 300KB file being transferred at the same time with another implementation. In all four combinations, Vegas performed better with higher throughput and fewer retransmissions. In another experiment the performance of Vegas, with two sets of values for a and b , were compared on a network with background traffic. For both configurations of Vegas, throughput was increased by more than 50% and the number of retransmissions was about half that of Reno. Other experiments included an experiment like the first one but with background traffic. The results were similar to that of the first experiment. Again the results were similar to those in the second experiment.

Another experiment was to test the fairness of both schemes by putting multiple connections through a bottleneck. They tested the scheme where some connections that had different propagation delays and where the propagation delays were the same.

2.16 Gateway based schemes:-

A problem with end to end congestion control schemes is that the presence of congestion is detected through the effects of congestion, e.g., packet loss, increased round trip time, changes in the throughput gradient, etc., rather than the congestion itself e.g. overflowing queues. There can also be a problem with fairness and non-compliant sources. It seems logical then to place the congestion control mechanism at the location of the congestion, i.e., the gateways. The gateway knows how congested it is and can notify sources explicitly, either by marking a congestion bit, or by dropping packets. The main drawback to marking packets with a congestion bit, as opposed to simply dropping them, is that TCP makes no provision for it currently. Floyd in states that some have proposed sending Source Quench packets as ECN messages. Source Quench messages have been criticized as consuming network bandwidth in a congested network making the problem worse.

2.17 Random Early Detection:-

Introduction

One method for gateways to notify the source of congestion is to drop packets. This is done automatically when the queue is full. The default algorithm is when the queue is full drop the any new packets. This is called Tail Drop. Anóther algorithm is when the queue is full and a new packet arrives, one packet is randomly chosen from the queue to be dropped. The drawback to Tail Drop and Random Drop gateways is that it drops packets from many connections and causes them to decrease their windows at the same time resulting in a loss of throughput. Early Random Drop gateways are a slight improvement over Tail Drop and Random Drop in that they drop incoming packets with a fixed probability whenever the queue size exceeds a certain threshold.

2.18 Algorithm

Floyd proposes a new method called Random Early Detection (RED) gateways. In this method, once the average queue is above a certain threshold the packets are dropped (or marked) with a certain probability related to the queue size. In describing this paper we will consider only the case were the RED gateway drops packes to indicate congestion rather than marking them. In describing the next algorithm we will discuss marking packets.

To calculate the average queue size the algorithm uses an exponentially weighted moving average:

$$\text{avg} = (1-wq)\text{avg} + wq*\text{Queue_Size}$$

The author goes into detail describing how to determine the upper and lower bounds for wq.

The probability to drop a packet, pb, varies linearly from 0 to maxp as the average queue length varies from the minimum threshold, minth, to the maximum threshold, maxth. The chance that a packet is dropped is also related to the size of the packet. The probability to

drop an individual packet, p_a , increases as the number of packets since the last dropped packet, count , increases:

$$p_b = \frac{\text{maxb}(\text{avg}-\text{minth})}{(\text{maxth}-\text{minth})}$$

$$p_b = p_b * \text{Packet_Size} / \text{Max_Packet_Size}$$

$$p_a = p_b / (1 - \text{count} * p_b)$$

In this algorithm as the congestion increases, more packets are dropped. Larger packets are more likely to be dropped than smaller packets which use less resources.

2.19 Performance

Simulations were run for Tail Drop, Random Drop and RED gateways.

The RED gateway showed higher throughput for smaller buffer sizes than the other algorithms. It was also shown that RED was not as biased against burst traffic as were Tail Drop or Random Drop.

2.20 Discussion

When RED is implemented to drop packets rather than mark them, it handles misbehaving sources well. If a source is using more than its fair share of the bandwidth, then by the probabilistic function, more of its packets will be dropped. However if the gateway marks the packets, it is up to the sources to comply.

3.1 Introduction

The operation of TCP congestion control when the receiver can misbehave, as might occur with a greedy Web client. We first demonstrate that there are simple attacks that allow a misbehaving receiver to drive a standard TCP sender arbitrarily fast, without losing end-to-end reliability. These attacks are widely applicable because they stem from the sender behavior specified in RFC rather than implementation bugs. We then show that it is possible to modify TCP to eliminate this undesirable behavior entirely, without requiring assumptions of any kind about receiver's behavior. This is a strong result: with our solution a receiver can only *reduce* the data transfer rate by misbehaving, thereby eliminating the incentive to do so.

End-to-end congestion control mechanisms, such as those used in TCP, are the primary means used for sharing scarce bandwidth resources in the Internet. These mechanisms implicitly rely on both endpoints to cooperate in determining the proper rate at which to send data. Obviously, if the sending endpoint misbehaves, and does not obey the appropriate congestion control algorithms, then it may send data more quickly than well-behaved hosts – possibly forcing competing traffic to be delayed or discarded. Less obviously, a misbehaving *receiver* can achieve the same result.

While the possibility of such attacks has been hinted at previously

vulnerability and the potential impact are not fully appreciated. We note that the population of receivers is extremely large (all Internet users) and has both the incentive (faster Web surfing) and the opportunity (open source operating systems) to exploit this vulnerability. In this paper, we explore the impact that a misbehaving receiver can have on TCP congestion control. We present two kinds of results. First, we identify several vulnerabilities that can be exploited by a malicious receiver to defeat TCP congestion control. This can be done in a manner that does not break end-to-end reliability semantics and that relies only on the standard behavior of correctly implemented TCP senders. Tests against live Web servers using a modified TCP implementation that we produced for this purpose, we show that it is possible to modify the design of TCP to eliminate this behavior – without requiring that the receiver be trusted in any manner. With our the sender to transmit data at a slower rate than it otherwise would, thus harming only itself. Because our work has serious practical ramifications for an Internet that depends on trust to avoid congestion collapse, we also describe backwards-compatible mechanisms that can be implemented at the sender to mitigate the effects of untrusted receivers. *Ref-no.2*

As far as we are aware, the division of trust between sender and receiver that has not been studied previously in the context of congestion control. While end-to-end congestion control protocols assume that both sender and receiver behave correctly, in many environments the interests of sender and receiver may differ considerably creating significant incentives to violate this “good faith” doctrine. For example, in many wide-area data retrieval applications, such as Web browsing, the sender's interest is to provide

uniform service to all clients requesting data, while the interest of each client receiver is to maximize its own data throughput. TCP's congestion control specification and consequently undermining the fairness and stability provided therein.

The potential congestion resulting from aggressive *senders* has received significant attention from the networking community and has produced proposals for per-flow bandwidth reservation and mechanisms to detect and limit "unfriendly" flows in the network. These solutions, if workable, would solve the more general problem of unconstrained data transmission and would make the issue of trust in end-to-end congestion control less pressing. However, given that it is unlikely that such mechanisms will be widely deployed in the near term, we feel it is still prudent to consider the potential impact of untrusted receivers on the congestion control mechanisms in today's Internet.

Vulnerabilities

By systematically considering sequences of message exchanges, we have been able to identify several vulnerabilities that allow misbehaving receivers to control the sending rate of unmodified, conforming TCP senders. This section describes these vulnerabilities and techniques for exploiting them. In addition to denial-of-service attacks, these techniques can be used to enhance the performance of the attacker's TCP sessions at the expense of behaving clients.

3.2 TCP review *Ref-no.8*

we describe the rudiments of their behavior below to allow those unfamiliar with TCP to understand the vulnerabilities explained later. For simplicity, we consider TCP without the Selective Acknowledgment option (SACK), although the vulnerabilities we describe also exist when SACK is used.

TCP is a connection-oriented, reliable, ordered, byte-stream protocol with explicit flow control. A sending host divides the data stream into individual segments, each of which is no longer than the Sender Maximum Segment Size (SMSS) determined during connection establishment. Each segment is labeled with explicit sequence numbers to guarantee ordering and reliability. When a host receives an in-sequence segment it sends a cumulative acknowledgment (ACK) in return, notifying the sender that all of the data preceding that segment's sequence number has been received and can be retired from the sender's retransmission buffers.

If an out-of-sequence segment is received, then the receiver acknowledges the next contiguous sequence number that was *expected*. If outstanding data is not acknowledged for a period of time, the sender will timeout and retransmit the unacknowledged segments.

TCP uses several algorithms for congestion control, most notably *slow start* and *congestion avoidance*.

Ref-no.9

Each of these algorithms controls the sending rate by manipulating a congestion window (*cwnd*) that limits the number of outstanding unacknowledged bytes that are allowed at any time. When a connection starts, the slow start algorithm is used to quickly increase *cwnd* to reach the bottleneck capacity. When the sender infers that a segment has been lost it interprets this as an implicit signal of network overload and decreases *cwnd* quickly. After roughly approximating the bottleneck capacity, TCP switches to the congestion avoidance algorithm which increases the value of *cwnd* more slowly to probe for additional bandwidth that may become available. We now describe three attacks on this congestion control procedure that exploit a sender's vulnerability to non-conforming receiver behavior.

3.4 ACK division

TCP uses a byte granularity error control protocol and consequently each TCP segment is described by sequence number and acknowledgment fields that refer to byte offsets within a TCP data stream.

However, TCP's congestion control algorithm is implicitly defined in terms of segments rather than bytes. For example, the most recent specification of TCP's congestion control behavior, states:

During slow start, TCP increments *cwnd* by at most SMSS bytes for each ACK received that acknowledges

During congestion avoidance, *cwnd* is incremented by 1 full-sized segment per round-trip time (RTT).

The incongruence between the byte granularity of error control and the segment granularity (or more precisely, SMSS granularity) of congestion control leads to the following vulnerability:

Attack 1:

Upon receiving a data segment containing N bytes, the

receiver divides the resulting acknowledgment into M, where $M \ll N$, separate acknowledgments – each covering one of M distinct pieces of the received data segment.

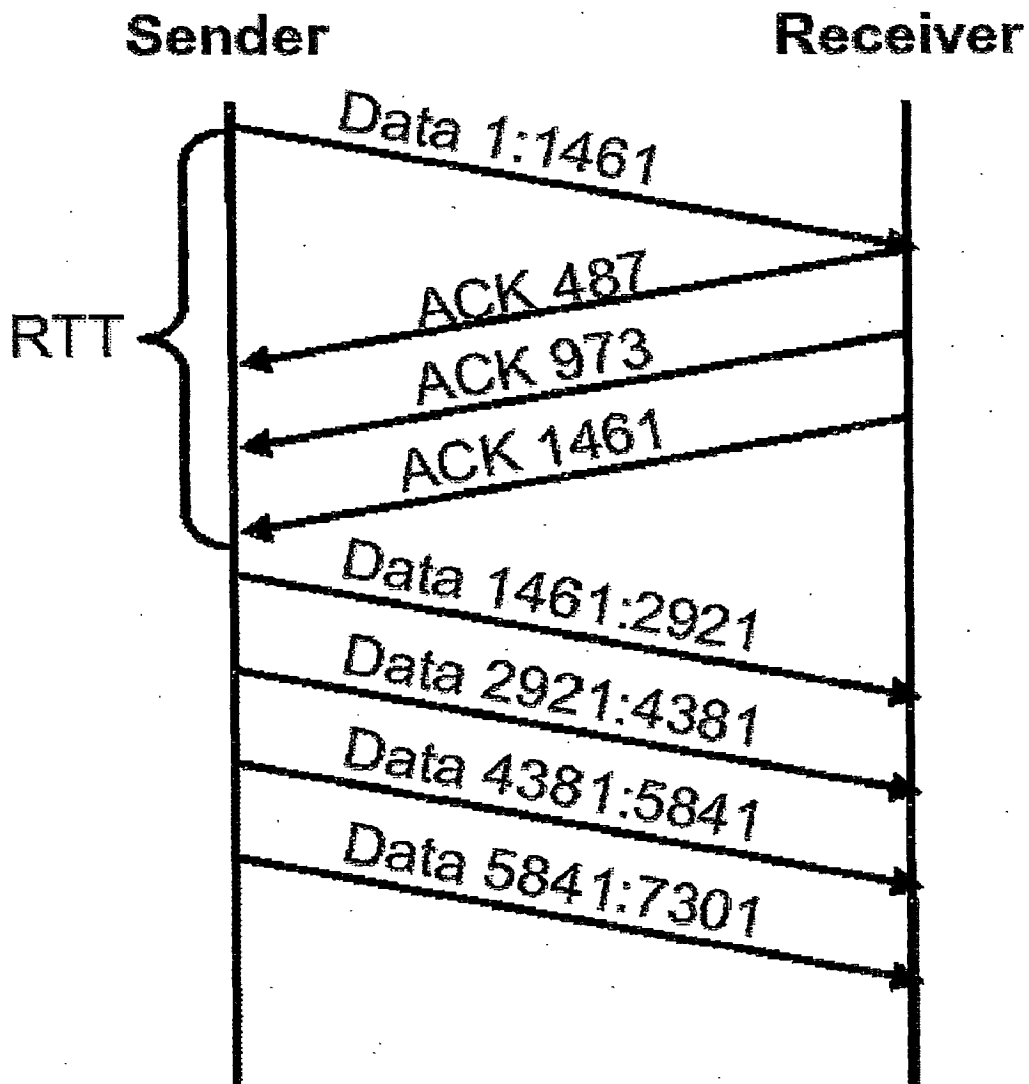


Figure 1: Sample time line for a ACK division attack. The sender begins with $cwnd=1$, which is incremented for each of the three valid ACKs received. After one round-trip time, $cwnd=4$, instead of the expected value of $cwnd=2$.

This attack is demonstrated in Figure 1 with a time line. Here, each message exchanged between sender and receiver is shown as a labeled arrow, with time proceeding down the page. The labels indicate the type of message, data or acknowledgment, and the sequence space consumed. In this example we can see that each acknowledgment is valid, in that it covers data that was sent and previously unacknowledged.

This leads the TCP sender to grow the congestion window at a rate that is M times faster than usual. The receiver can control this rate of growth by dividing the segment at arbitrary points – up to one acknowledgment per byte received (when $M=N$). At this limit, a sender with a 1460 byte MSS could *theoretically* be coerced into reaching a congestion window in excess of the normal TCP sequence space (4GB) in only four roundtrip times! Moreover, while high rates of additional acknowledgment traffic may increase congestion on the path to the sender, the penalty to the receiver is negligible since the cumulative nature of acknowledgments inherently tolerates any losses that may occur.

3.5 DupACK spoofing

Ref-no.10

TCP uses two algorithms, *fast retransmit* and *fast recovery*, to mitigate the effects of packet loss. The fast retransmit algorithm detects loss by observing three duplicate acknowledgments and it immediately retransmits what appears to be the missing segment. However, the receipt of a duplicate ACK also suggests that segments

are leaving the network. The fast recovery algorithm employs this information as follows

Set *cwnd* to *ssthresh* plus $3 * \text{SMSS}$. This artificially “inflates” the congestion window by the number of segments (three) that have left the network and which the receiver has buffered.

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.

1 Of course the practical transmission rate is ultimately limited by other factors such as sender buffering, receiver buffering and network bandwidth.

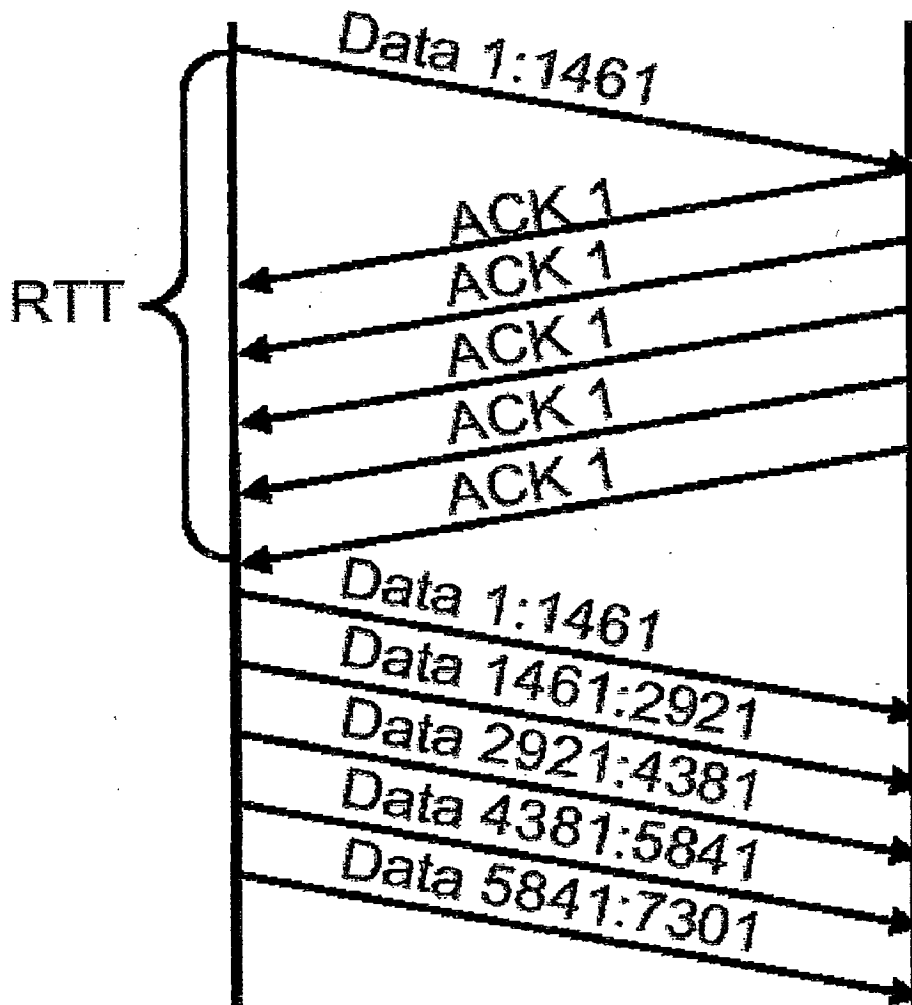


Figure 2: Sample time line for a DupACK spoofing attack. The receiver

forges multiple duplicate ACKs for sequence number 1. This causes the sender to retransmit the first segment and send a new segment for each additional forged duplicate ACK.

There are two problems with this approach. First, it assumes that each segment that has left the network is full sized – again an unfortunate interaction of byte granularity error control and segment granularity congestion control.

Second, and more important, because TCP requires that duplicate ACKs be *exact* duplicates, there is no way to ascertain which data segment they were sent in response to. Consequently, it is impossible to differentiate a “valid” duplicate ACK, from a forged, or “spoofed”, duplicate ACK. For the same reason, the sender cannot distinguish ACKs that are accidentally duplicated by the network itself from those generated by a receiver. In essence, duplicate ACKs are a signal that can be used by the receiver to force the sender to transmit new segments into the network as follows:

Attack 2:

Upon receiving a data segment, the receiver sends a long stream of acknowledgments for the last sequence number received (at the start of a connection this would be for the SYN segment).

Ref-no.12

Figure 2 shows a time line for this technique. The first four ACKs for the same sequence number cause the sender to retransmit the first segment. However, *cwnd* is now set to its initial value plus $3 \cdot \text{SMSS}$, and increased by SMSS for each additional duplicate ACK, for a total of 4 segments (as per the fast recovery algorithm). Since duplicate ACKs are indistinguishable, the receiver does not need to wait for new data to send additional acknowledgments. As a result, the sender will return data at a rate directly proportional to the rate at which the receiver sends acknowledgments. After a

period, the sender will timeout. However, this can easily be avoided if the receiver acknowledges the missing segment and enters fast retransmit again for a new, later, segment.

3.6 Optimistic ACKing

Implicit in TCP's algorithms is the assumption that the time between

a data segment being sent and an acknowledgment for that segment returning is at least one round-trip time. Since TCP's congestion window growth is a function of round-trip time (an exponential function during slow start and a linear function during con-

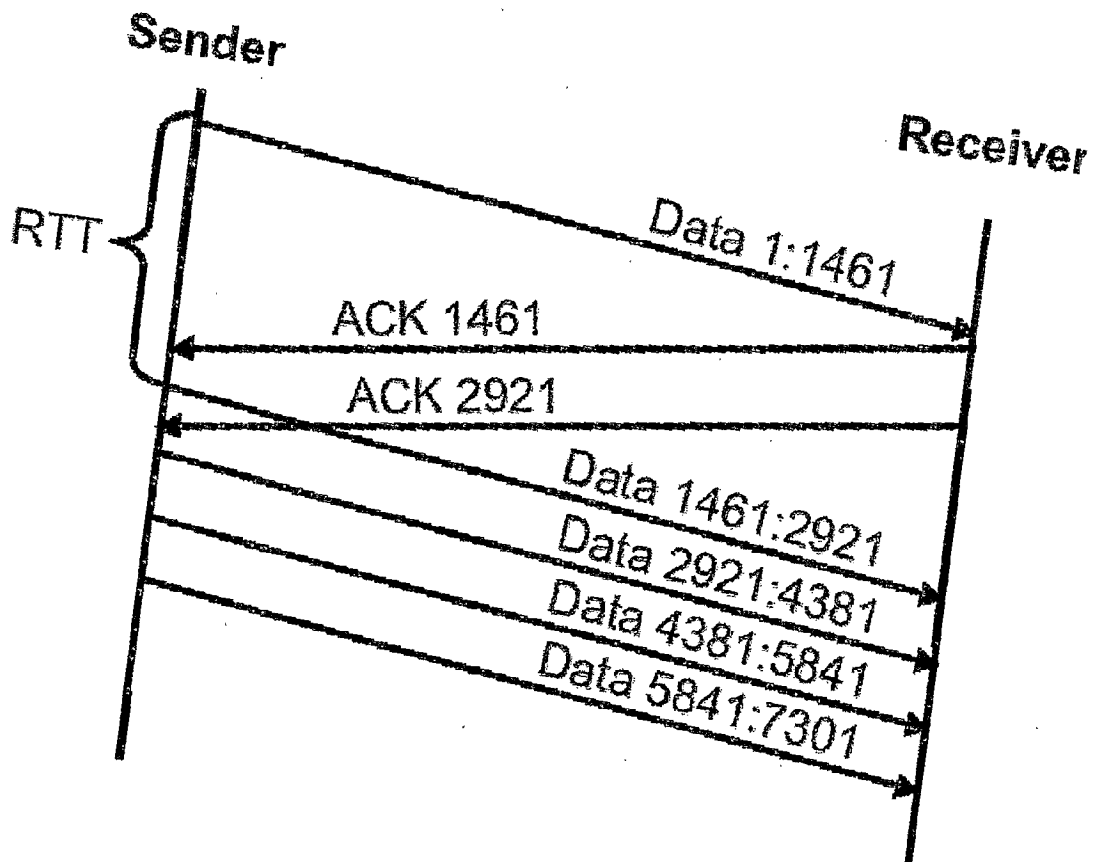


Figure 3: Sample time line for optimistic ACKing attack.
 The ACK for the second segment is sent before the segment itself is received, leading the receiver to grow $cwnd$ more quickly than otherwise. At the end of this example, $cwnd=3$, rather than the expected value of $cwnd=2$. (sender-receiver pairs with shorter round-trip congestion avoidance), sender-receiver pairs with shorter round-trip

times will transfer data more quickly.

However, the protocol does not use any mechanism to enforce its assumption. Consequently, it is possible for a receiver to *emulate* a shorter round-trip time by sending ACKs optimistically for data it has not yet received:

Attack 3:

Upon receiving a data segment, the receiver sends a stream of acknowledgments anticipating data that will be sent by the sender.

Ref-no.12

This technique is demonstrated in Figure 3. Note that while it is easy for the receiver to anticipate the correct sequence numbers to use in each acknowledgment (since senders generally send full-sized segments), this accuracy is not necessary. As long as the receiver acknowledges new data the sender will transmit additional segments. Moreover, if an ACK arrives for data that has not yet been sent, this is generally ignored by the sending TCP – allowing a sender to be arbitrarily aggressive in its generation of optimistic ACKs.

Unlike the previous attacks, this technique does not necessarily preserve end-to-end reliability semantics – if data from the sender is lost it may be unrecoverable since it has already been acknowledged.

However, new features in protocols such as HTTP-1.1 allow receivers to request particular byte-ranges within a data object

. This suggests a strategy in which data is gathered

on one connection and lost segments are then collected selectively with application-layer retransmissions on another. Optimistic

ACKing could be used to ramp the transfer rate up to the

bottleneck rate immediately, and then hold it there by sending acknowledgments in spite of losses. This ability of the receiver to

conceal losses is extremely dangerous because it eliminates the only congestion signal available to the sender. A malicious attacker could conceal *all* losses and therefore lead a sender to increase *cwnd* indefinitely – possibly overwhelming the network with useless packets.

3.7 Designing robust protocols Ref-no.4

We believe TCP's vulnerabilities arise from a combination of unstated assumptions, casual specification and a pragmatic need to develop congestion control mechanisms that are backward compatible with previous TCP implementations. In retrospect, if the contract between sender and receiver had been defined explicitly these vulnerabilities would have been obvious.

Principle 1. Every message should say what it means: the interpretation of the message should depend only on its content.

Principle 2. The conditions for a message to be acted upon should be clearly set out so that someone reviewing a design may see whether they are acceptable or not.

Principle 3. If the identity of a principal is essential to the meaning of a message, it is prudent to mention the principal's name explicitly in the message.

3.8 ACK division

This vulnerability arises from an ambiguity about how ACKs should be interpreted – a violation of the second principle. TCP's error-control allows an ACK to specify an arbitrary byte offset in the sequence space while the congestion control specification assumes that an ACK covers an entire segment.

There are two obvious solutions: either modify the congestion control mechanisms to operate at byte granularity or guarantee that

segment-level granularity is always respected. The first solution is virtually identical to the “byte counting” modifications to TCP discussed in If *cwnd* is not incremented by a full SMSS, but only proportional to the amount of data acknowledged, then ACK division attacks will have no effect. The second, perhaps simpler, solution is to only increment *cwnd* by one SMSS when a valid ACK arrives that covers the entire data segment sent.

3.9 DupACK spoofing

During fast recovery and fast retransmit, TCP's design violates the first principle – the meaning of a duplicate ACK is implicit, dependent on previous context, and consequently difficult to verify.

TCP assumes that all duplicate ACKs are sent in response to unique and distinct segments. This assumption is unenforceable without some mechanism for identifying the data segment that led to the generation of each duplicate ACK. The traditional method for guaranteeing association is to employ a *nonce*. We present a simple version of such a nonce protocol below (we will extend it shortly):

Singular Nonce:

We introduce two new fields into the TCP packet format:

Nonce and Nonce reply. For each segment, the sender fills the Nonce field with a unique random number generated when the segment is sent. When a receiver generates an ACK in response to a data segment, it echoes the nonce value by writing it into the Nonce Reply field.

The sender can then arrange to only inflate *cwnd* in response to duplicate ACKs whose Nonce Reply value corresponds to a data segment previously sent and not yet acknowledged.

We note that the singular nonce, as we have described it so

far, is similar to the Timestamps option with two important differences. First, the Nonce field preserves association for duplicate ACKs, while the Timestamps option does not (preferring instead to reuse the previous timestamp value). Second, and more important, because Timestamps is a *option*, a receiver has the choice to not participate in its use. We cannot rely on misbehaving clients to voluntarily participate in their own policing. For the same reason, we cannot rely on other TCP options, such as proposed extensions to SACK, to eliminate this vulnerability.

ACK Division

Unfortunately, our fix requires the modification of clients and servers and the addition of a TCP field. While it is the only complete solution we have discovered, there are sender-only heuristics which can mitigate, although not eliminate, the impact of the DupACK spoofing attack in a purely backward compatible manner. In particular, the sender can maintain a count of outstanding segments sent above the missing segment. For each duplicate acknowledgment this count is decremented and when it reaches zero any additional duplicate acknowledgments are ignored. This simple fix appears to limit the number of segments wrongly sent to contain no more than $cwnd - MSS$ bytes. Unfortunately, a clever receiver can acknowledge the missing segment and then repeat the process indefinitely unless other heuristics are employed to penalize this behavior (e.g. by refusing to enter fast retransmit multiple times in a single window as suggested).

3.10 Optimistic ACKing

The optimistic ACK attack is possible because ACKs do not contain

any proof regarding the identity of the data segment(s) that caused them to be sent. In the context of the third principle described earlier, a data segment is a principal and an ACK is the message of concern.

This problem is also well addressed using a nonce. If a nonce can't be guessed by the receiver, then ACKs with valid nonces imply that a full round-trip time has taken place (man-in-the-middle attacks notwithstanding).

However, the singular nonce we have described is imperfect because it does not mirror the cumulative nature of TCP. Acknowledgments can be delayed or lost, yet the cumulative property of TCP's sequence numbers ensures that the most recent ACK can cover all previous data. In contrast, the singular nonce only provides evidence that a single segment was received. A misbehaving sender could still mount a denial of service attack by concealing lost data, yet still sending back ACKs with valid nonces.

Ref-no.4

3.13 About Misbehaving receiver...

we have described how a receiver can manipulate the TCP congestion control function managed by the sender, and how the sender can prevent these manipulations. Our work highlights two results that we believe are significant yet not widely appreciated: TCP, which was originally designed for a cooperative environment, contains several vulnerabilities that an unscrupulous receiver can exploit to obtain improved service at the expense of other network clients or to implement a denial-of-service attack.

We have described ACK division, DupACK spoofing and Optimistic ACK mechanisms and implemented them to demonstrate that the attacks are both real and widely applicable.

The design of TCP can be modified, without changing the nature of the congestion control function, to eliminate these vulnerabilities. We have described the workings of a new Cumulative Nonce approach that accomplishes this in a simple yet effective manner. We have also identified and described sender-only modifications that can be deployed immediately to reduce the scope of the vulnerabilities without receiver-side modifications.

Our work can readily be extended to other protocols. While the Cumulative Nonce was defined in the context of TCP, it could be adapted to any sender-based congestion control scheme. This might prove fruitful for unreliable transports, for example, either those that are explicitly TCP-friendly, or other rate adaptive mechanisms, like those employed by RealAudio. A Cumulative Nonce could also be used more widely to aid in the design of other kinds of protocols. This is because it effectively defines a sequencing mechanism between untrusted parties that, because it is lightweight, idempotent and cumulative, is well suited to network environments.

Beyond these immediate results, our work raises more speculative protocol design issues. TCP was originally designed for a cooperative environment, and its evolution through the years has built on this base. Given this, it is perhaps not so surprising that we were able to find the vulnerabilities we did, because they naturally arise when the sender and receiver represent different interests. With the

growth of the Internet, however, it is arguable that “separate interests” should be assumed by default. Protocol functions that are managed by one party would then be designed to minimize the trust they place in other parties. We observe that this kind of “separation of interests” will require new mechanisms, such as a Cumulative Nonce, to guarantee that different parties respect a common behavioral contract.

4.1 Introduction

Congestion refers to a loss of network performance when a network is heavily loaded. Since congestive phenomena can cause data loss, large delays in data transmission, and a large variance in these delays, controlling or avoiding congestion is a critical problem in network management and design. This dissertation presents some approaches for congestion control .

Early research in computer data networking led to the development of reservationless store-and-forward data networks . These networks are prone to congestion since neither the number of users nor their workload are regulated. Essentially, the efficiency gained by statistical multiplexing of network resources is traded off with the possibility of congestion. This problem was recognized quite early , and a number of congestion control schemes were proposed; references provide a detailed review of these.

In the past three years, there has been a renewed interest in congestion control, We feel that at least three factors have been responsible. First, the spread of networks such as ARPANET and their interconnection, has created a very large Internet whose size has made it unmanageable. The large number of users and a complete decentralization of network management made it inevitable that congestion would pose problems sooner or later.

Transmission Control Protocol (TCP) to intelligently react to congestion, and to recover from it . The success of these efforts brought congestion control into focus as a major research area in the Internet community. *Ref-no.4*

The other factor is social, rather than technological. The networking community has long been divided into two camps: the computer data networking community, and the telecommunications community. However, in recent years, the telecom community has realized the

benefits of packet switching, resulting in the Asynchronous Transmission Mode (ATM) . Similarly, data networking researchers have realized that they need to provide realtime bounds on data transfer for services such CD-quality audio and interactive video .

The present time appears to be critical for the design of future high speed networks, and in particular, their congestion control mechanisms. In this dissertation, we propose a number of ideas that we believe are useful for high speed networks. We hope that our work will contribute to the ongoing debate about congestion control.

4.2. Environment of discourse

We survey congestion control techniques in two types of wide-area networks (though the dissertation is limited to techniques suitable for the first type). In both networks, data is sent from *sources* of data to *sinks* through intermediate store-and-forward switching nodes. Sources of data could be human users, transferring characters in a remote login session, or transferring files.

For our purposes, we will refer to processes at OSI layer five and above as data sources. Sinks are the ultimate destinations of the data. They are the peer processes of the sources that receive and consume the received data, and they are typically assumed to acknowledge the receipt of each packet. *Switches* route and schedule incoming packets on outgoing lines, placing data in *output buffers* when the arrival rate exceeds the service rate. The simplex stream of packets between a source of data and its sink is called a *conversation*. Usually, a conversation corresponds to a pair of transport level endpoints, for example, two BSD sockets .

The first type of network under consideration, called a *reservationless network*, is an abstract model for networks such as the Internet. In such a network, while intermediate switches may reserve buffers (which does not reduce statistical multiplexing of the bandwidth), they may not reserve bandwidth (which does). Hosts on reservationless networks are assumed to be connected...

... directly to switches, that in turn connect to other switches or hosts. A switch could be a piece of software that resides in a host, or could be a separate piece of hardware. The other type of networks are those where switches reserve both bandwidth and buffers on behalf of *Virtual Circuits* (VCs) (such as in Datakit). We call these *reservation-oriented networks*. We assume that these networks carry two types of traffic: performance-oriented traffic, which usually needs some form of real-time delay, bandwidth and jitter guarantees, and best-effort data traffic, which does not make such demands .

Since no bandwidth is reserved on behalf of best-effort traffic , the best-effort component of a reservation-oriented network can be modeled as a reservationless network. Hence, schemes that are designed for reservationless networks can be transferred, with appropriate modifications, to reservation oriented networks.

We believe that most future generation networks will tend to be reservation-oriented. Nevertheless, there are still some valid reasons to study congestion control in reservationless networks.

Ref-no.5

First, reservationless networks will always be able to use bandwidth more efficiently than reservation-oriented networks due to the gain from statistical multiplexing. So, network providers who want to optimize cost will continue to build reservationless networks. Second, the techniques that are developed for congestion control can be applied to control best-effort traffic in reservation-oriented networks. Thus, the results of this work will apply even in those networks. Third, reservationless networks are currently the most common type of computer network. We believe that because of inertia, and a desire to stay with known and proven technology, they will continue to exist in the future.

4.3 What is congestion?

we are not aware of a satisfactory definition of congestion. We now discuss some common definitions, point out their flaws, and then propose a new definition that we consider to be superior.

Some common definitions of congestion

Since congestion occurs at high network loads, definitions of congestion focus on some aspect of network behavior under high load. We first discuss a scenario that leads to network congestion in reservationless networks, and then motivate some definitions.

Consider a reservationless network, where, due to some reason, the short term packet arrival rate at some switch exceeds its service rate. (The service rate is determined by the processing time per packet and the bandwidth of the output line. Thus, the bottleneck could be either the switch's CPU or the outgoing line: in either case, there is congestion.) At this point, packets are buffered, leading to delays. The additional delay can cause sources to time out and retransmit, increasing the load on the bottleneck. This feedback leads to a rapidly deteriorating situation where retransmissions dominate the traffic, and effective throughput rapidly diminishes. Further, if there is switch to switch flow control (as in ARPANET etc.), new packets may not be allowed to enter the switch, and so packets might be delayed at a preceding switch as well. This can lead to deadlock, where all traffic comes to a standstill.

Note that three things happen simultaneously. First, the queueing delay of the data packets increases. Second, there may be packet losses. Finally, in the congested state, the traffic is dominated by retransmissions, so that the effective data rate *decreases*. The standard definitions of congestion are thus of the form: **“A network is congested if, due to overload, condition X occurs”**, where **X is excessive queueing delay, packet loss or decrease in effective throughput**. The first definition is used in references , the second in reference , and the third in reference .

These definitions are not satisfactory for several reasons. First, delays and losses are indices of performance that are being improperly used as indices of congestion, since the change in the indices may be due to symptoms of phenomena other than congestion.

Second, the definitions do not specify the exact point at which the network can be said to be congested (except in a deterministic network, where the knee of the load-delay curve, and hence congestion, is well defined, but that is the trivial case). For example, while a network that has mean queueing delays in each switch of the order of 1 to 10 service times is certainly not congested, it is not clear whether a network that has a queueing delay of 1000 service times is congested or not. It does not seem possible to come up with any reasonable threshold value to determine congestion!

Third, a network that is congested from the perspective of one user is not necessarily congested from the perspective of another. For example, if user A can tolerate a packet loss rate of 1 in 1000, and user B can tolerate a packet loss rate of 1 in 100, and the actual loss rate is 1 in 500, then A will claim that the network is congested, whereas B will not. A network should be called uncongested only if all the users agree that it is

Ref-no.5

4.4 New definition

From the discussion above, it is clear that network congestion depends on a user's perspective.

A user who demands little from the network can tolerate a loss in performance much better than a more demanding user. For example, a user who uses a network only to send and receive electronic mail will be happy with a delivery delay of a day, while this performance is unacceptable for a user who uses a network for real-time audio communication. The key point is the notion of the *utility* that a user gets from the network, and how this utility degrades with network loading.

The concept of 'utility' used here is borrowed from economic theory. It is used to refer to a user's preference for a resource, or a set of resources (often called a resource bundle).

Strictly speaking, the utility of a user is a number that represents the relative preference of that user for a resource (or performance) bundle, so that, if a user prefers bundle A to bundle B, the utility of A is greater than the utility of B. For example, if A is {end-to-end delay of 1 second, average throughput 200 pkts/second}, and B is {end-to-end delay of 100 seconds, average throughput 20000 pkts/second}, a user may prefer A to B, and we would assign a utility to A that is greater than the utility of B, while another user may do the opposite. In classic microeconomic theory,

utilities are represented by a function over the resources. Since utilities express only a preference ordering, utility functions are insensitive to monotonic translations, and the utilities of two users cannot be compared; the function can only be used to relatively rank two resource bundles from the point of view of a single user. An example of a utility function is $\alpha T - (1 - \alpha)RTT$, where α is a weighting constant, T is the average throughput over some interval, and RTT is the average round-trip-time delay over the same interval. As the throughput increases, the utility increases, and as delays increase, the utility decreases. The choice of α determines the relative weight a user gives to throughput and delay.

A delay-sensitive user will choose $\alpha \rightarrow 0$, whereas a delay-insensitive user's $\alpha \rightarrow 1$.

In practice, a utility function may depend on a threshold. For example, a user may state that he or she is indifferent to delay, as long as it is less than 0.1 seconds. Thus, if the user gets a delay of 0.05 seconds during some interval of time, and 0.06 seconds in a later period, as far as the user is concerned, there has been no loss of utility. However, if some user's utility *does* to be congested.

Definition

A network is said to be congested from the perspective of user i if the utility of i decreases due to an increase in network load.

Remarks:

1. A network can be congested from the perspective of one user, and uncongested from the perspective of another.
2. A network can be said to be strictly uncongested if no user perceives it to be congested.

3. A user's utility may decrease due to something other than network load, but the user may not be able to tell the difference. The onus on the user is to determine the cause of the loss of utility, and to take appropriate corrective action. This definition is better than existing definitions since it avoids the three problems raised earlier.

First, we make a clear distinction between a performance index and a congestion index. It is possible for a performance metric to decrease (for example, for *RTT* to increase), without a change in the congestion index. Second, the definition makes it clear that congestion occurs from the point of view of each individual user. Finally, the point of congestion is precisely the one where the user detects a loss of utility. No further precision is necessary, since, if the users are not dissatisfied with the available service, then the network performance, no matter how poor it is in absolute terms, is satisfactory. Our definition places congestion control in a new light. A network that controls congestion, by our definition, must be responsive to the utility function of the users, and must be able to manage its resources so that there is no loss of utility as the load increases. Thus, the network *must* be able to differentiate between conversations, and prioritize conversations depending on the stringency of their owner's utility. A naive approach that ignores the user's quality-of-service requirements is automatically ruled out by this definition.

4.5 Congestion control

The previous section presented a new definition of congestion; this section describes congestion control. Two styles of control, proactive and reactive control, are presented. It is shown that congestion control must happen at several different time scales.

4.5.1. Proactive and reactive control

Congestion is the loss of utility to a user due to an increase in the network load. Hence, congestion control is defined to be the set of mechanisms that prevent or reduce such a deterioration. Practically speaking, a network can be said to control congestion if it provides each user with mechanisms to specify and obtain utility

from the network. For example, if some user desires low queueing delays, then the system should provide a mechanism that allows the user to achieve this objective. If the network is unable to prevent a loss of utility to a user, then it should try to limit the loss to the extent possible, and, further, it should try to be fair to all the affected parties. Thus, in reservationless networks, where a loss of utility at high loads is unavoidable, we are concerned not only with the extent to which utility is lost, but also the degree to which the loss of utility is fairly distributed to the affected users.

A network can provide utility in one of two ways. First, it can request that each user specify a performance requirement, and can reserve resources so that this level of performance is always available to the user. This is proactive or reservation-oriented congestion control. Alternatively, users can be allowed to send data without reserving resources, but with the possibility that, if the network is heavily loaded, they may receive low utility from the network. The second method is applicable in reservationless networks. In this case, users must adapt to changes in the network state, and congestion control refers to ways in which a network can allow users to detect changes in network state, and corresponding mechanisms that adapt the user's flow to changes in this state.

In a strict proactive scheme, the congestion control mechanism is to make reservations of network resources so that resource availability is deterministically guaranteed to admitted conversations. In a reactive scheme, the owners of conversations need to monitor and react to changes in network state to avert congestion. Both styles of control have their advantages and disadvantages. With proactive control, users can be guaranteed that they will never experience loss of utility. On the other hand, to be able to make this guarantee, the number of users has to be restricted, and this could lead to underutilization of the network. Reactive control allows much more flexibility in the allocation of resources. Since users are typically not guaranteed a level of utility by the network, resources can be statistically multiplexed. However, there is always a chance that correlated traffic bursts will overload the network, causing performance degradation, and hence, congestion. It is important to realize that proactive and reactive control are not mutually exclusive.

Hybrid schemes can combine aspects of both approaches. One such hybrid scheme is for the network to provide statistical guarantees . For example, a user could be guaranteed an end to end delay of less than 10 seconds with 0.9 probability. Such statistical guarantees allow a network administrator to overbook resources in a controlled manner. Thus, statistical multiplexing gains are achieved, but without completely giving up performance guarantees.

Another hybrid scheme is for the network to support two types of users: guaranteed service users and best-effort users . Guaranteed service (GS) users are given a guarantee of quality of service, and resources are reserved for them. Best-effort (BE) users are not given guarantees and they use up whatever resources are left unutilized by GS users. Finally, a server may reserve some minimum amount of resources for each user. Since every user has some reservation, some minimum utility is guaranteed. At times of heavy load, users compete for resources kept in a common pool . Assuming some degree of independence of traffic, statistical multiplexing can be achieved without the possibility of a complete loss of utility.

4.5.2. Time scales of control

Refno.11

Congestion is a high-load phenomenon. The key to congestion control lies in determining the time scale over which the network is overloaded, and taking control actions on that time scale. This is explained below. Consider the average load on a single point-to-point link. Note that the ‘average load’ is an *interval-based metric*. In other words, it is meaningless without also specifying the time interval over which the average is measured. If the average load is high over a small averaging interval, then the congestion control mechanism (for example, the reservation mechanism) has to deal with resource scheduling over the same small time scale. If the average load is high over a longer time scale, the congestion control mechanism needs to deal with the situation over the longer time scale *as well as* on shorter time scales.

An example should clarify this point. Consider a conversation on a unit capacity link. If

the conversation is bursty, then it could generate a high load over, say, a 1ms time scale, though the average load over a 1 hour time scale could be much smaller than 1. In this case, if the conversation is delay-sensitive, then the congestion control scheme must take steps to satisfy the user delay requirement on the 1ms time scale. Over longer time scales, since the average demand is small, there is no need for congestion control.

On the other hand, if the conversation has a high average demand on the 1 hour time scale as well as the 1ms time scale, then congestion control has to be active on both time scales. For example, it may do admission control (which works over the 1 hour time scale) to make sure that network resources are available for the conversation.

Simultaneously, it may also make scheduling decisions (which work on the 1ms time scale) to meet the delay requirements. This example illustrates three points. First, congestion control must act on several different time scales simultaneously. Second, the mechanisms at each level must cooperate with each other. Scheduling policies without admission control will not ensure delay guarantees. At the same time, the admission control policy must be aware of the nature of the scheduling policy to decide whether or not to admit a conversation into the network. Third, the time scale is the time period over which a user sees changes in the network state. A congestion control mechanism that is sensitive to network state must operate on the same time scale.

We now discuss five time scales of control: those of months, one day, one session, multiple round trip times (RTTs), and less than one RTT. We believe that the design of congestion control mechanisms for each time scale should be based on sound theoretical arguments. This has the obvious advantages over an *ad hoc* approach: general applicability, ease of understanding, and formal provability of correctness. At each time scale of control, a different theoretical basis is most appropriate, and this is discussed below.

4.5.3 Session

In connection-oriented networks, a session is the period of time between a call set-up and a call teardown. Admission control in connection oriented-networks is essentially congestion control on the time scale of a session: if the admission of a new conversation could degrade the quality of service of other conversations in the network, then the new conversation should not be admitted. This is yet another form of congestion control. At the time a conversation is admitted to the network, the network should ensure that the resources requested by the conversation are what it really needs. Thus, the admission control scheme should give incentives to users to declare their resource needs accurately .

4.5.4 Multiple round trip times

One round trip time (RTT) is the fundamental time constant for feedback flow control. It is the minimum time that is needed for a source of data to determine the effect of its sending rate on the network . Congestion control schemes that probe network state and do some kind of filtering on the probes operate on this time scale. Examples are various window adjustment schemes . The theoretical bases for these approaches lie in queueing theory and control theory. The queueing theory approach is well studied , but requires strong assumptions about the network. such as: Poisson arrivals from all sources, exponential service time distribution at all servers, and independence of traffic. Since observations of real networks have shown that none of these assumptions are satisfied in practice about traffic behavior, service rates and so on, that do not hold in practice .

Assuming that each packet is acknowledged, multiple acknowledgements can be received each RTT. If information about the state of the network is extracted from each acknowledgement,

4.5.6 Less than one RTT

On a scale of less than once per RTT, congestion control can be considered to be

identical to scheduling data at the output queues of switches. The goal of a scheduling policy is to decide which data unit is the next to be delivered on a trunk. This choice determines the bandwidth, delay, and jitter received by each conversation, and hence the choice of the scheduling discipline is critical. A scheduling discipline that does *not* vary its allocations as a function of the network load is hence a congestion control mechanism. Examples are the Virtual Clock scheme and Stop-and-Go queueing

4.5.7 Need for congestion control in future networks

Congestion is a severe problem in current reservationless networks. However, in future networks the available bandwidths and switching speeds will be several orders of magnitude larger. Why should congestion arise in such networks? There are several reasons:

Speed Mismatch : If a switch connects a high speed line to a slower line, then a bursty conversation can, when sending data at the peak rate, fill up its buffer share, and subsequently lose packets at the switch. This creates congestion for loss-sensitive conversations.

This source of congestion will persist in high-speed networks, in fact, it is probably more likely in such networks.

Topology : If several input lines simultaneously send data through a switch to a single outgoing line, the outgoing line can be overloaded, leading to large queueing delays, and possible congestion for delay-sensitive traffic. This is a special case of the speed mismatch problem noted earlier.

Increased Usage : Memory sizes have increased exponentially during the last decade. Yet, the demand for memory has remained, since larger memory sizes have made it feasible to develop applications that require them. Drawing a parallel to this trend, we postulate that as bandwidth increases, new applications (such as real time video) will

demand these enormous bandwidths. As the available bandwidth gets saturated, the network will be operated in the high-load zone, and congestive problems are likely to reappear.

Misbehavior : Congestion can be induced by misbehaving sources (such as broken sources that send a stream of back-to-back packets). Future networks must protect themselves and other sources from such misbehavior, which will continue to exist.

Dynamics : As network speeds increase, the dynamics of the network also changes. Since queues can build up faster, congestive phenomena can be expected to occur much more rapidly, and perhaps have catastrophic effects . From these observations, we conclude that even though future networks will have larger trunk bandwidths and faster switches, congestion will not disappear.

4.6. Fundamental assumptions

Underlying any congestion control scheme are some implicit assumptions about the network environment. These unstated assumptions largely determine the nature of the control scheme and its performance limits. We consider some of these assumptions in this section.

4.6.1. Administrative control

Can we, as designers of congestion control mechanisms, assume administrative control over the behavior of sources Or, can we assume administrative control only over the behavior of switches? Some schemes assume that we can control sources but not switches e.g. Others assume that we can control both the sources and the switches . Still others assume complete control over the switches, and the ability to monitor source traffic, but no control over source traffic .

This assumption should be constrained by reality. In our work, we assume that we have administrative control over switches. However, source behavior is assumed to be outside our direct control (though it can be monitored, if needed). The implication is that the network must take steps to protect itself and others from malicious or misbehaving users. (Of course, users that abuse the network because of a hardware failure, such as a jammed ethernet controller, are always a threat, even when the sources can be controlled administratively.)

4.6.2. Source complexity

Ref-no.7

How complex should one assume sources to be? Since we do not have administrative control over sources, we assume that sources will perform actions that will maximize their own utility from the network. If the congestion control scheme allows intelligent users to manipulate the scheme for their own benefit, they will do so. On the other hand, users may not have the capability to respond to complex directives from the network, so we cannot assume that all users will act intelligently. In other words, while a congestion control scheme should not assume sophisticated behavior on the part of the users, at the same time, it should not be open to attack from such users.

4.6.3. Gateway complexity

Some authors assume that switches can be made complex enough to set bits on packet headers, or even determine user utility functions, whereas others assume that switches simply route packets. Ignoring monetary considerations, since we have administrative control, we can make switch control algorithms as complex as we wish, constrained only by speed requirements.

It has been claimed that for high speed operation, switches should be dumb and fast. We believe that speed does not preclude complexity. What we need is a switch that is fast *and* intelligent. This can be achieved by

1. having hardware support for rapid switching
2. optimizing for the average case
3. removing signaling information from the data path

4. choice of scheduling algorithm
5. an efficient call processing architecture .

Thus, we will assume that a switch can make fairly intelligent decisions, provided that this can be done at high speeds.

4.6.4. Bargaining power

The ultimate authority in a computer network lies in the ability to drop packets (or delay them). Since this authority lies with the switches, they ultimately have all the bargaining power. In other words, they can always coerce sources to do what they want them to do (unless this is so ridiculous that a source would rather not send any data). Any scheme that overlooks this fact loses a useful mechanism to control source behavior. Thus, schemes that treat switches and sources as peer entities are fundamentally flawed: they need to posit cooperative sources precisely because they ignore the authority that is automatically vested in switches.

4.6.5. Responsibility for congestion control *Refno.11*

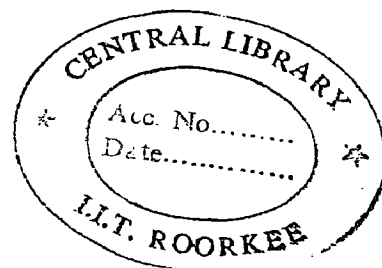
Either the sources or the switches could be made responsible for congestion control. If sources are responsible, they must detect congestion and avert it. If switches are responsible, they must take steps to ensure that sources reduce their traffic when congestion occurs, or allocate resources to avert congestion.

We believe that congestion control is a network function. If we leave the responsibility for it to sources that are not under our administrative control, then we are endangering the network.

Further, the congestion detection and management functionality has to be duplicated at each of the (many) sources. In contrast, it is natural to make the fewer, controllable switches responsible for congestion control. Note that responsibility is not the same as functionality. In other words, having responsibility for congestion does not mean that the switches have to actually perform all the actions necessary for congestion control switches can enforce rules that make it incentive compatible for Sources to help in containing congestion. For example, a Fair Queueing switch has the responsibility for

511007

(50.)



congestion control, but it does congestion control by forcing *sources* to behave correctly during congestion.

4.6.7. Traffic model

Ref-no.1

The choice of a traffic model influences the design of a congestion control scheme, since the scheme is evaluated with respect to this model. There are hidden dangers here: for example, schemes that assume Poisson sources may not be robust, if, in practice, traffic does not obey this distribution. It is best to design schemes that are insensitive to the choice of the traffic model. This is achieved if a scheme does not make assumptions about the arrival distribution of packets at the switches.

What should be the traffic model? We do not have much data at our disposal, since there are no high-speed WANs yet available to measure. However, there are three trends that point to a reasonable model. First, the move towards integration of data, telephony and video services indicates that some number of sources in our environment will be phone and video sources.

These can generate high bandwidth traffic over periods of time spanning minutes or hours. Second, existing studies have shown that data traffic is very bursty. This tendency will certainly be exaggerated by increases in line speeds. Finally, we note that current applications are mostly of two sorts - low bandwidth interactive conversations, and high bandwidth offline bulk data transfer. At higher speeds, the bulk data transfers that last several seconds today will collapse into bursts. This reinforces our belief that future traffic will basically be bursty.

To sum up, we expect that the traffic will be generated by two kinds of sources: one that demands a sustained high bandwidth, and the other that generates bursts of traffic at random intervals of time. We call these sources 'FTP' and 'Telnet' in this thesis (these terms are probably outdated, but we use them for convenience). This model is fairly simple, and does not involve any assumptions about packet arrival distributions.

Thus, schemes that work for this model will probably work for a large variety of parametrically constrained models as well (e.g. for traffic where the bursts are exponentially or uniformly distributed). Since the traffic characteristics for future networks are still unknown, this model is speculative. However, we think that it is as reasonable as any that have so far been studied.

4.7.1. Reservationless networks

In reservationless networks, control has to be reactive. A reactive congestion control scheme is implemented at two locations: at the switches, where congestion occurs, and at the sources, which control the net inflow of packets into the network. Typically, a switch uses some metric (such as overflow of buffers) to determine the onset of congestion, and implicitly or explicitly communicates this problem to the sources, which reduce their input traffic.

4.7.2 Congestion detection

How is a switch or source to detect congestion? There are several alternatives. The most common one is to notice that the output buffers at a switch are full, and there is no space for incoming packets. If the switch wishes to avoid packet loss, congestion avoidance steps can be taken when some fraction of the buffers are full. A time average of buffer occupancy can help smooth transient spikes in queue occupancy. A switch may monitor output line usage. It has been found that congestion occurs when trunk usage goes over a threshold (typically 90%) and so this metric can be used as a signal of impending congestion. The problem with this metric is that congestion avoidance could keep the output line underutilized, leading to possible inefficiency.

4.7.3 Communication

Communication of congestion information from the congested switch to a source can be implicit or explicit. When communication is explicit, the switch sends information in packet headers or in control packets such as Source Quench packets, choke packets, state-exchange packets, rate-control messages, or throttle packets to the source. Implicit communication occurs when a source uses probe values, retransmission timers, throughput monitoring, or delay monitoring to indicate the (sometimes only

suspected) occurrence of congestion.

Explicit communication imposes an extra burden on the network, since the network needs to transmit more packets than usual, and this may lead to a loss in efficiency. On the other hand, with implicit communication, a source may not be able to distinguish between congestion and other performance problems, such as a hardware problem. Thus, the communication channel is quite noisy, and a cause of potential instability.

4.7.5 Flow control

A number of congestion control schemes have been proposed that operate at the sources. These schemes use the loss of a packet (or the receipt of choke information) to reduce the source sending rate in some way. The two main types of schemes are choke schemes and rate-control schemes.

In a choke scheme, a source shuts down when it detects congestion. After some time, the source is allowed to start up again. Choking is not efficient, since the reaction of the sources is too abrupt. The stability of the choke scheme has not been analyzed, but the simple

In a rate-control scheme, when a source detects congestion it reduces the rate at which it sends out packets, either using a window adjustment scheme or a rate adjustment scheme. The latter is particularly suitable for sources that do rate based flow control. The advantage of rate control schemes over choke schemes is that rate control allows a gradual transition between sending no packets at all to sending full blast. Rate control seems to be an attractive

4.7.7 Reservation-oriented networks *Ref-no.1*

In reservation-oriented networks, network resources can be allocated at the start of each session. Then, the network can guarantee a performance level to a conversation by performing admission control. This can guarantee congestion control, but perhaps at the cost of underutilization of network resources.

Here, the network places a limit on the size of the flow control window of each conversation, and the connection establishment packet reserves a full window's worth of buffer space at each intermediate switch. Thus, every virtual circuit, once established, is guaranteed to find enough buffers for each outstanding packet, and packet loss is avoided.

The complementary scheme is to reserve bandwidth instead of buffers. This is the approach taken by Zhang in the Flow Network, by Ferrari *et al* in their real-time channel establishment scheme. In a hybrid scheme described in reference a source makes a reservation for buffers at the beginning of a call, and a reservation for bandwidth before the start of each burst. This allows bandwidth to be efficiently shared, but each burst experiences a round trip time delay.

There are four major problems with any naive reservation scheme: scaling, queueing delay, underutilization and enforcement.

4.7.9 Delay

In a proactive scheme, at overload, a full window could be buffered at the bottleneck. When this happens, the queueing delay could be unacceptable. Delays can be bounded by computing the worst-case delay at the time of call set-up, and doing admission control.

4.7.10 Underutilization

Ref-no. 1

The major problem with reservations is that the network could be underutilized: an overzealous admission control scheme could prevent congestion by allowing only a few conversations to enter. This is not acceptable. The crux of the problem lies in determining how many conversations can be admitted into the network without reducing the performance guarantees made to the existing conversations. This has been studied by Ferrari *et al*. One solution to the problem is to define statistical guarantees, where some degree of performance loss can be tolerated.

An efficient way to monitor, (if necessary), reshape user traffic behavior to make it less bursty, is the leaky-bucket scheme.

Other schemes do enforcement at each switch. This is through some form of round-robin like queueing discipline at each switch. In the Flow network, the Virtual Clock mechanism is used .

4.7.12 Quality of service

One can view congestion control as being able to guarantee quality of service at high loads. There has been some previous work in guaranteeing quality of service in networks. Postel made an early suggestion for reservationless networks, though this was not studied in any depth . Stop-and-go queueing provides conversations with bandwidth, delay , and is similar in spirit to Hierarchical Round Robin scheme .

/*
*/

CONCLUSION

Computer networks have an explosive growth over the past few years and with that growth have come severe congestion problems. For example, it is now common to see internet gateways drop 10% of the incoming packets because of local buffer overflows. Our investigation of some of these problems has shown that much of the cause lies in transport protocol implementations (*not* in the protocols themselves): The 'obvious' ways to implement a window-based transport protocol can result in exactly the wrong behavior in response to network congestion.

The Equation-based congestion control for unicast traffic. Most best-effort traffic in the current Internet is well-served by the dominant transport protocol, TCP. However, traffic such as best-effort unicast streaming multimedia could find use for a TCP-friendly congestion control mechanism that refrains from reducing the sending rate in half in response to a single packet drop. With our mechanism, the sender explicitly adjusts its sending rate as a function of the measured rate of loss events, where a *loss event* consists of one or more packets dropped within a single round-trip time (RTT). We use both simulations and experiments over the Internet to explore performance.

Since congestion occurs at high network loads, congestion focus on some aspect of network behavior under high load. A scenario that leads to network congestion in reservationless networks,

Consider a reservationless network, where, due to some reason, the short term packet arrival rate at some switch exceeds its service rate. (The service rate is determined by the processing time per packet and the bandwidth of the output line. Thus, the bottleneck could be either the switch's CPU or the outgoing line: in either case, there is congestion.) At this point, packets are buffered, leading to delays. The additional delay can cause sources to time out and retransmit, increasing the load on the bottleneck. This feedback leads to a rapidly deteriorating situation where retransmissions dominate the traffic, and

effective throughput rapidly diminishes. Further, if there is switch to switch flow control (as in ARPANET etc.), new packets may not be allowed to enter the switch, and so packets might be delayed at a preceding switch as well. This can lead to deadlock, where all traffic comes to a standstill. Note that three things happen simultaneously. First, the queuing delay of the data packets increases. Second, there may be packet losses. Finally, in the congested state, the traffic is dominated by retransmissions, so that the effective data rate decreases. The potential congestion resulting from aggressive *senders* has received significant attention from the networking community and has produced proposals for per-flow bandwidth reservation and mechanisms to detect and limit “unfriendly” flows in the network. These solutions, if workable, would solve the more general problem of unconstrained data transmission and would make the issue of trust in end-to-end congestion control.

REFERENCES

- [1] EDGE, S. W. An adaptive timeout algorithm for retransmission across a packet switching network. In *Proceedings of SIGCOMM '99* (Mar. 1999), ACM.
- [2] FELLER, W. *Probability Theory and its Applications*, second ed., vol. II. John Wiley & Sons, 1999.
- [3] HAJEK, B. Stochastic approximation methods for decentralized control of multiaccess communications. *IEEE Transactions on Information Theory IT-31*, 2 (Mar. 1996).
- [4] HAJEK, B., AND VAN LOON, T. Decentralized dynamic control of a multiaccess broadcast channel. *IEEE Transactions on Automatic Control AC-27*, 3 (June 2000).
- [5] *Proceedings of the Sixth Internet Engineering Task Force* (Boston, MA, Apr. 1998). Proceedings available as NIC document IETF-87/2P from DDN Network Information Center, SRI International, Menlo Park, CA.
- [6] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO International Standard 8473, Information Processing Systems — Open Systems Interconnection — Connectionless-mode Network Service Protocol Specification*, Mar. 1999
- [7] JACOBSON, V. Congestion avoidance and control. In *Proceedings of SIGCOMM '88* (Stanford, CA, Aug. 1999), ACM.
- [8] JAIN, R. Divergence of timeout algorithms for packet retransmissions. In *Proceedings Fifth Annual International Phoenix Conference on Computers and Communications* (Scottsdale, AZ, Mar. 1996).
- [9] JAIN, R. A timeout-based congestion control scheme for window flow-controlled networks. *IEEE Journal on Selected Areas in Communications SAC-4*, 7 (Oct. 1986).

- [10] JAIN, R., RAMAKRISHNAN, K., AND CHIU, D.-M. Congestion avoidance in computer networks with a connectionless network layer. Tech. Rep. DEC-TR-506, Digital Equipment Corporation, Aug. 1998.
- [11] KARN, P., AND PARTRIDGE, C. Estimating round-trip times in reliable transport protocols. In *Proceedings of SIGCOMM '87* (Aug. 1999), ACM.
- [12] KELLY, F. P. Stochastic models of computer communication systems. *Journal of the Royal Statistical Society B* 47, 3 (1999), 379–395.
- [14] KLEINROCK, L. *Queueing Systems*, vol. II. John Wiley & Sons, 1976.
- [15] KLINE, C. Supercomputers on the Internet: A case study. In *Proceedings of SIGCOMM '87* (Aug. 2000), ACM.
- [16] LJUNG, L., AND SÖDERSTRÖM, T. *Theory and Practice of Recursive Identification*. MIT Press, 1983.
- [17] LUENBERGER, D. G. *Introduction to Dynamic Systems*. John Wiley & Sons, 2001.
- [18] MILLS, D. *Internet Delay Experiments*. ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, Dec. 1988. RFC
- [19] NAGLE, J. *Congestion Control in IP/TCP Internetworks*. ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, Jan. 1984. RFC-896.
- [20] PRUE, W., AND POSTEL, J. *Something A Host Could Do with Source Quench*. ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, July 1987. RFC-1016.
- [21] ROMKEY, J. *A Nonstandard for Transmission of IP Datagrams Over Serial Lines: Slip*. ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, June 1988. RFC-1055.
- [22] ZHANG, L. Why TCP timers don't work well. In *Proceedings of SIGCOMM '86* (Aug. 1986), ACM.

APPENDIX

Programming Part

Language Used : Visual Basic 6.0

```
Dim i As Integer, a As Integer, b As Integer, c As Integer, d As Integer, e As Integer
```

```
Private Sub Command1_Click() // integer declaration and click interface
```

```
t.Enabled = True
```

```
t1.Enabled = True
```

```
End Sub
```

```
Private Sub Command2_Click()
```

```
t2.Enabled = True
```

```
t1.Enabled = True
```

```
End Sub
```

```
Private Sub Command3_Click()
```

```
t3.Enabled = True
```

```
t1.Enabled = True
```

```
End Sub
```

```
Private Sub Command4_Click()
```

```
t4.Enabled = True
```

```
t1.Enabled = True
```

```
End Sub
```

```
Private Sub Command5_Click()
```

```
t5.Enabled = True
```

```
t1.Enabled = True
```

```
End Sub
```

```
Private Sub Command6_Click()
```

```
t7.Enabled = True           // Pixel arrangement.  
End Sub
```

```
Private Sub Form_Load()
```

```
End Sub
```

```
Private Sub t_Timer()
```

```
  If i <= 3 Then
```

```
    If l1(0).Left < 4600 Then
```

```
      l1(0).Visible = True
```

```
      l1(0).Left = l1(0).Left + 100
```

```
    End If
```

```
  Else
```

```
    t6.Enabled = True
```

```
  End If
```

```
End Sub
```

```
Private Sub t1_Timer()
```

```
  Text1 = i
```

```
  If a = 0 Then
```

```
    If l1(0).Left > 4560 Then
```

```
      i = i + 1
```

```
      a = a + 1
```

```
    End If
```

```
  End If
```

```
  If b = 0 Then
```

```
    If l1(1).Left > 4560 Then
```

```
      i = i + 1
```

```
      b = 1
```

```
    End If
```

```
  End If
```

```
If c = 0 Then
If l1(2).Left > 4560 Then
i = i + 1
c = 1
End If
End If
```

```
If d = 0 Then
If l1(3).Left > 4560 Then
i = i + 1
d = 1
End If
End If
```

```
If e = 0 Then
If l1(4).Left > 4560 Then
i = i + 1
e = 1
End If
End If
```

```
End Sub
```

```
Private Sub t2_Timer()
If i <= 3 Then
If l1(1).Left < 4600 Then
l1(1).Visible = True
l1(1).Left = l1(1).Left + 100
End If
Else
t6.Enabled = True
End If
End Sub
Private Sub t3_Timer()
```

```
If i <= 3 Then
If l1(2).Left < 4600 Then
l1(2).Visible = True
l1(2).Left = l1(2).Left + 100
End If
Else
t6.Enabled = True
End If
End Sub
Private Sub t4_Timer()
If i <= 3 Then
If l1(3).Left < 4600 Then
l1(3).Visible = True
l1(3).Left = l1(3).Left + 100
End If
Else
t6.Enabled = True
End If
End Sub
Private Sub t5_Timer()
If i <= 3 Then
If l1(4).Left < 4600 Then
l1(4).Visible = True
l1(4).Left = l1(4).Left + 100
End If
Else
t6.Enabled = True
End If
End Sub
Private Sub t6_Timer()
If l1(0).Left < 4560 Then
```

```
If l1(0).Visible = True Then
l1(0).Visible = False
Else
l1(0).Visible = True
End If
End If
```

```
If l1(1).Left < 4560 Then
If l1(1).Visible = True Then
l1(1).Visible = False
Else
l1(1).Visible = True
End If
End If
```

```
If l1(2).Left < 4560 Then           // Label visibility.
If l1(2).Visible = True Then
l1(2).Visible = False
Else
l1(2).Visible = True
End If
End If
```

```
If l1(3).Left < 4560 Then
If l1(3).Visible = True Then
l1(3).Visible = False
Else
l1(3).Visible = True
End If
End If
```



```
If I1(4).Left < 4560 Then
If I1(4).Visible = True Then
I1(4).Visible = False
Else
I1(4).Visible = True
End If
End If
```

```
End Sub
```

```
Private Sub t7_Timer()
```

```
If I1(0).Left < 10000 And I1(0).Left > 4600 Then
I1(0) = LoadPicture("c:\teer1.bmp")
I1(0).Left = I1(0).Left + 210
I1(0).Top = I1(0).Top - 120
End If
```

```
If I1(1).Left < 10000 And I1(1).Left > 4600 Then
I1(1).Left = I1(1).Left + 210
End If
```

```
If I1(2).Left < 10000 And I1(2).Left > 4600 Then
I1(2) = LoadPicture("c:\teer2.bmp")
I1(2).Left = I1(2).Left + 210
I1(2).Top = I1(2).Top + 85
End If
```

```
If I1(3).Left < 10000 And I1(3).Left > 4560 Then
I1(3) = LoadPicture("c:\teer3.bmp")
I1(3).Left = I1(3).Left + 210
I1(3).Top = I1(3).Top + 30
End If
```

```
If I1(4).Left < 10000 And I1(4).Left > 4560 Then
I1(4).Left = I1(4).Left + 210
End If
```

```
If i > 3 Then
i = i - 1
t6.Enabled = False
End If
```

```
If l1(0).Left >= 10000 Then
l1(0).Picture = LoadPicture("c:\teer.bmp")
l1(0).Left = 120
l1(0).Top = 2160
End If
```

```
If l1(1).Left >= 10000 Then
l1(1).Picture = LoadPicture("c:\teer.bmp")
l1(1).Left = 120
l1(1).Top = 2160
End If
```

```
If l1(2).Left >= 10000 Then
l1(2).Picture = LoadPicture("c:\teer.bmp")
l1(2).Left = 120
l1(2).Top = 2160
End If
```

```
If l1(3).Left >= 10000 Then
l1(3).Picture = LoadPicture("c:\teer.bmp")
l1(3).Left = 120
l1(3).Top = 2160
End If
```

```
If l1(4).Left >= 10000 Then
l1(4).Picture = LoadPicture("c:\teer.bmp")
l1(4).Left = 120
l1(4).Top = 2160
End If
```

End Sub

Private Sub Timer1_Timer()

If 11(6).Visible = True Then

11(6).Visible = False

Else

11(6).Visible = True

End If

If 11(7).Visible = True Then

11(7).Visible = False

Else

11(7).Visible = True

End If

If 11(8).Visible = True Then

11(8).Visible = False

Else

11(8).Visible = True

End If

If 11(9).Visible = True Then

11(9).Visible = False

Else

11(9).Visible = True

End If

If 11(10).Visible = True Then

11(10).Visible = False

Else

11(10).Visible = True

End If

End Sub

Private Sub tt_Timer()

If ll(17).Left < 960 Then

ll(17).Left = ll(17).Left + 100

End If

If ll(17).Left > 960 Then

ll(17).Left = Default

End If

If ll(15).Left < 2160 And ll(16).Top > 1400 Then

ll(15).Visible = True

ll(15).Left = ll(15).Left + 300

End If

If ll(15).Left > 2060 Then

ll(11).Picture = LoadPicture("c:\teer.bmp")

ll(15).Left = 720

End If

If ll(16).Top < 2150 Then

ll(16).Visible = True

ll(16).Top = ll(16).Top + 100

End If

If ll(16).Top > 2150 Then

ll(16).Top = 1320

ll(16).Visible = False

End If

If ll(16).Top > 1500 Then

ll(14).Picture = LoadPicture("")

End If

If l1(14).Picture = Empty Then

l1(13).Picture = LoadPicture("")

l1(14).Picture = LoadPicture("c:\teer.bmp")

End If

If l1(13).Picture = Empty Then

l1(12).Picture = LoadPicture("")

l1(13).Picture = LoadPicture("c:\teer.bmp")

End If

If l1(12).Picture = Empty Then

l1(11).Picture = LoadPicture("")

l1(12).Picture = LoadPicture("c:\teer.bmp")

End If

End Sub

/*
/*****
/

Programm for different sources sending packets to a particular destination

Dim i As Integer, x As Integer, a As Integer, b As Integer, z As Integer, c As Integer, d
As Integer, e As Integer, f As Integer, g As Integer

Dim a1 As Integer, b1 As Integer, c1 As Integer, d1 As Integer, e1 As Integer, f1 As
Integer, g1 As Integer, z1 As Integer

Private Sub Command1_Click()

Command1.Enabled = False

For i = 0 To 6

l(i).Top = 2800

l(i).Left = 800

```
l(i).BackColor = vbBlack
```

```
Next i
```

```
t.Enabled = True
```

```
t2.Enabled = False
```

```
End Sub
```

```
Private Sub Command2_Click()
```

```
Command2.Enabled = False
```

```
For i = 0 To 6
```

```
l1(i).Top = 1500
```

```
l1(i).Left = 5900
```

```
l1(i).BackColor = vbBlack
```

```
Next i
```

```
t1.Enabled = True
```

```
t3.Enabled = False
```

```
End Sub
```

```
Private Sub Form_Click()
```

```
Command1.Enabled = True
```

```
Command2.Enabled = True
```

```
End Sub
```

```
Private Sub t_Timer()
```

```
If l(0).Left < 11100 And l(0).Visible = True Then
```

```
l(0).Left = l(0).Left + 100
```

```
End If
```

```
If l(0).Left > 1000 And l(1).Left < 11100 And l(1).Visible = True Then
```

```
l(1).Left = l(1).Left + 100
```

```
End If
```

```
If l(1).Left > 1000 And l(2).Left < 11100 And l(2).Visible = True Then
```

```
l(2).Left = l(2).Left + 100
```

```
End If
```

```
If l(2).Left > 1000 And l(3).Left < 11100 And l(3).Visible = True Then  
l(3).Left = l(3).Left + 100  
End If
```

```
If l(3).Left > 1000 And l(4).Left < 11100 And l(4).Visible = True Then  
l(4).Left = l(4).Left + 100  
End If
```

```
If l(4).Left > 1000 And l(5).Left < 11100 And l(5).Visible = True Then  
l(5).Left = l(5).Left + 100  
End If
```

```
If l(5).Left > 1000 And l(6).Left < 11100 And l(6).Visible = True Then  
l(6).Left = l(6).Left + 100  
End If
```

```
If a1 = 0 Then  
If l(0).Left > 10900 And l(0).Visible = True Then  
z1 = z1 + 1  
a1 = 1  
End If  
End If
```

```
If b1 = 0 Then  
If l(1).Left > 10900 And l(1).Visible = True Then  
z1 = z1 + 1  
b1 = 1  
End If  
End If
```

```
If c1 = 0 Then
If l(2).Left > 10900 And l(2).Visible = True Then
z1 = z1 + 1
c1 = 1
End If
End If
```

```
If d1 = 0 Then
If l(3).Left > 10900 And l(3).Visible = True Then
z1 = z1 + 1
d1 = 1
End If
End If
```

```
If e1 = 0 Then
If l(4).Left > 10900 And l(4).Visible = True Then
z1 = z1 + 1
e1 = 1
End If
End If
```

```
If f1 = 0 Then
If l(5).Left > 10900 And l(5).Visible = True Then
z1 = z1 + 1
f1 = 1
End If
End If
```

```
If g1 = 0 Then
If l(6).Left > 10900 And l(6).Visible = True Then
z1 = z1 + 1
```



```
g1 = 1
End If
End If
```

```
If l(6).Left >= 11100 Then
l(0).Left = 11100
l(1).Left = 11100
t.Enabled = False
t2.Enabled = True
End If
Text6 = z1
End Sub
```

```
Private Sub t1_Timer()
If l1(0).Top < 2800 Then
l1(0).Visible = True
l1(0).Top = l1(0).Top + 100
End If
If l1(0).Left > 700 And l1(0).Top >= 2800 Then
l1(0).Left = l1(0).Left - 100
End If
```

```
If l1(0).Top > 1700 Then
If l1(1).Top < 2800 Then
l1(1).Visible = True
l1(1).Top = l1(1).Top + 100
End If
If l1(1).Left > 700 And l1(1).Top >= 2800 Then
l1(1).Left = l1(1).Left - 100
End If
End If
```

```
If l1(1).Top > 1700 Then
  If l1(2).Top < 2800 Then
    l1(2).Visible = True
    l1(2).Top = l1(2).Top + 100
  End If
  If l1(2).Left > 700 And l1(2).Top >= 2800 Then
    l1(2).Left = l1(2).Left - 100
  End If
End If
```

```
If l1(2).Top > 1700 Then
  If l1(3).Top < 2800 Then
    l1(3).Visible = True
    l1(3).Top = l1(3).Top + 100
  End If
  If l1(3).Left > 700 And l1(3).Top >= 2800 Then
    l1(3).Left = l1(3).Left - 100
  End If
End If
```

```
If l1(3).Top > 1700 Then
  If l1(4).Top < 2800 Then
    l1(4).Visible = True
    l1(4).Top = l1(4).Top + 100
  End If
  If l1(4).Left > 700 And l1(4).Top >= 2800 Then
    l1(4).Left = l1(4).Left - 100
  End If
End If
```

```
If l1(4).Top > 1700 Then
If l1(5).Top < 2800 Then
l1(5).Visible = True
l1(5).Top = l1(5).Top + 100
End If
If l1(5).Left > 700 And l1(5).Top >= 2800 Then
l1(5).Left = l1(5).Left - 100
End If
End If
```

```
If l1(5).Top > 1700 Then
If l1(6).Top < 2800 Then
l1(6).Visible = True
l1(6).Top = l1(6).Top + 100
End If
If l1(6).Left > 700 And l1(6).Top >= 2800 Then
l1(6).Left = l1(6).Left - 100
End If
End If
```

```
If a = 0 Then
If l1(0).Left < 800 And l1(0).Visible = True Then
z = z + 1
a = 1
End If
End If
```

```
If b = 0 Then
If l1(1).Left < 800 And l1(1).Visible = True Then
z = z + 1
b = 1
```

End If

End If

If c = 0 Then

If l1(2).Left < 800 And l1(2).Visible = True Then

z = z + 1

c = 1

End If

End If

If d = 0 Then

If l1(3).Left < 800 And l1(3).Visible = True Then

z = z + 1

d = 1

End If

End If

If e = 0 Then

If l1(4).Left < 800 And l1(4).Visible = True Then

z = z + 1

e = 1

End If

End If

If f = 0 Then

If l1(5).Left < 800 And l1(5).Visible = True Then

z = z + 1

f = 1

End If

End If

```
If g = 0 Then
If l1(6).Left < 800 And l1(6).Visible = True Then
z = z + 1
g = 1
End If
End If
```

```
If l1(6).Left < 800 Then
t1.Enabled = False
t3.Enabled = True
End If
Text5 = z
End Sub
```

```
Private Sub t2_Timer()
If l(0).Left > 700 Then
l(0).BackColor = vbGreen
l(0).Visible = True
l(0).Left = l(0).Left - 100
End If
If z1 < 7 Then
If l(1).Left > 700 And l(0).Left < 10600 Then
l(1).BackColor = vbBlue
l(1).Visible = True
l(1).Left = l(1).Left - 100
End If
End If
```

```
If l(1).Left <= 700 Then
t2.Enabled = False
Timer1.Enabled = True
End If
```

```

End Sub
Private Sub t3_Timer()
If l1(0).Left < 5900 Then
l1(0).BackColor = vbGreen
l1(0).Visible = True
l1(0).Left = l1(0).Left + 100
End If
If l1(0).Left >= 5900 And l1(0).Top >= 1500 Then
l1(0).Top = l1(0).Top - 100
End If

If z < 7 Then
If l1(1).Left < 5900 And l1(0).Left > 1000 Then
l1(1).BackColor = vbBlue
l1(1).Visible = True
l1(1).Left = l1(1).Left + 100
End If
If l1(1).Left >= 5900 And l1(1).Top >= 1500 Then
l1(1).Top = l1(1).Top - 100
End If
End If

If l1(1).Top <= 1500 Then
l1(1).Visible = False
ttt.Enabled = True
t3.Enabled = False
End If
End Sub
Private Sub t4_Timer()
For i = 0 To 6
Text3 = l(0).Left

```

```
Text4 = l1(0).Left
Text1 = l(0).Top
Text2 = l1(0).Top
If l(0).Top = l1(i).Top And l(0).Visible = True And l(0).BackColor = vbBlack Then
If l(0).Left = l1(i).Left Then
s1.Visible = True
s1.Top = l(0).Top - 200
s1.Left = l(0).Left
l(0).Visible = False
l1(i).Visible = False
t5.Enabled = True
End If
End If
Next i
```

```
For i = 0 To 6
Text3 = l(0).Left
Text4 = l1(0).Left
Text1 = l(0).Top
Text2 = l1(0).Top
If l(1).Top = l1(i).Top And l(1).Visible = True And l(1).BackColor = vbBlack Then
If l(1).Left = l1(i).Left Then
s1.Visible = True
s1.Top = l(1).Top - 200
s1.Left = l(1).Left
l(1).Visible = False
l1(i).Visible = False
t5.Enabled = True
End If
End If
Next i
```

```
For i = 0 To 6
Text3 = l(0).Left
Text4 = l1(0).Left
Text1 = l(0).Top
Text2 = l1(0).Top
If l(2).Top = l1(i).Top And l(2).Visible = True And l(2).BackColor = vbBlack Then
If l(2).Left = l1(i).Left Then
s1.Visible = True
s1.Top = l(2).Top - 200
s1.Left = l(2).Left
l(2).Visible = False
l1(i).Visible = False
t5.Enabled = True
End If
End If
Next i
```

```
For i = 0 To 6
Text3 = l(0).Left
Text4 = l1(0).Left
Text1 = l(0).Top
Text2 = l1(0).Top
If l(3).Top = l1(i).Top Then
If l(3).Left = l1(i).Left Then
```

```
End If
End If
Next i
```

```
For i = 0 To 6
```



```
Text3 = l(0).Left
Text4 = ll(0).Left
Text1 = l(0).Top
Text2 = ll(0).Top
If l(4).Top = ll(i).Top Then
If l(4).Left = ll(i).Left Then
```

```
End If
```

```
End If
```

```
Next i
```

```
For i = 0 To 6
```

```
Text3 = l(0).Left
```

```
Text4 = ll(0).Left
```

```
Text1 = l(0).Top
```

```
Text2 = ll(0).Top
```

```
If l(5).Top = ll(i).Top Then
```

```
If l(5).Left = ll(i).Left Then
```

```
End If
```

```
End If
```

```
Next i
```

```
For i = 0 To 6
```

```
Text3 = l(0).Left
```

```
Text4 = ll(0).Left
```

```
Text1 = l(0).Top
```

```
Text2 = ll(0).Top
```

```
If l(6).Top = ll(i).Top Then
```

```
If l(6).Left = ll(i).Left Then
```

```
End If
End If
Next i
End Sub
Private Sub t5_Timer()
If x < 6 Then
If s1.Visible = True Then
s1.Visible = False
Else
s1.Visible = True
End If
x = x + 1
End If
If x > 5 Then
t5.Enabled = False
s1.Visible = False
x = 0
End If
End Sub
```

```
Private Sub Timer1_Timer()
Select Case z1
Case 1
Case 2
Case 3
Case 4
If l(0).Left < 11100 And l(0).Visible = True Then
l(0).BackColor = vbBlack
l(0).Left = l(0).Left + 100
End If
```

```
If l(0).Left > 1000 And l(1).Left < 11100 And l(1).Visible = True Then
l(1).BackColor = vbBlack
l(1).Visible = True
l(1).Left = l(1).Left + 100
End If
```

```
If l(1).Left > 1000 And l(2).Left < 11100 And l(2).Visible = True Then
l(2).BackColor = vbBlack
l(2).Visible = True
l(2).Left = l(2).Left + 100
End If
```

Case 5

```
If l(0).Left < 11100 And l(0).Visible = True Then
l(0).BackColor = vbBlack
l(0).Visible = True
l(0).Left = l(0).Left + 100
End If
```

```
If l(0).Left > 1000 And l(1).Left < 11100 And l(1).Visible = True Then
l(1).BackColor = vbBlack
l(1).Visible = True
l(1).Left = l(1).Left + 100
End If
```

Case 6

```
If l(0).Left < 11100 And l(0).Visible = True Then
l(0).BackColor = vbBlack
l(0).Visible = True
l(0).Left = l(0).Left + 100
End If
```

End Select

End Sub

```
Private Sub ttt_Timer()
For i = 0 To 6
l1(i).BackColor = vbBlack
Next i
Select Case z
Case 3
If l1(2).Top > 1700 Then
If l1(3).Top < 2800 Then
l1(3).Visible = True
l1(3).Top = l1(3).Top + 100
End If
If l1(3).Left > 700 And l1(3).Top >= 2800 Then
l1(3).Left = l1(3).Left - 100
End If
End If

If l1(0).Top < 2800 Then
l1(0).Visible = True
l1(0).Top = l1(0).Top + 100
End If
If l1(0).Left > 700 And l1(0).Top >= 2800 Then
l1(0).Left = l1(0).Left - 100
End If

If l1(0).Top > 1700 Then
If l1(1).Top < 2800 Then
l1(1).Visible = True
l1(1).Top = l1(1).Top + 100
End If
If l1(1).Left > 700 And l1(1).Top >= 2800 Then
```

l1(1).Left = l1(1).Left - 100

End If

End If

If l1(1).Top > 1700 Then

If l1(2).Top < 2800 Then

l1(2).Visible = True

l1(2).Top = l1(2).Top + 100

End If

If l1(2).Left > 700 And l1(2).Top >= 2800 Then

l1(2).Left = l1(2).Left - 100

End If

End If

Case 4

If l1(0).Top < 2800 Then

l1(0).Visible = True

l1(0).Top = l1(0).Top + 100

End If

If l1(0).Left > 700 And l1(0).Top >= 2800 Then

l1(0).Left = l1(0).Left - 100

End If

If l1(0).Top > 1700 Then

If l1(1).Top < 2800 Then

l1(1).Visible = True

l1(1).Top = l1(1).Top + 100

End If

If l1(1).Left > 700 And l1(1).Top >= 2800 Then

l1(1).Left = l1(1).Left - 100

End If

End If

If l1(1).Top > 1700 Then

If l1(2).Top < 2800 Then

l1(2).Visible = True

l1(2).Top = l1(2).Top + 100

End If

If l1(2).Left > 700 And l1(2).Top >= 2800 Then

l1(2).Left = l1(2).Left - 100

End If

End If

Case 5

If l1(0).Top < 2800 Then

l1(0).Visible = True

l1(0).Top = l1(0).Top + 100

End If

If l1(0).Left > 700 And l1(0).Top >= 2800 Then

l1(0).Left = l1(0).Left - 100

End If

If l1(0).Top > 1700 Then

If l1(1).Top < 2800 Then

l1(1).Visible = True

l1(1).Top = l1(1).Top + 100

End If

If l1(1).Left > 700 And l1(1).Top >= 2800 Then

l1(1).Left = l1(1).Left - 100

End If

End If

Case 6

If $l1(0).Top < 2800$ Then

$l1(0).Visible = True$

$l1(0).Top = l1(0).Top + 100$

End If

If $l1(0).Left > 700$ And $l1(0).Top \geq 2800$ Then

$l1(0).Left = l1(0).Left - 100$

End If

End Select

End Sub