

**ADAPTIVE CLUSTERING
IN
MOBILE, MULTIMEDIA, MULTIHOP
WIRELESS NETWORKS**

A DISSERTATION

*submitted in partial fulfilment of the
requirements for the award of the degree*

of

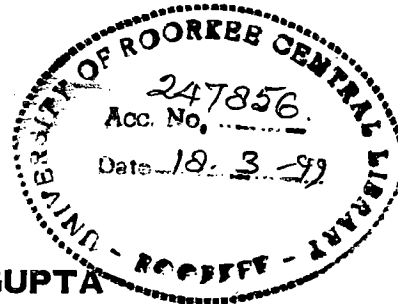
MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE AND TECHNOLOGY

By

SANJEEV GUPTA



**DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
UNIVERSITY OF ROORKEE
ROORKEE-247 667 (INDIA)**

JANUARY, 1999

CANDIDATE'S DECLARATION

I hereby declare that the work presented in this dissertation titled "**ADAPTIVE CLUSTERING IN MOBILE, MULTIMEDIA MULTIHOP WIRELESS NETWORKS**" in partial fulfillment of the requirements for the award of the degree of **Master of Technology in Computer Science and Technology**, submitted in the *Department of Electronics and Computer Engineering, University of Roorkee, Roorkee*, is an authentic record of my own work carried out during the period from July 1998 to January 1999 under the guidance of **Dr. (Mrs.) KUMKUM GARG**, *professor, Department of Electronics and Computer Engineering, University of Roorkee, Roorkee*.

I have not submitted the matter embodied in this dissertation for the award of any other degree or diploma.

Date: 21-1-1999

Place: Roorkee



(SANJEEV GUPTA)

CERTIFICATE

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

Date: 21-1-1999

Place: Roorkee


Dr. (Mrs.) KUMKUM GARG
Professor
Department of Electronics &
Computer Engineering
University of Roorkee
Roorkee.

ACKNOWLEDGEMENTS

At the submission of this dissertation work, I take the opportunity to express my deep sense of gratitude and indebtedness to my guide, **Dr. (Mrs.) KUMKUM GARG**, *professor, Department of Electronics and Computer Engineering, University of Roorkee, Roorkee* for her invaluable, tireless guidance and constructive criticisms throughout this dissertation. It is only due to her constant motivation and moral support, I was able to pull through many difficult phases of the work and bring it to a successful completion.

I would like to thank Dr. MANOJ MISHRA for his immense technical help.

Finally I would like to thank all my friends, who directly or indirectly helped me in completing this task successfully.


SANJEEV GUPTA

ABSTRACT

The dissertation work simulates self-organizing wireless Mobile, Multimedia, Multihop(M3) networks. The basis of the network is clustering, i.e. nodes are organized into non-overlapping clusters. The clusters are independently controlled. Since the network supports mobility, these clusters are to be dynamically reconfigured. This adaptive clustering algorithm very well takes care of overheads for configuring and reconfiguring clusters. It is fast deployable, making it useful for emergency networks. The network relies on a code-division multiple access scheme for multimedia support. The main advantages of network architecture are, that it provides spatial reuse of bandwidth, and that bandwidth can be shared or reserved in a controlled fashion in each cluster.

(Simulation results confirm that this architecture provides an efficient and stable infrastructure for the integration of different types of traffic in a dynamic radio network.)

In our work, we first find the minimum transmission range within which all nodes can access all other nodes. We then implement the given clustering algorithm. We also suggest a modified clustering algorithm and show through simulation results, how our algorithm increases the number of real time connections accepted ((Further, this algorithm can sustain more mobility, that is, it requires less reconfiguring on node movement.))

The implementation is done in C language on the TATA ELAXI RISC system under UNIX environment.

CONTENTS

	Page no.
CANDIDATE'S DECLARATION	(i)
ACKNOWLEDGEMENTS	(ii)
ABSTRACT	(iii)
1. INTRODUCTION AND BACKGROUND	1
1.1 Introduction to M3 Network	1
1.2 Statement of Problem	2
1.3 Organization of Dissertation	3
2. ADAPTIVE CLUSTERING	4
2.1 The Multi-Cluster Architecture	4
2.1.1 Assumption and Definition	5
2.1.2 The Clustering Algorithm	6
2.2 Cluster Maintenance in the Presence of Mobility	7
2.3 Code Assignment	8
2.4 Network Initialization	9
3. MAC LAYER PROTOCOL	10
3.1 Channel Access scheme	10
3.1.1 Collision-Free Channel Access Scheme	10
3.1.2 The inter-cluster communication	11
3.2 Acknowledgement for Datagram	13

4. NETWORK LAYER PROTOCOL	15
4.1 Bandwidth in Cluster Infrastructure	15
4.2 Bandwidth Reservation for VC Traffic	16
4.3 Adaptive Routing for Real-time Traffic	18
4.4 Routing for Datagram	20
5 DESGN AND IMPLEMENTATION	21
5.4 Simulation of Traffic	21
5.5 Simulation of Routing	22
5.6 Description of Function Used	22
6. CONCLUSION	24
6.4 Discussion Of Result	24
6.5 Future Scope of Work	27

REFERENCES

APPENDICES: Software Listing

INTRODUCTION AND BACKGROUND

1.1 MOBILE, MULTIMEDIA, MULTIHOP (M3) WIRELESS NETWORKS

Current wireless systems, such as ^{ABSTRACT}cellular systems, have fixed network and fixed base stations or servers that are linked by a wired backbone infrastructure. In some cases such as emergency disaster relief, when the backbone is not available, this type of architecture is infeasible. In this ^{paper}thesis we discuss a network architecture which overcomes these constraints. This is a wireless network, which is adaptable to a variety of transmission environments, networks configurations and user services (including data, voice and image). The architecture enables rapid deployment and dynamic reconfiguration of a network of wireless stations.

In conventional cellular communication, a mobile node is only one hop away from a base station. Another type of model, based on radio to radio multihopping, has been evolving to serve a growing number of applications which rely on a fast deployable, multihop, wireless infrastructure. Classic applications for this are battlefield communication, disaster recovery and search and rescue. A recent addition to this set is the “ad hoc” personnel communication network, which could be rapidly deployable on a campus, for example, to support collaborative computing and access to Internet during special events. The main advantage of multihopping through wireless repeaters is to reduce battery power and to increase network capacity (via spatial reuse).

More precisely, in the ^{paper}thesis, we are concerned with the design of efficient Multihop, Mobile, Multimedia (M3) wireless networks. The M3 problem has

been recognized as a very difficult problem. Over a decade ago, the ARPA sponsored Packet Radio Network did provide an efficient solution to multihop, mobile requirement of battlefield and disaster relief communication. It fell short, however, of supporting multimedia services.

Recently, the M3 problem was revisited under the ARPA sponsored WAMIS and GLOMO projects [9]. In this scheme the network is dynamically partitioned into clusters where each cluster uses different spreading-codes. Clusters (with code separation) improve spatial reuse. They also make it easier to manage real-time connections since each cluster can manage its own bandwidth.

In this thesis we have dealt with the above mentioned M3 wireless networks by simulating them and doing a performance analysis based on clustering algorithm given in the [11]. We then show that, by limiting the cluster size, a considerable improvement in real-time connection acceptance and datagram throughput can be obtained (detailed in our modified clustering algorithm, chapter 2).

1.2 STATEMENT OF THE PROBLEM

The dissertation deals with the problem of

- simulating the M3 network
- finding the minimum transmission range within which every node can access all other nodes
- simulating the clustering algorithm given in [11].
- simulating the modified clustering algorithm and comparing the various cluster properties with the given algorithm's properties.
- finding the performance of the network in terms of real-time connections accepted and datagram throughput.

1.3 ORGANISATION OF DISSERTATION

Including this introductory chapter in M3 networks, this dissertation is organized as follows:

In chapter 2 we define the various terms used in M3 networks and then describe the given adaptive clustering algorithm. The emphasis is on the important issue of overheads while dealing with mobility of nodes. We also present our modified clustering algorithm.

Chapter 3 describes the collision free channel access scheme and deals with the reliability of datagrams through piggybacked reservation scheme.

Chapter 4 gives details of routing of real-time traffic and datagrams. It explains a loop free highly dynamic routing algorithm for mobile nodes.

Chapter 5 shows the simulation model and explains the various functions and routines used in the simulation of the above algorithms.

Chapter 6 gives details of the results obtained using both the given and the modified clustering algorithms. It also lists the open problems in the area and gives suggestion for further work.

Software listing is given in Appendices.

ADAPTIVE CLUSTERING

In order to support multimedia traffic, the wireless network layer must guarantee QoS (bandwidth and delay) to real-time traffic components. Our approach for QoS to multimedia consists of the following two steps: 1) partitioning of multimedia network into clusters, so that controlled, accountable bandwidth sharing can be accomplished in each cluster; and 2) establishment of virtual circuits with QoS guarantee.

2.1 THE MULTICLUSTER ARCHITECTURE [11]

Most hierarchical clustering architectures for mobile radio networks are based on the concept of a clusterhead. The clusterhead acts as a local coordinator of transmission within the cluster. It differs from the base station concept in current cellular systems in that it does not have special hardware and in fact it is dynamically selected among the set of stations. However, it does extra work with respect to ordinary stations, and therefore it may become the bottleneck of the cluster. To overcome these difficulties, in our approach, we abandon the clusterhead approach altogether and adopt a fully distributed algorithm.

The objective of the clustering algorithm is to find an interconnected set of clusters covering the entire node population.

2.1.1 ASSUMPTIONS AND DEFINITIONS

Each node contains an identical transceiver, which can either transmit or receive at any given time. In addition, each node uses an omni-directional antenna for transmission. We assume there is a particular set of spread spectrum code with low cross-code interface. Since the number of codes we can use is very limited, spatial reuse of codes will be important. Finally, all radio nodes use the same power for transmission. The following definitions and notations will be frequently used in the thesis.

Definition 1: (System Topology)

The system topology is a graph $G=(V, E)$, where V is the set of nodes, and e is the set of logical edges. It is used to represent a radio network. There is only one transceiver in each node and the network operates in a half-duplex mode. A logical edge (x, y) means that node y is node X 's one-hop neighbor under the current transmitting power, and vice-versa.

In Fig. 1 we have an example topology.

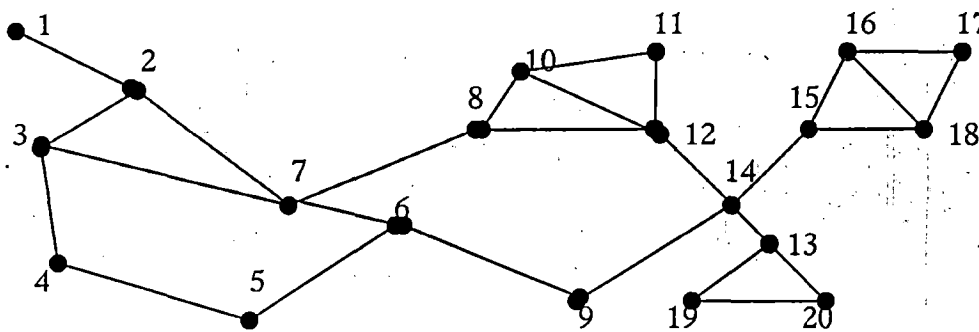


Fig. 1 System topology

Definition 2: (Distance of two nodes)

The distance $d(x, y)$ of two nodes x and y of G is defined to be the minimal number of hops from x to y .

Definition 3: (Cluster)

A *cluster* C_i is a set of nodes, where for any two nodes x, y which are elements of C_i , $d(x,y) \leq 2$. Namely any two nodes in a cluster are at most two hops apart.

Definition 4: (Degree of a Topology)

The *degree* of a topology is the number of clusters in the topology.

Definition 5: (Repeater, Bridge and the Order of a Repeater)

For an edge $u=(x, y)$, x and y are called repeaters if they belong to different clusters. u is called a *bridge*. The number of clusters, which a repeater can reach in one hop, is called *the order of the repeater*. The order of a repeater includes the cluster, which it belongs to. Thus, the minimal order of a repeater is 2.

2.1.2 THE CLUSTERING ALGORITHM

Let $T1(x)$ be the set of one-hop neighbors of the node x , which has the maximum number of one-hop neighbors.

- 1 $i = 0$
- 2 $x = \min(v)$
- 3 $C_i = \{x\} \cup T1$
 $V = v - C_i$
 $E = E - w(C_i)$
- 4 If v is not $\{ \}$ then $i = i+1$ and go to 2;
else stop.

We have made a modification in step3 of the algorithm by limiting the size of clusters; in case x has more than $n/4$ one-hop neighbors, we include only the nearest $\lfloor n/4 \rfloor$ nodes.

So in the modified clustering algorithm $C_i = \{x\} \cup T2$, where $T2$ is the set of the nearest $\lfloor n/4 \rfloor$ nodes.

This is the centralized version of the clustering algorithm and the advantage of the algorithm lies in the fact that it can be implemented in fully distributed manner i.e. without need of any clusterhead.

2.2 CLUSTER MAINTENANCE IN THE PRESENCE OF MOBILITY [1]

In the dynamic radio network, nodes can 1) change location, 2) be removed, and 3) be added. A topological change occurs when a node disconnects or connects from/to all or parts of its neighbors, thus altering the cluster structure. System performance is affected by frequent cluster changes. Therefore it is important to design a cluster maintenance scheme to keep the cluster infrastructure as stable as possible. In this respect, the proposed algorithm is more robust than the referred one because it chooses those nodes, which are nearest. The cluster maintenance scheme was designed to minimize the number of node transitions from one cluster to another.

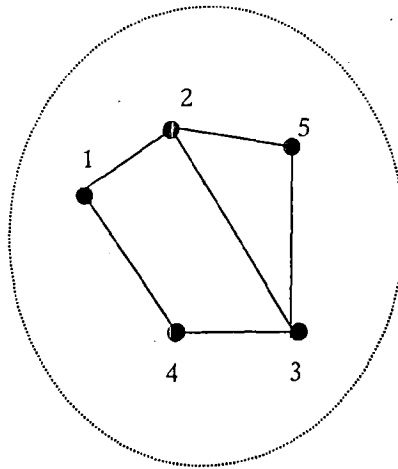


Fig. 2(a) Clustering

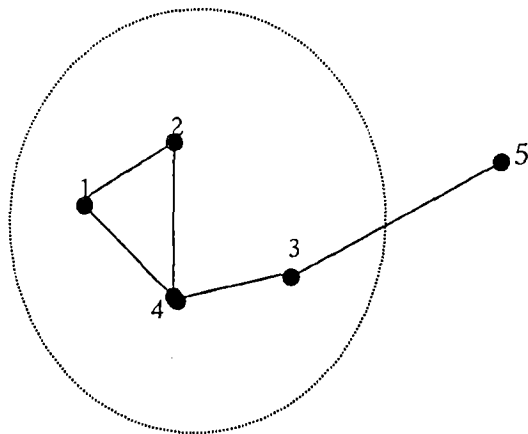


Fig 2(b) Reclustering

Let us take an example, as shown in **Fig. 2**. There are five nodes in the cluster and the hop distance is no more than two. Because of mobility, the topology changes to the configuration shown in **Fig 2(b)**. At this time, $d(1, 5) = d(2, 5) = 3 > 2$, where $d(i, j)$ is the hop distance between j and i . So the cluster needs to be reconfigured. Namely, we should decide which node(s) should be removed from the current cluster. We let the highest connectivity node and its neighbors stay in the original cluster, and we remove the other nodes. We know that each node only keeps the information of its locality, that is, its one-hop and two-hop neighbors. Upon discovering that member say x , of its cluster is no longer in its locality, node y should check if the highest connectivity node is a one-hop neighbor. If so, y removes x from its cluster. Otherwise, y changes cluster.

Two steps are required to maintain the cluster architecture:

Step 1: Check if there is any member of my cluster that has moved out of my locality.

Step 2: If yes, decide whether I should change cluster or remove the nodes not in my locality, from my cluster.

We see the example shown in **Fig 2(b)**. Node 4 is the highest connectivity node. Thus node 4 and its neighbors $\{1,2,3\}$ do not change cluster. However node 5 should either join another cluster or form a new cluster. If a node intends to join a cluster, it has to check first if all members of this cluster are in its locality. Only in this case can it join the cluster.

2.3 CODE ASSIGNMENT [1]

Each node has a transceiver, which can either transmit or receive at any given time. In the spread -spectrum code-division system, the receiver should be set to

the same code as the designated transmitter. We assume there is a small set of good spread-spectrum codes, which have low cross-correlation. Since the numbers of codes we can use are limited, the spatial reuse of code will be important. Thus each cluster is assigned a single code which is different from the codes used in the neighbor clusters.

The essence of our transmitter based code assignment scheme is “within a cluster, every node uses a common transmitting code so that there is no **intercluster** collision. If no two nodes in a cluster are transmitting simultaneously, there will be no **intracluster** collision” .

2.4 NETWORK INITIALIZATION [1]

Initialization is carried out using a common “control” code. A node, which does not yet belong to a cluster, listens to a control code until timeout. Then it transmit its own ID (using the control code), and repeats the procedure until it hears from one of the neighbors. Channel access in this phase is CSMA. This basic communication facility allows nodes to organize themselves in clusters following the algorithm just described. Once a cluster is formed, the cluster leader communicates with the neighbors (using the control code) to select the codes. Only when the code assignment is completed (i.e. each cluster has been assigned its code) can user data be accepted by nodes and transmitted in the network.

MAC LAYER PROTOCOL

In this chapter, we introduce the medium access control (**MAC**) protocol. The aim here is to support integrated traffic (datagram and real time) efficiently. We will assume fixed packet size.

3.1 THE CHANNEL ACCESS SCHEME

The two nodes, which are communicating, may or may not belong to same cluster. So communication between two nodes which belong to different clusters, requires two steps, namely intracluster and intercluster communication.

3.1.1 COLLISION-FREE CHANNEL ACCESS SCHEME

Since our system is distributed and each cluster uses a common channel for packet transmission, we employ a round-robin (**RR**) scheme, which completely rotates the access priority among the nodes, to make the channel access distributed and conflict-free. The **RR** scheme gives each node in turn an opportunity to transmit a packet. In addition, the short propagation-to-transmission time ratio makes **CSMA** suitable as an access scheme. Thus, we implement **RR** over **CSMA** slotted **ALOHA** (**CSMA-RR**) for packet transmission within a cluster. In this scheme, if a node, say x , relinquishes its turn to transmit, its one-hop neighbors contend for this free time slot. The right to transmit in the next time slot passes to the next node of x in logical sequence.

Since our system assumes a common transmitting code in each cluster, there is no intercluster collision. The receiver must tune to transmitter's code to receive the packet. It is relatively easy to maintain time (i.e. slot) synchronization within each cluster. So the channel will be assumed slot synchronized. This is much easier than maintaining slot synchronization across the entire network. It is important to note that in this scheme synchronization is required only within a cluster.

3.1.2 INTER-CLUSTER COMMUNICATION

In addition to the two codes (for information and **ACK**) assigned to each cluster, two codes are also assigned to each edge which connects a pair of adjacent clusters for inter-cluster communication. Namely, there are two channels, the transmission (**Xmt**) channel and the acknowledgment (**Ack**) channel, for each pair of adjacent clusters.

Each repeater must periodically listen to different codes (in fact, as many codes as its order). We assume that the repeater, when it is free from voice traffic, shares its time randomly among the various codes.

The access on the transmitting channel can be simply carried out in a **CSMA unslotted ALOHA** fashion. Namely, if the channel is sensed busy, or if the transmission is unsuccessful (no **ACK**), the packet is regarded as backlogged.

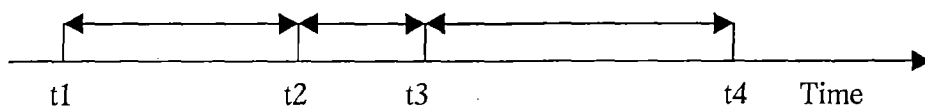


Fig. 3.1 Random Delay

As Fig 3.1 shows, each backlogged packet repeatedly attempts to retransmit at randomly selected times separated by random delays t . If the channel is idle at one of these times, the packet is transmitted, which continues until such a transmission is successful. Upon receiving a packet successfully, the receiver uses the *Ack* channel to transmit an explicit ACK packet immediately.

At the first packet transmission, the voice source reports the spreading code, which is different from the code used in X_{mt} channel and will be used for the following voice packet transmission. Thus, after the first successful transmission, the receiver listens to this spreading code. On the other hand, the voice source schedules its next transmission at a fixed time T_s as intra-cluster communication and uses the piggyback reservation with packet transmission to reserve the time slot.

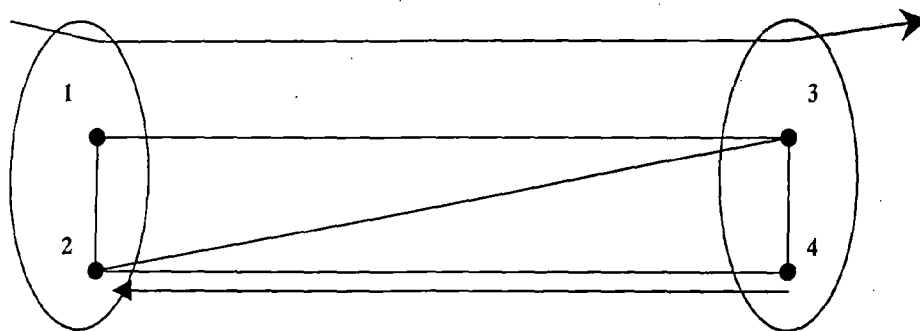


Fig. 3.2

For example in Fig. 3.2, suppose that a voice stream is transmitting through edge (1,3). The voice packets are encoded by another spreading code different from the code used in X_{mt} channel. At this time, the node 4 can transmit a data packet to node 2 through X_{mt} channel without collision at node 3.

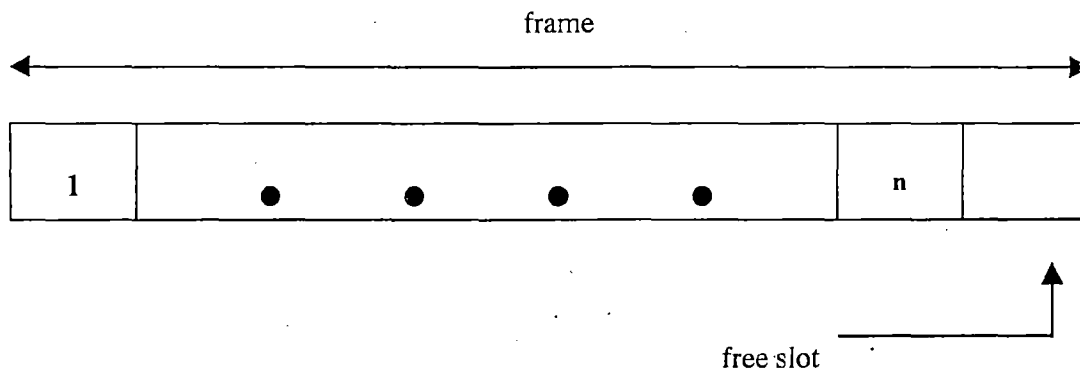


Fig. 3.3. Channel Access Frame within a Cluster

As shown in the Fig. 3.3, we assume that there are n nodes in a cluster. Time is divided into slots, which are grouped into frames. In each frame, a free slot is reserved for a new node joining the cluster. Using the control code, the nodes in the cluster take turns to transmit periodically in the free slot, their cluster and code information, for the purpose of attracting new nodes. When a node decides to join a cluster, it listens to the channel for a period of time, and then uses this free slot to transmit packets temporarily. Since cluster switches are infrequent, one free slot will suffice. The frame is readjusted after each join/leave.

3.2 ACKNOWLEDGMENT FOR DATAGRAM

Datagram traffic is error sensitive. Thus, it is important to design a reliable transmission for datagrams. Each cluster has a dedicated code for transmission. Since every node can only transmit packets in its assigned TDMA slots, we use an implicit acknowledgement scheme. Upon receiving a packet successfully, the intended receiver piggybacks the

acknowledgment on its data packet at its assigned slot. The transmitter listens to the receiver's slot and code. If a time out occurs, it retransmit the packet.

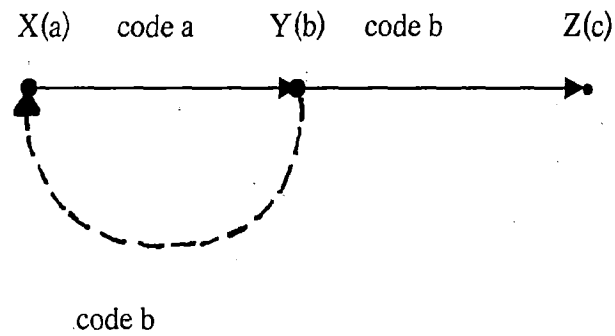


Fig. 3.4 Implicit acknowledgement scheme

Fig. 3.4 illustrates this implicit ACK scheme. Node x uses code a to transmit its packet to y, and listens to code b for ACK. Node y receives the packet successfully. When its transmitting slot comes, y piggybacks an ACK for x on the packet, which it transmitted to z.

NETWORK LAYER PROTOCOL

A multimedia application such as digital audio or video has much more stringent QoS requirement than a traditional datagram application. For a network to deliver QoS guarantees, it must reserve and control resources. Routing is the first step in resource reservation. The routing protocol first finds a path with sufficient resources. Then, the resource setup protocol makes the reservations along the path.

4.1 BANDWIDTH IN CLUSTER INFRASTRUCTURE [1]

The key resource for the multimedia QoS support is bandwidth. We define bandwidth in our cluster infrastructure for the purpose of real-time connection support, as the number of real-time connections that can pass through that node. Since, in our scheme a node can at most transmit one packet per frame, the bandwidth of a node is given by

$$\text{bandwidth} = (\text{int}) (\text{cycle time}/\text{frame time})$$

here CYCLE TIME is the maximum interval between two real-time packets.

and frame time of a cluster depends on how many nodes there are in the cluster.

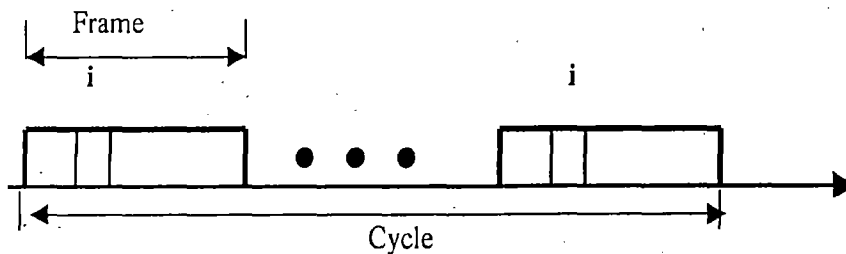


Fig.4.1 Noise Bandwidth



Fig. 4.1 shows the slots dedicated to node i in the cycle, which correspond to node i 's bandwidth. For example, if cycle time is 24, consider cluster C1, where frame size is equal to six slots. Thus the node bandwidth in C1 is $24/6 = 4$. Since there are three VCs passing through node c , the available bandwidth for node c is one.

4.2 BANDWIDTH RESERVATION FOR VC TRAFFIC [1]

A real-time connection is set up using a fast reservation approach. We assume that real-time packets arrive at constant time intervals. The first data packet in the multimedia stream makes the reservation along the path. Once the first data packet is accepted on a link, a transmission window is reserved (on that link) at appropriate time intervals for all of the subsequent packets in the connection. The window is released when idle for a pre-specified number of cycles.

Each real-time connection is assigned to a VC which is an end-to-end path along which slots have been reserved. The path and slot of a VC may change dynamically during the lifetime of a connection due to mobility. Each node schedules each of its slots to transmit either datagram or VC traffic. Since real time traffic needs guaranteed bandwidth during its active period, each node has to reserve its own slots to the VC at connection set up time.

When a node intends to set up a VC to its neighbor, it transmits the first session of the packet in its TDMA slot. After successfully receiving the packet, the intended receiver will set up the reservation for receiving the next packet, since the next transmission time is piggybacked on the current packet. Since the sender always uses the same code to transmit packets, the intended receiver only needs to lock on that code when the reserved slot comes. If the link is not broken due to mobility, the subsequent packet will be received successfully.

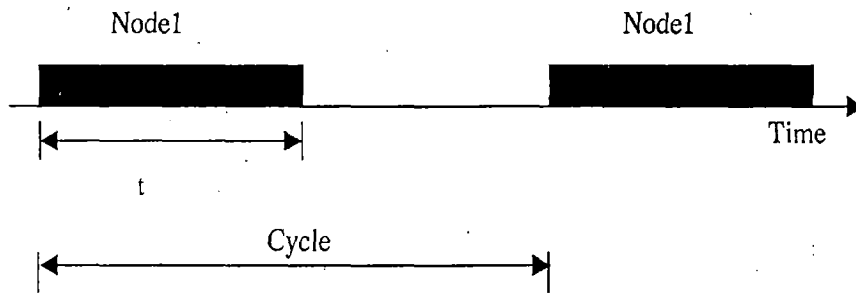


Fig.4.2 Bandwidth Reservation

Let **CYCLE** be the maximum interval tolerated between two real-time packets. The first packet of the real-time session is treated as a data packet and is transmitted using **TDMA**. It has higher priority than data packets in the local queue. A real-time source schedules its next transmission after a time **CYCLE** following a successful transmission, and piggybacks the reservation with the current transmission. **Fig. 4.2** shows that node *i* successfully transmits the first real-time packet and it reserves the time slots for the real-time session. The receiver has to listen to the sender's transmitting code when the reserved time slot comes. So, for real time sources, transmission is always collision-free and the maximal delay is guaranteed. At the end of real-time session (i.e., the reservation field is set to zero), the reservation is automatically canceled.

Because of the limitation of node bandwidth in a cluster, the number of real-time sessions which can pass through a node, is restricted. Slots, which are not reserved by voice traffic, are accessed according to a **TDMA** protocol. Datagram packets become backlogged when real-time traffic starts building up. For example, we consider the case when bandwidth of a node is completely used by a real-time session. Hence, there are $\lfloor \text{CYCLE}/((n+1)*t) \rfloor$ real-time sessions (in **Fig. 4.2**) over a node, where *n* is the number of members in the cluster and *t* is the packet transmission time. No other source can construct a VC which passes through the saturated node until one of the VC's over the saturated node ends its transmission and bandwidth become available.

ADAPTIVE ROUTING FOR REAL-TIME TRAFFIC [4]

When real-time traffic is considered to transmit over the dynamic network, the objective of routing protocol is to keep communication going. Routing optimality (e.g., shortest path) is of secondary importance; the routing protocol must be capable of establishing new routes for real-time sessions quickly when a topological change destroys existing routes. So we set the goal of the bandwidth routing algorithm to **“find the shortest path such that the free bandwidth this above the minimum requirement”**.

To compute the **bandwidth** constrained shortest path, we use the **DSDV** (destination sequenced distance vector) [4] routing algorithm which was proven to be loop free. Loop freedom follows from the fact that the updates generated by a destination are sequentially numbered. In our shortest path computation, the weight of each link is equal to one (i.e. minimal hop distance routing). The bandwidth constraint is simply accounted for by setting to infinity the weights of all the links to/from a node with zero bandwidth. An advantage of this scheme is to distribute the real-time traffic evenly across the network. A cluster with small frame size will allow more connections to pass through it, since it has more “bandwidth” per node.

In addition to load balancing, our routing scheme also supports the alternative path. This is very important in the mobile environment, where links will fail because of mobility. In such an environment, routing optimality is of secondary importance. The used routing protocol is capable of finding new routes quickly when a topological change destroys existing routes. To this end, the algorithm proposes to maintain secondary paths, which can be used immediately when the primary path fails.

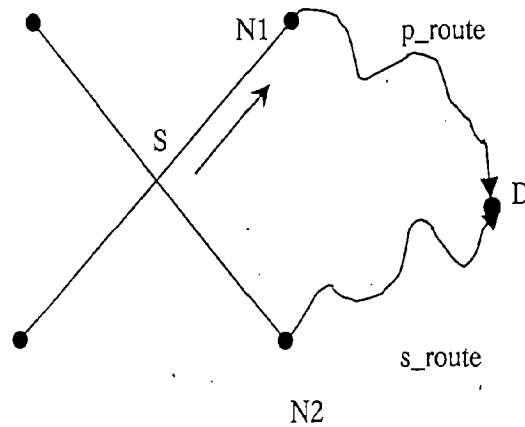


Fig. 4.3 Standby routing

For example in Fig. 4.3, each node uses the primary route to route its packets. When the first link on the path (s, N1) fails, the secondary path (s, N2) becomes the primary path, and another standby path (s, N3) will be computed as shown in the Fig.4.4.

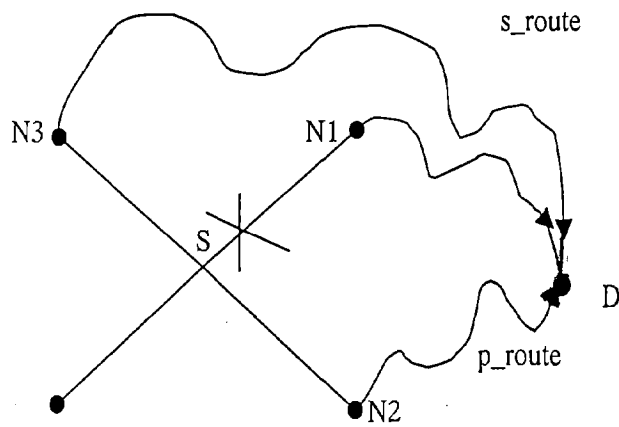


Fig. 4.4 The primary route fails and the standby route becomes the primary route. Another standby route is constructed.

It is to be pointed out that these routes are using different immediate successors to avoid failing simultaneously.

The secondary route is easily computed using the DSDV algorithm. Referring to the Fig. 4.3, we see that each neighbor of node S periodically informs S of its distance to D. The neighbor with shortest distance yields the primary route. The runner-up yields the secondary route. The scheme guarantees that the first link is different for the two paths. Furthermore, the standby route computation requires no extra table, message exchange or computation overhead. Also the standby route is as loop free as the primary route is.

ROUTING FOR DATAGRAMS [1]

In order to minimize delay for real time traffic we choose even those paths which may not be shortest, this is what we have achieved through the bandwidth constrained routing. But to achieve efficiency, datagrams are always pushed into the network in a way so that they always follows the shortest path from source to destination.

IMPLEMENTATION AND DESIGN

The multicluster architecture is simulated using various C-structures. We performed several sets of experiments in order to evaluate the performance as a function of traffic and system parameters.

The simulation is described in the following sections.

5.1 SIMULATION OF TRAFFIC

The channel rate is 800 Kbits/sec (the nominal rate of the radio under development, in the ARPA sponsored WAMIS project). All data packets (datagram and real-time) are of 4 kbits.

The offered traffic consists of two components, real-time and datagram. A new real-time session is generated on average every second (Poisson arrival model) between a random pair of nodes. Session duration is exponential with average duration of 3 minutes. The real-time cycle is allowed to vary from 1 pkt/sec to 10 pkts/sec.

We simulated the following condition by using two C-lists of structures, one for real-time and another for datagrams. Each structure of real-time traffic has members to show number of packets 1) that will be transmitted in the session 2) after which the new real-time session will start 3) source and destination between which the real-time session is to start. Another C-list is for datagrams where each structure need to contains only source and destination information, because in our experiments we assumed uniform datagram rate, namely 10 pkts/sec.

5.2 SIMULATION OF ROUTING

We used highly dynamic Destination-Sequenced Distance-Vector (DSDV) which guarantees loop freedom. The routing for the real-time traffic is "bandwidth constrained shortest path" where as for the datagram it is "shortest path".

We simulated the above mentioned properties of our routing technique by taking each node as a structure. We assigned each node an initial available bandwidth, which is calculated on the basis of number of nodes in the cluster to which the particular node belongs and CYCLE-time. Then for each real-time session we find the path. If path is available (that is enough bandwidth is available at all the nodes between source and destination) then we reserve the bandwidth along the path. Each node then calculates the currently available bandwidth. Our strategy gives path as per available bandwidth, that is, if enough bandwidth is not available along the shortest path, then less efficient path will be given if enough slots are available along that path.

For the datagram case we set the currently available bandwidth to infinity, so it always follows the shortest path. While finding the path we do not consider the node, which is already in the path, so we ensure loop freedom.

5.3 DESCRIPTION OF FUNCTIONS USED

Many functions are used in the simulation work; we list here some important functions. However, an exhaustive list of all functions used can be found in the APPENDICES.

random_deploy() : By this function we generate different topology with n uniformly distributed nodes in a 100*100 square area.

one_hop_neighbor() : The function finds the one-hop neighbor, if any, of all the nodes in the topology. The transmission range is varied and for each value, it finds the one-hop-neighbors.

cluster_form() : The function makes the clusters with the help of function find_min which find the nearest given number of minimum nodes.

random_redeploy() : In our simulation, every 100 ms, each node moves in a direction uniformly distributed over the interval $(0, 2\pi)$, covering a distance of $(0, 3)$.

Function redeploys the nodes as per defined mobility.

cluster_form() : With the help of many other function, this function does the important job of making clusters.

modify_list2() : After the formation of cluster, the function removes all those nodes which were in the cluster from the one-hop-neighbor list of another nodes. Thus the function ensure non-overlapping clusters.

effected_node() : Because of mobility the nodes will migrate from one cluster to another or form a new cluster. The function counts the number of nodes that are effected per 100 ms.

count_order() : The function finds order of each repeater respectively.

count_repeater() and **count_bridges()** : Functions find number of repeaters and order of each repeater respectively.

count_bridge() and **cluster_size()** : Functions find the number of bridges and average size of clusters in the topology.

real_traffic_generation() : The function generate the real-time traffic between the random pair of nodes on average every second (Poisson arrival model). Session duration is exponential with 3-minute average.

send_data_gram() : The function generate the datagrams between random pair of nodes at the constant rate of 10 pkts/sec.

CONCLUSION

We studied the problem of M3 networks in the following sequence as:

1. Finding the minimum transmission range within which each node can access all other nodes.
2. Applying the given and modified clustering algorithm to nodes and then comparing the various clustering parameters, namely, average cluster size, number of repeaters, order of repeaters, average number of bridges and finally the number of nodes which are affected due to mobility

We performed several sets of experiments with varying number of nodes namely, 20, 30 and 40. At each node, value we then vary the transmission range.

Finally, we studied the system performance in terms of number of real-time connections and datagrams accepted in a mixed traffic condition, where we vary the rate of real-time traffic.

6.1 DISCUSSION OF RESULTS

Connectivity:

Connectivity is defined as the fraction of node pairs, which can communicate through single or multiple hops. We want to see the impact of transmission range on connectivity.

We assumed an ideal network model where a link can be established between any two nodes within transmission range of each other. We note in **Fig. 6.1** that in order to

guarantee that all nodes can communicate with each other, the transmission range should be more than 30 for $n = 40$ and more than 40 for $n = 20$.

Repeaters:

Repeaters relay packets from one cluster to another. Since our topology is dynamic the reliability of packet routing is important to guarantee the integrity of network services. Thus, the existence of at least one path between a pair of nodes is required. The number of repeaters will affect the number of paths. Namely, the larger the fraction of nodes which are repeaters, the larger the number of alternate path.

Our modified algorithm is superior to the given one in this context, as can be seen by comparing Fig. 6.2(a) and 6.2(b). In the given algorithm the number of repeaters starts to decrease after the transmission range of 55 but in ours it continues to increase with range, the peak is approximately $3/4^{\text{th}}$ of the number of nodes which is more than the peak obtained from the given algorithm.

Further, the **number of bridges** will also affect the number of paths, these are larger in our clustering algorithm, as is seen from Fig. 6.3.

Average cluster size:

This parameter actually does not give the correct information about cluster size. At a particular transmission range, say 45, with number of nodes 20, in the given algorithm, since there is no restriction on cluster size, the typical clusters formed are of 13,4,1,1 (number of nodes) size respectively. Average cluster size comes out to, be 4. We see that average cluster size in the previous algorithm does not reveal the fact that there is

need of restricting the size of clusters especially when transmission range is high, say, above 70, where one of the clusters takes approximately $3/4^{\text{th}}$ of the total nodes and the other clusters are merely of size 1.

In our algorithm, we restrict the size of clusters so they are approximately of the same size and it can be seen from **Fig. 6.4(b)** that it does not shoot up at high transmission range as in **Fig. 6.4(a)**.

Order of the repeater:

Every repeater is time shared among the set of adjacent clusters, that is, its spreading code must be transmitted to these clusters. So, the order of a repeater should be small in order to maintain efficient operation, with as few code changes as possible (the minimal order of repeater is two).

We see that our algorithm makes the operation of the system, a bit complex. Increasing of the order of repeaters at high transmission range is not of concern because optimal transmission range is far less than 100. At 20 nodes it is 40, where average order is approximately 2.7 (2.5 in previous algorithm), at 30 nodes it is 35 where average order is 2.9 (2.7 in the previous algorithm) and at nodes 40 it is 30 where order is 3.0 (2.9 in previous algorithm).

We see from **Fig. 6.5(b)** that increase in order of repeaters is marginal at the concerned range of operation.

Affected node:

By comparing **Figures 6.6 (a) and (b)** we can say in our algorithm, node migration is almost nil at the concerned range of operation, that is affected nodes are approximately zero. It is important since it reduces the reconfiguration overheads.

Number of real-time connection accepted:

This is an important parameter for the improvement of which we compel to make modification in the existing algorithm. We see from **Fig. 6.7** that at transmission of 45, with nodes 20 and real-time packet rate at 10 pkts/sec, in previous algorithm only 10 of the request are accepted out of 100. Our algorithm improves it up to 30. We see if we allow real-time connection to send only 5 pkts/sec almost all requests are accepted.

We conclude that our modification loses only marginally in controlling the order of repeaters, but it improves various other parameters substantially of which great concern were number of nodes migrated due to mobility, and real-time and datagrams accepted.

6.2 FUTURE SCOPE OF WORK

Our modified clustering algorithm improved the number of real-time connection accepted and datagram throughput. Further, the architecture now can bear more node mobility. Our gain is substantial in terms of number of nodes effected due to mobility. But the price paid is increase in the order of repeaters. We restricted the cluster size irrespective of number of nodes. A proper selection of size with different number of nodes can decrease the order of repeaters.

Further, in our network, we always give preference to real-time traffic and go on reserving the bandwidth for real-time session till it is available. This will lockout datagram traffic. System performance in this context can be improved if some optimal fraction of bandwidth is made free for datagrams.

We have used DSDV routing algorithm to ensure loop freedom. There is need of finding good functional values of few parameter of this routing algorithm such as average convergence time, full update period, incremental update period.

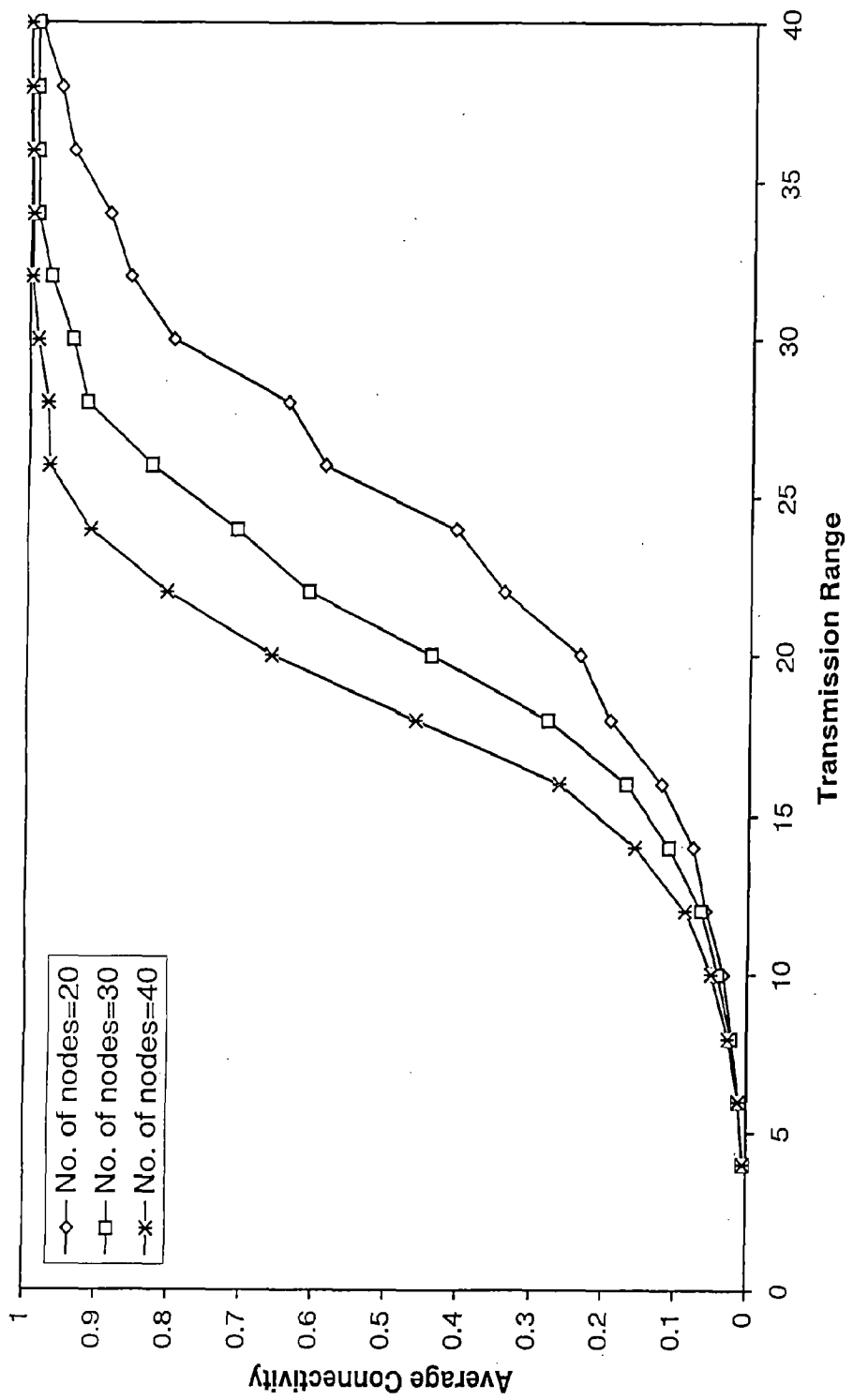


Fig. 6.1 Connectivity Property

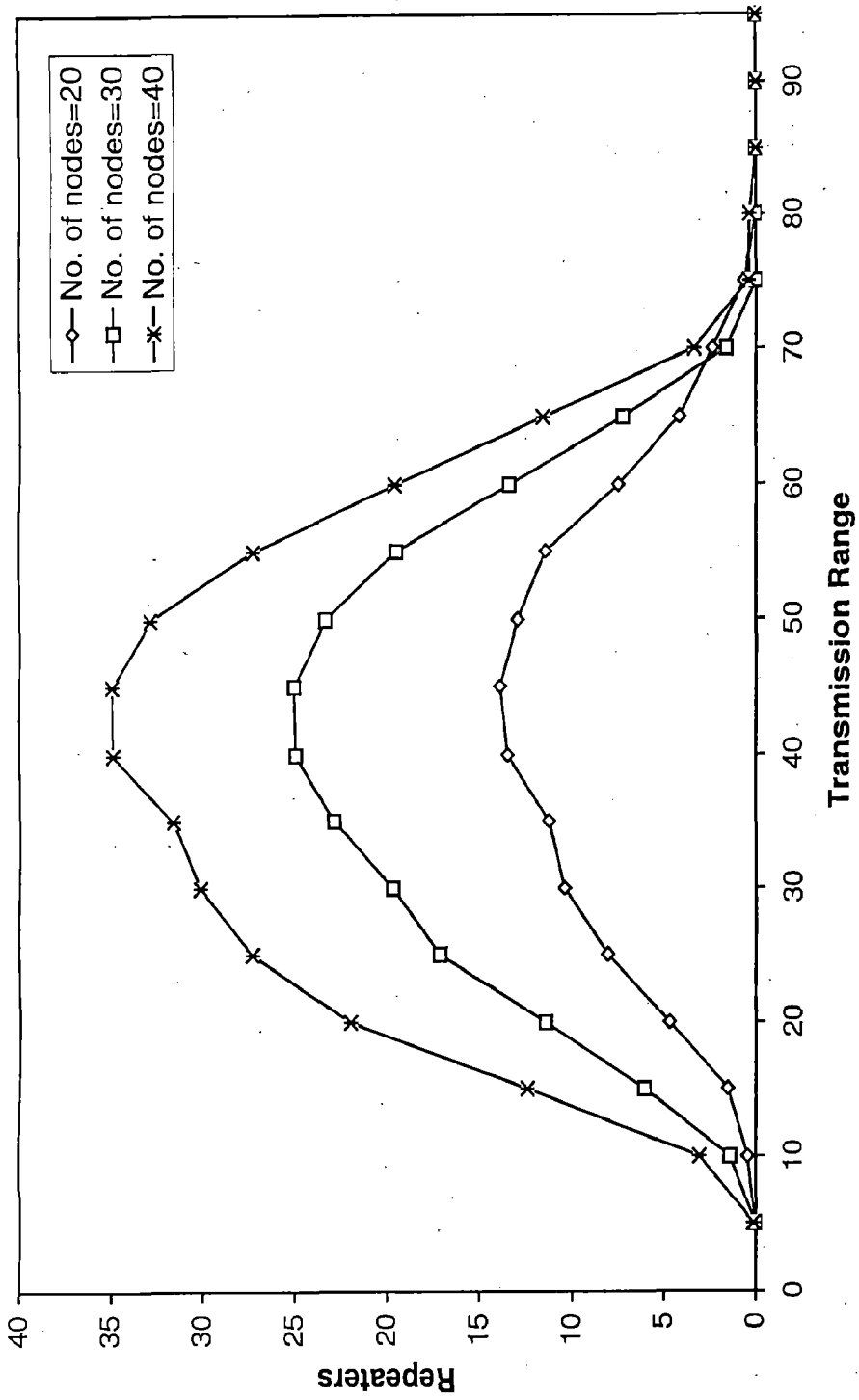


Fig. 6.2(a) Number of Repeaters (Clustering)

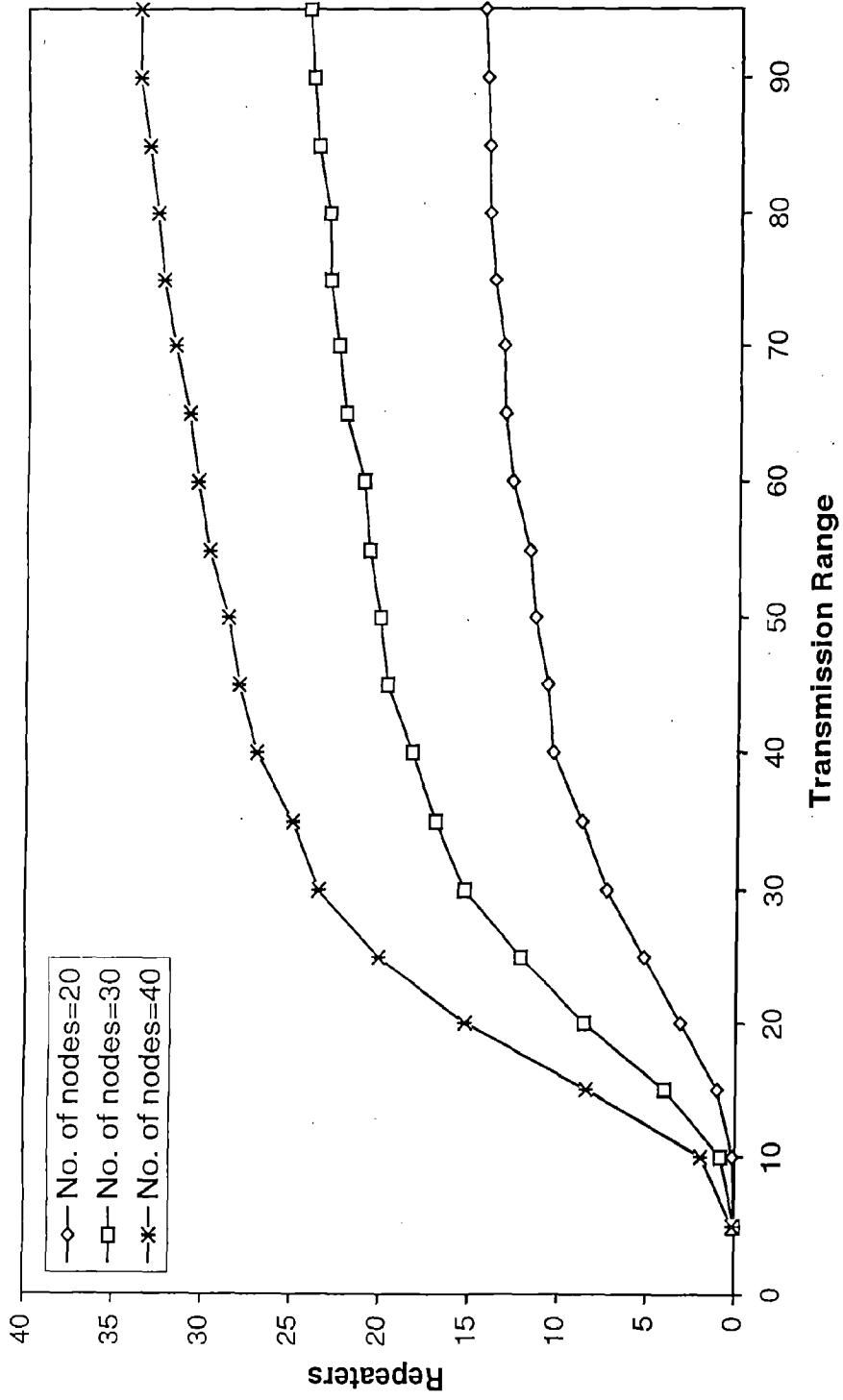


Fig. 6.2(b) Number of Repeaters (Modified Clustering)

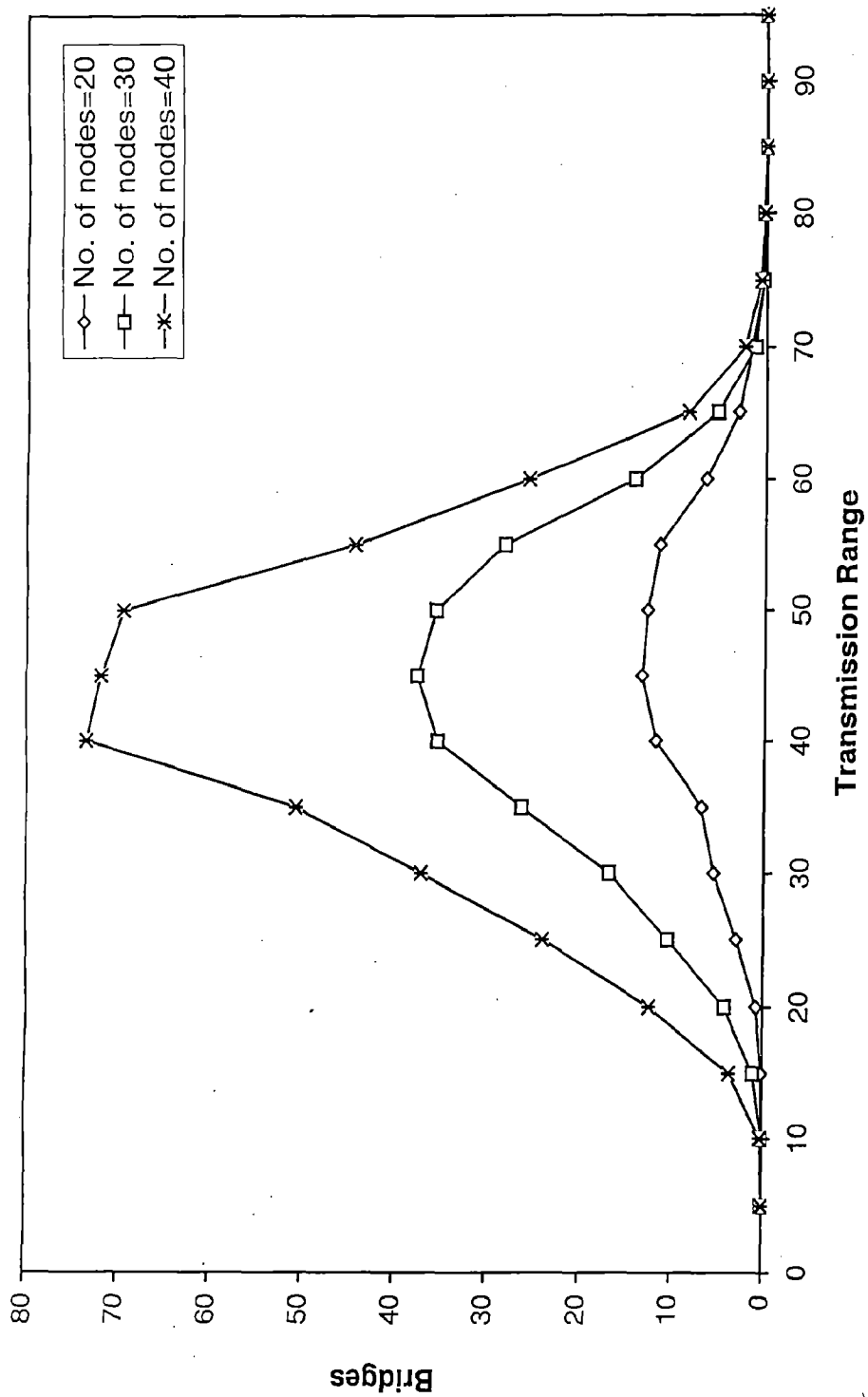


Fig. 6.3(a) Number of Bridges (Clustering)

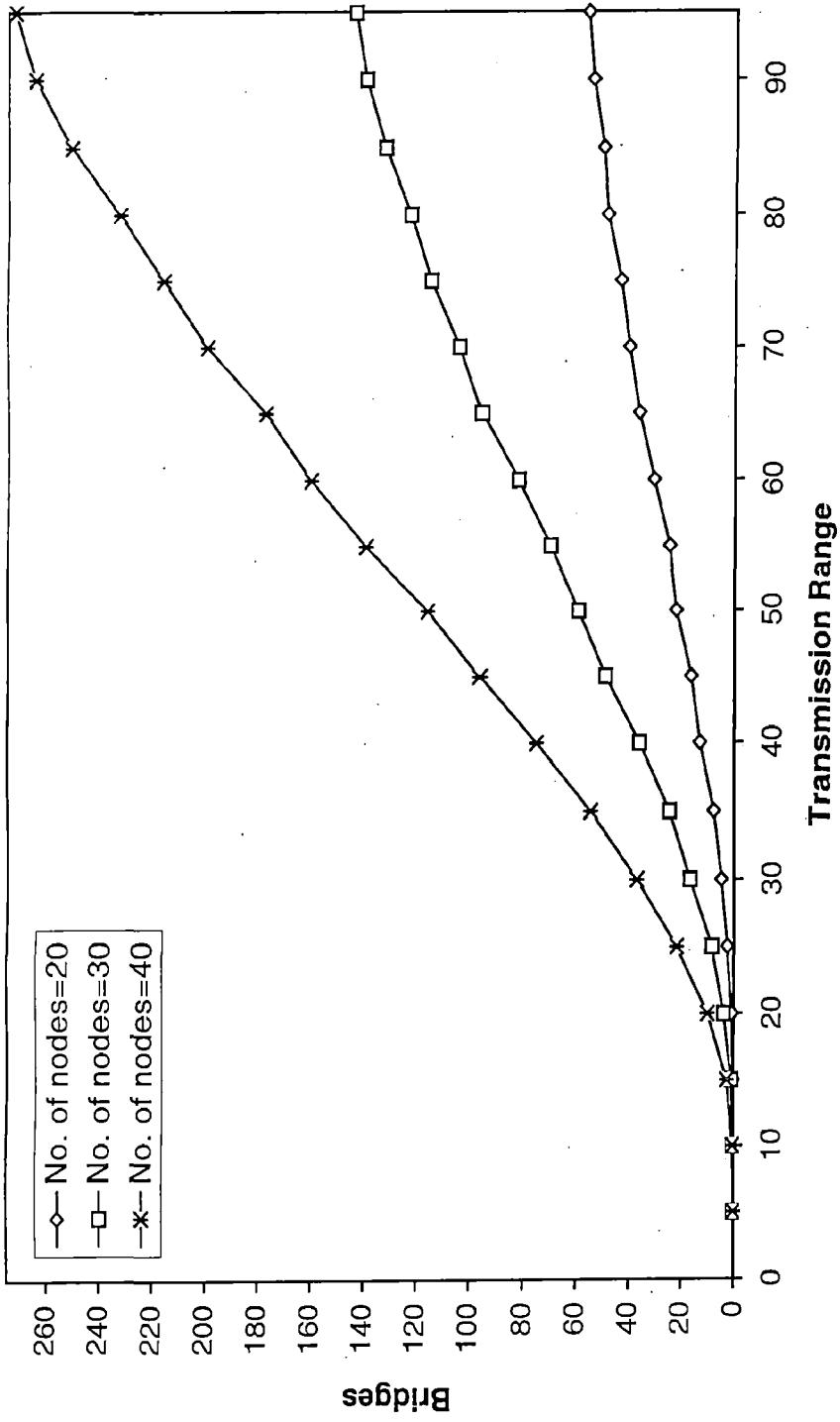


Fig. 6.3(b) Number of Bridges (Modified Clustering)

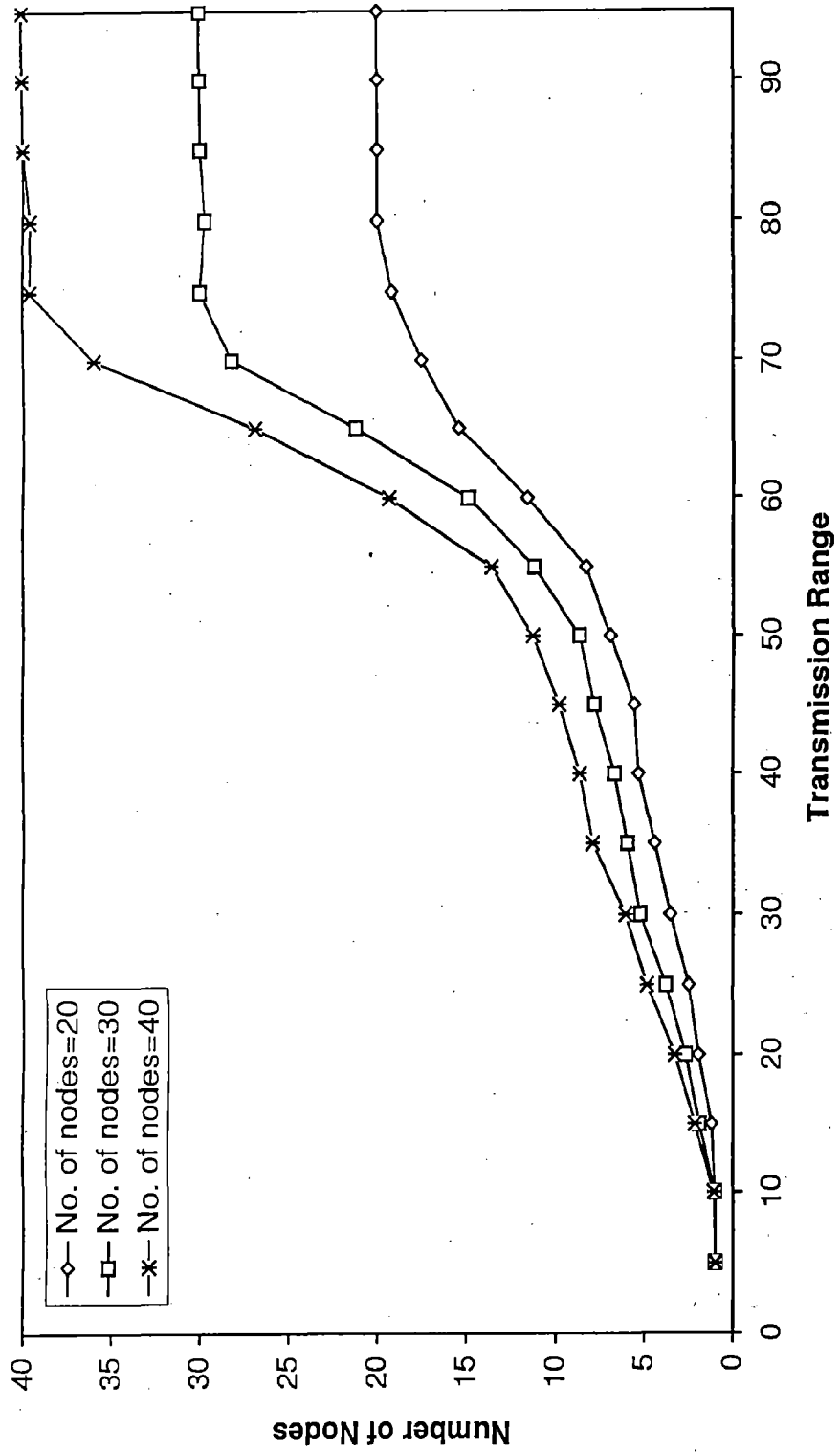


Fig. 6.4(a) Size of cluster (Clustering)

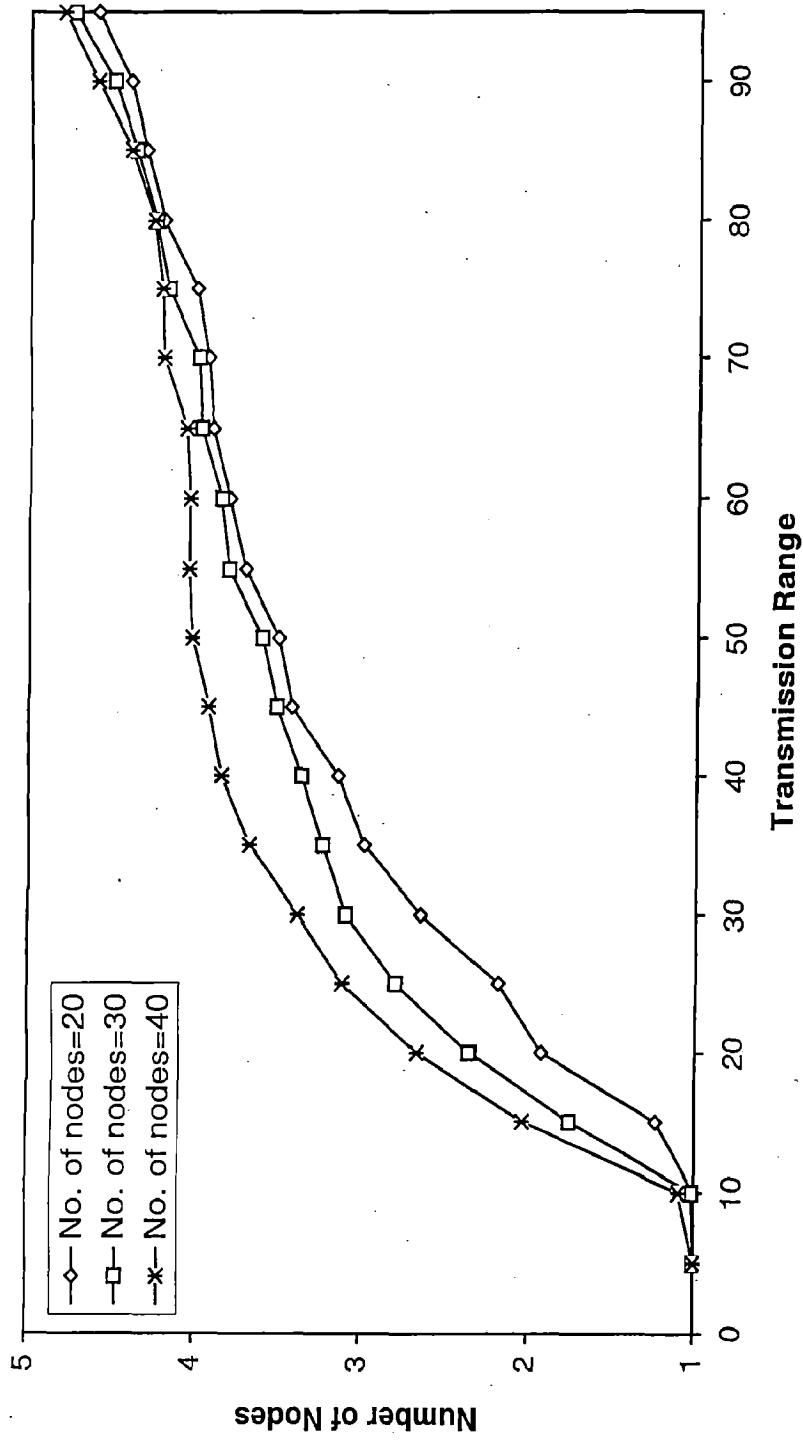


Fig. 6.4(b) Size of cluster (Modified Clustering)

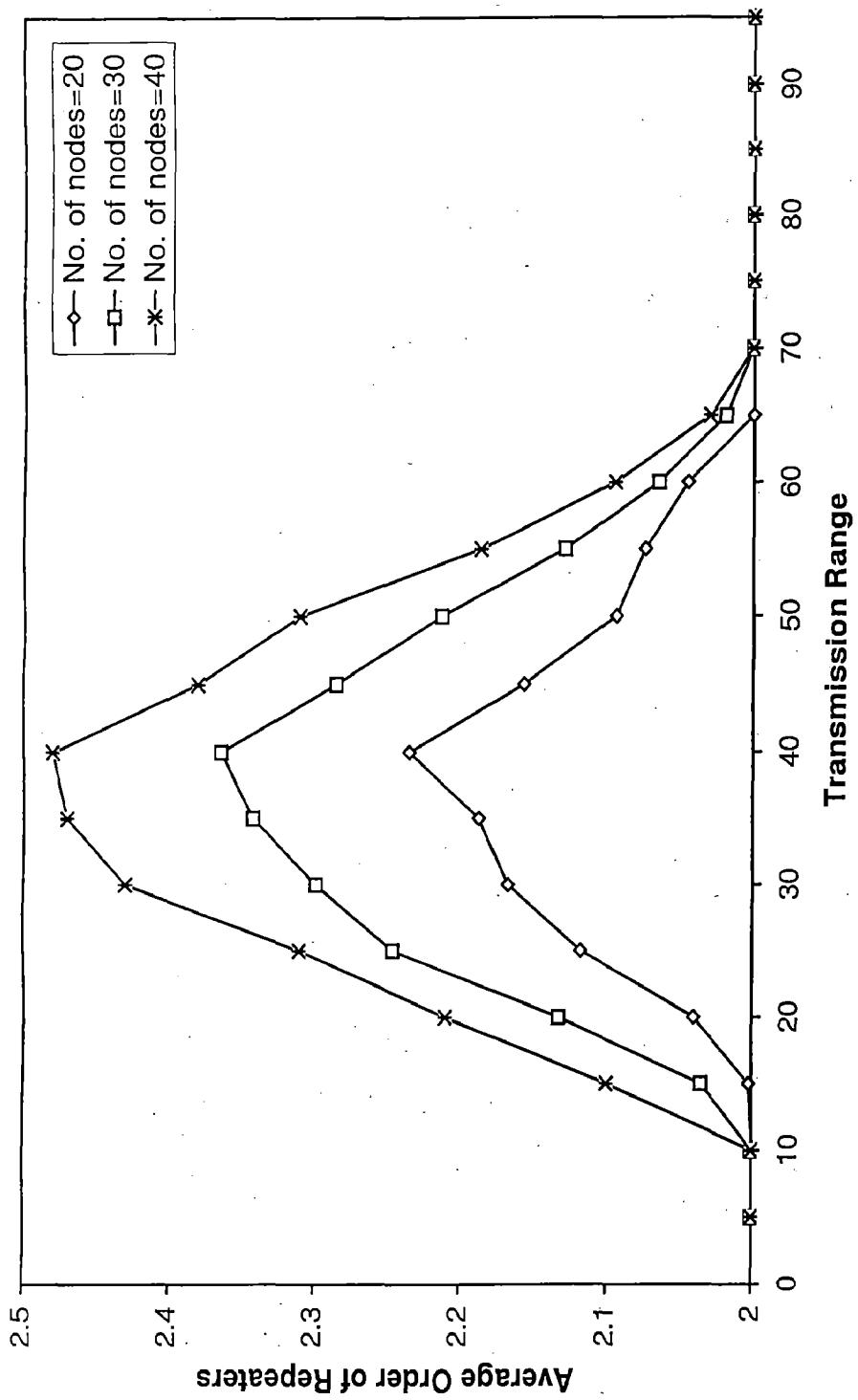


Fig. 6.5(a) Average Order of Repeater (Clustering)

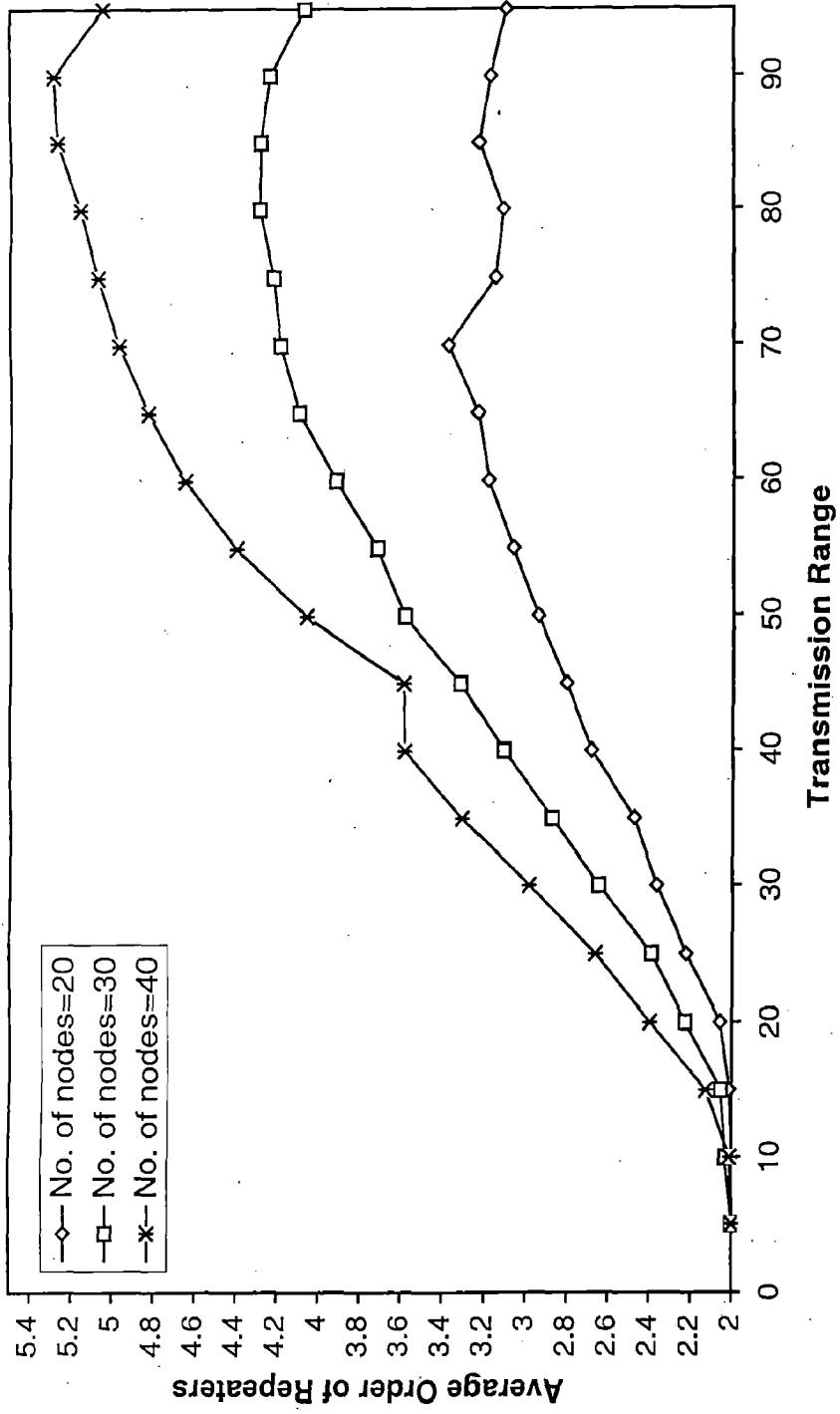


Fig. 6.5 (b) Average Order of Repeater (Modified Clustering)

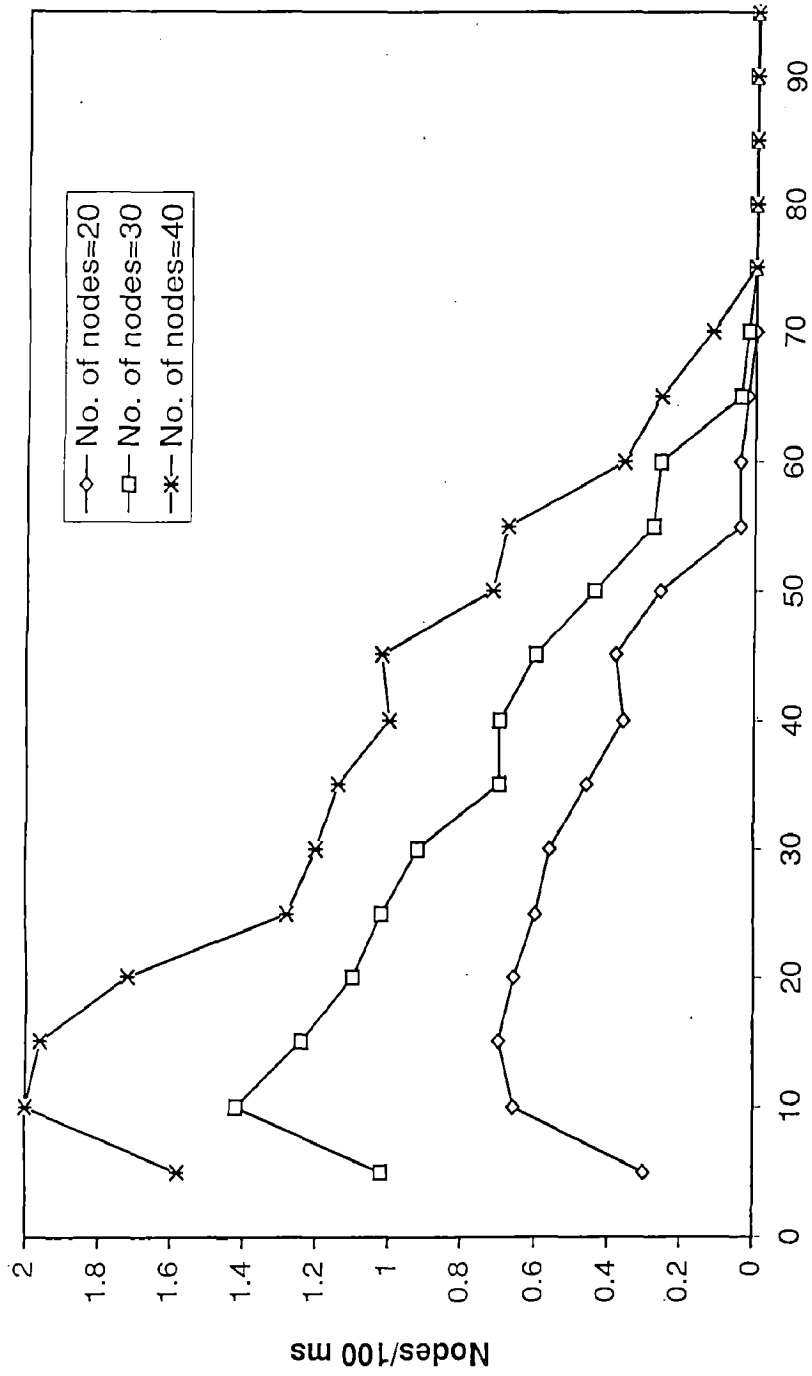


Fig. 6.6(a) Stability of the Multicluster Architecture (Clustering)

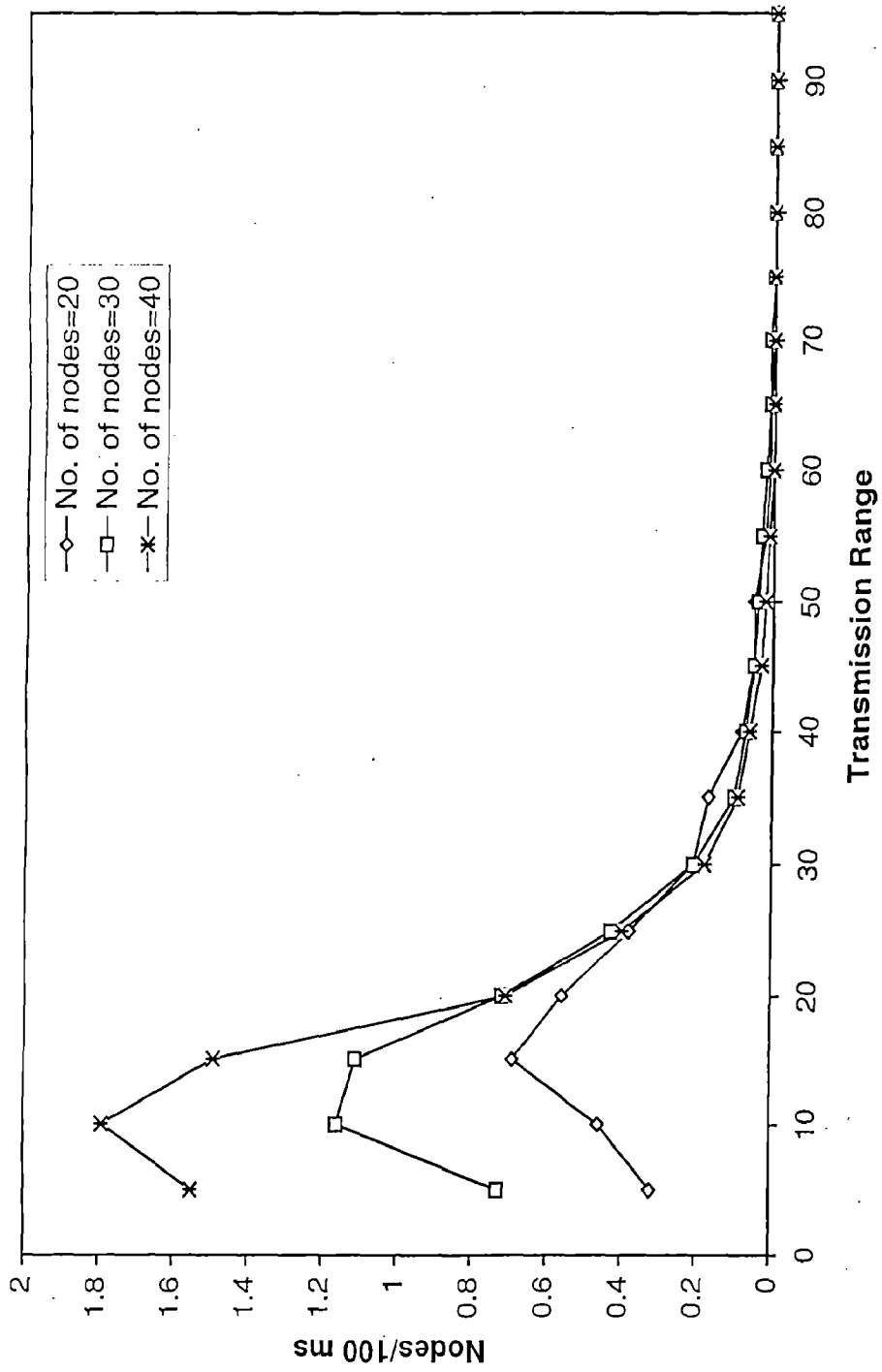


Fig.6.6(b) Stability of the Multicluster Architecture (Modified Clustering)

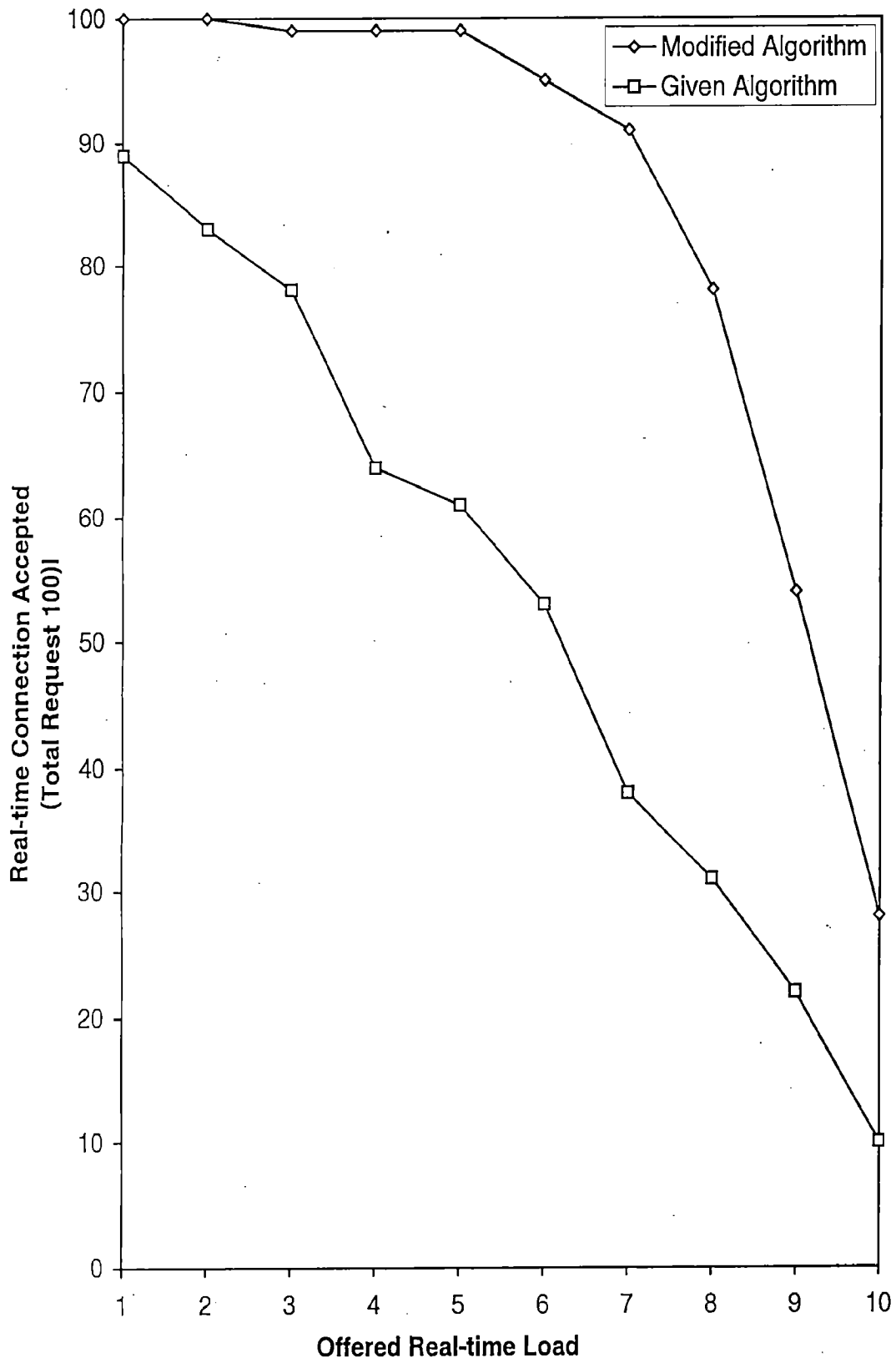


Fig. 6.7 Throughput of Mix Traffic

REFERENCES

1. C.R. Lin and M.Gerla, "Adaptive Clustering for Mobile Wireless Networks", IEEE J. Select. Areas Commun., pp.1265-1275, Sept. 1997.
2. D.J. Baker, J. Wieselthier, and A. Ephermides, "A Distributed Algorithm for Scheduling the Activation of Links in a Self Organizing, Mobile, Radio Network", in Proc. IEEE ICC'82, pp. F.6.1- 2F.6.5.
3. C.E. Perkins and P.Bhagwat, "Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers",in Proc. ACM SIGCOMM'94, pp. 234-244.
4. C.R.lin and M.Gerla, "A Distributed Architecture for Multimedia in a Multihop Dynamic Packet Radio Network", in Proc. IEEE GLOBECOM'95, Nov., 1995, pp. 1468-1472.
5. C.R. Lin and M. Gerla , "Asynchronous Multimedia Multihop Wireless Networks", in Proc. IEEE INFOCOM'97.
6. T. Hou and V. Li, "Transmission Range Control in Multihop Radio NETWORKs", IEEE Trans. Commun. Pp 38-44,Jan. 1986.
7. L. HU, "Topology Control for Multihop Packet Radio Networks", IEEE Trans. Commun., pp 1474-1481, Oct. 1993.
8. I. Chalmatac and S.S. Pinter, "Distributed Node Organization Algorithm for Channel Access in a Multihop Dynamic Radio Network", IEEE Trans. Comput., pp. 728-737, june 1987.
9. A. Alwan, R.Bagrodia, N. Bambos, M. Gerla, L. Kleinrock, J. short and J. Villasenor, "Adaptive Mobile Multimedia Networks", IEEE Personal Commun., pp 34-51, Apr 1996.

10. Andrew S. Tanenbaum, "Computer Networks", 3rd Edition, Chap. 4, Prentice Hall of India Private Limited, 1997.
11. C.R. Lin and M.Gerla , "Multimedia Transport in Multihop Dynamic Packet Radio Network", in Proc. IEEE INFOCOM'97.

APPENDIX A

```
/******APPENDIX INCLUDES ALL HEADER FILES******/
```

```
/******struct.h******/
```

```
struct node {
    int          no,mark,one_hop_nei,cluster_id,order;
    int          con_mark,avai_bw,temp_bw,cluster_member,m_mark;
    float        x,y;
    struct one_hop *one;
    struct node  *next;
};
typedef struct node *pnode;

struct one_hop {
    int          mark,no;
    struct one_hop *next;
    struct node  *member;
};
typedef struct one_hop *pone;

struct cluster {
    struct one_hop *another_cluster;
    struct cluster *next;
    int          total_member,cluster_id,cluster_size;
};
typedef struct cluster *pcluster;
pcluster  m;
int      cluster_no,repeaters;

struct real_traffic{
    int          source,desti,no_of_packet,selection,next_sess_packet;
    struct real_traffic *next;
};
typedef struct real_traffic *p_rtraffic;

struct hop {
    int          nodeno,hopno;
    struct hop  *next;
};
typedef struct hop *phop;

struct data_traffic {
    struct data_traffic *next;
};
```

```

                phop                dpath;
        };
typedef struct data_traffic *pdtraffic;

struct rem_path {
        phop                ppath;
        int                no_of_pack;
        struct rem_path *next;
};
typedef struct rem_path *p_rem_path;

struct con_node {
        pnode                pn;
        struct con_node *next;
};
typedef struct con_node *pcon;

struct backlog_data {
        phop                dpath;
        struct backlog_data *next;
};
typedef struct backlog_data *pback;
pback backlist;

pnode getcon_node() {
        pcon p;
        p=(pcon) malloc(sizeof(struct con_node));
        p->next=NULL;
        return(p);
}

pnode getnode() {
        pnode p;
        p=(pnode) malloc(sizeof(struct node));
        p->next=NULL;
        p->mark=NO;
        p->m_mark=NO;
        p->one=NULL;
        p->con_mark=NULL;
        p->cluster_member=NULL;
        return(p);
}

pcluster getcluster() {
        pcluster p;
        p=(pcluster) malloc(sizeof(struct cluster));

```

```

        p->next=NULL;
        return(p);
    }

pone getone_hop() {
    pone p;
    p=(pone) malloc(sizeof(struct one_hop));
    p->next=NULL;
    p->mark=NULL;
    return(p);
}

p_rtraffic getrtraffic() {
    p_rtraffic p;
    p=(p_rtraffic) malloc(sizeof(struct real_traffic));
    p->next=NULL;
    return(p);
}

pdtraffic getdatagram() {
    pdtraffic p;
    p=(pdtraffic) malloc(sizeof(struct data_traffic));
    p->next=NULL;
    return(p);
}

phop gethop() {
    phop p;
    p=(phop) malloc(sizeof(struct hop));
    p->next=NULL;
    return(p);
}

phop getbacklog() {
    pback p;
    p=(pback) malloc(sizeof(struct backlog_data));
    p->next=NULL;
    return(p);
}

p_rem_path getrem_path() {
    p_rem_path p;
    p=(p_rem_path) malloc(sizeof(struct rem_path));
    p->next=NULL;
    return(p);
}

```

```

int datagram;
    /***/un_rand.h***/

/**FUNCTION RETURNS UNIFORM RANDOM NUMBER BETWEEN 0 AND 1***/

#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define NTAB 32
#define NDIV (1+(IM-1)/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)
#define MEAN 1
long N =12345;
long *idum=&N;

float un_rand(long *idum)
{
    int j;
    long k;
    static long iy=0;
    static long iv[NTAB];
    float temp;

    if(*idum<=0||!iy){
        if(-(*idum)<1)
            *idum=1;
        else
            *idum=-(*idum);
        for(j=NTAB+7;j>=0;j--){
            k=(*idum)/IQ;
            *idum=IA*(*idum-k*IQ)-IR*k;
            if(*idum<0)
                *idum+=IM;
            if(j<NTAB)
                iv[j]=*idum;
        }
        iy=iv[0];
    }
    k=(*idum)/IQ;
    *idum=IA*(*idum-k*IQ)-IR*k;
    if(*idum<0)

```

```

    *idum+=IM;
j=iy/NDIV;
iy=iv[j];
iv[j]=*idum;
if((temp=AM*iy)>RNMX)
    return RNMX;
else
    return temp;
}

```

```

/*****poi_rand.h*****/

```

```

/*****FUNCTION RETURNS POISSON DISTRIBUTED RANDOM NUMBER *****/
/*****WITH MEAN ARRIVAL RATE ONE*****/

```

```

#define PI 3.141592

```

```

float poi_rand(float xm,long *idum)
{
    float  gammln(float xx);
    float  ran1(long *idum);
    static float sq,alxm,g,oldm=-1.0);
    float  em,t,y;

    if(xm<12.0)
    {
        if(xm!=oldm)
        {
            oldm=xm;
            g=exp(-xm);
        }
        em=-1;
        t=1.0;
        do{
            ++em;
            t*=un_rand(idum);
        }while(t>g);
    }
    else
    {
        if(xm!=oldm)
        {
            oldm=xm;
            sq=sqrt(2.0*xm);
            alxm=log(xm);
            g=xm*alxm-gammln(xm+1.0);

```

```

    }
do{
do{
    y=tan(PI*un_rand(idum));
    em=sq*y+xm;
    }while(em<0.0);
em=floor(em);
t=0.9*(1.0+y*y)*exp(em*alxm-gammln(em+1.0)-g);
    }while(un_rand(idum)>t);
    }
return(em);
}

```

```
float gammln(float xx)
```

```

{
    double      x,y,tmp,ser;
    static double cof[6]= {76.180091,-86.505,24.014,1.2317,0.120e-2,-0.539e-5};
    int          j;
    y=x-xx;
    tmp=x+5.5;
    tmp=(x+0.5)*log(tmp);
    ser=1.0;
    for(j=0;j<=5;j++) ser+=cof[j]/++y;
    return(-tmp+log(2.5066*ser/x));
}

```

```

/*****exp_rand.h*****/

```

```

/**FUNCTION RETURNS EXPONENTIAL RANDOM NUMBER WITH MEAN 3***/

```

```
float exp_rand(float xm,long *idum)
```

```

{
    float v,U;
    U=un_rand(idum);
    v=-xm*log(U);
    return v;
}

```

APPENDIX B

******PROGRAM TO FIND MINIMUN TRANSMISSION RANGE******
****WHERE EACH NODE CAN ACCESS ALL OTHER NODES ** /**

```
#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include "random.h"
#define NULL 0

main() {
    int i,n,g,h,pair_uncon,z,N,L;
    float a,b,d,range,avg_con,net_avg_con;
    pnode k,m,q,o;
    conlist l,c;
    n=20;
    do{
        printf("\n\n no of nodes are n=%d",n);
        range=4;
    do{
        net_avg_con=0;
        for(L=0;L<100;L++){
            k=NULL;
            pair_uncon=0;
        for(i=n;i>=1;i--)
            {
                m = (pnode) malloc(sizeof(struct node));
                m->no=i;
                m->x=ran(&seed)*100;
                m->y=ran(&seed)*100;
                m->next=k;
                k=m;
            }

        list=k;
    do{
        k=list;
        first=NULL;
        a=k->x;
        b=k->y;
        if(list!=NULL){
            do{
                k=list;
```

```

do{
    d=sqrt((k->x-a)*(k->x-a)+(k->y-b)*(k->y-b));
    if(d<range){
        if(first==NULL){
            p=(conlist) getponode();
            first=p;
            l=first;
            c=p;
        }
        else{
            p=(conlist) getponode();
            l->next=p;
            l=p;
        }
        l->t=k;

        if(list==k){
            k=k->next;
            list=k;
        }

        else{
            q->next=k->next;
            k=k->next;
        }
        }/*END OF IF*/

    else{
        q=k;
        k=k->next;
    }

}while(k!=NULL);

q=list;
c=c->next;
if(c!=NULL){
    o=c->t;
    a=o->x;
    b=o->y;
}

}while((c&&list)!=NULL);

g=count_con() ;
h=count_uncon() ;

```



```

z=g*h;
pair_uncon=g*h+pair_uncon ;
}/**END OF IF**/
    }while(list!=NULL);
N=n*(n-1)/2;
avg_con=(float)(N-pair_uncon)/N;
net_avg_con=(net_avg_con+avg_con);
    }/**END OF ITERATIONS*/
printf("\n range = %f",range);
printf("    the avg_con is = %f",net_avg_con/100);

/*****START OF ITERATIONS WITH NEW RANGE VALUE*****/
range=range + 2;
}while(range<=40);

/*****SRART OF ITERATIONS WITH NEW NUMBER OF NODES*****/
n=n+10;
}while(n<=40);
}

```

```

struct node {
    int      no;
    float    x,y;
    struct node *next;
};
typedef struct node *pnode;
pnode list;

struct ponode {
    pnode    t;
    struct ponode *next;
};
typedef struct ponode *conlist;
conlist first,p;

```

```

/*****FUNCTION COUNT THE CONNECTED COUNT*****/
int count_con(){
    int    count;

    count=0;
    while(first!=NULL){
        count+=1;
        first=first->next;
    }
    return(count);
}

```

```
}
```

```
*****FUNCTION COUNT THE UNCONNECTED NODES*****
```

```
int count_uncon(){
    int count;
    pnode f;

    count=0;
    f=list;
    while(f!=NULL){
        count+=1;
        f=f->next;
    }
    return(count);
}
```

```
conlist getponode() {
    p=(conlist) malloc(sizeof(struct ponode));
    p->next=NULL;
    return(p);
}
```

APPENDIX C

/******PROGRAM FOR VARIOUS CLUSTERING PARAMETER******/

```
#include<stdio.h>
#include<math.h>
#include<malloc.h>
#define NO 0
#define NULL 0
#define YES 1
#include"un_rand.h"
#include"exp_rand.h"
#include"poi_rand.h"
#include"struct.h"

main()
{
    int    range,eff_node,eff,total_repeaters,avg_bridges,clusters,max_cluster_size;
    int    a,b,n,c,q;
    float  count,total_order,avg_cluster_size;
    pnode  list;
    pcluster list_of_cluster;

    n=30;
    do{
        printf("NODES ARE %d\n",n);
        printf("Range Avg_cluster_size\t Bridges\t Repeaters\t Order\t Effectuated_node\n\n");
        max_cluster_size=5;
        range=40;
        do{
            avg_cluster_size=NULL;
            eff=NULL;
            total_order=NULL;
            total_repeaters=NULL;
            avg_bridges=NULL;
            for(q=0;q<100;q++){

                list=(pnode) random_deploy(n);
                one_hop_neighbour(list,range);
                list_of_cluster=cluster_form(list,max_cluster_size);
                count=count_order(list,range,n);
                avg_bridges+=count_bridge(list);

                if(repeaters==NULL){
```

```

        count=2;
        repeaters=1;
    }
else
    total_repeaters+=repeaters;
    total_order+=count/repeaters;
    clusters=list_of_cluster->total_member;
    avg_cluster_size+=total_of_cluster_size((pcluster) list_of_cluster)/clusters;
    random_re_deploy(list,n);
    count_diff_cluster(list_of_cluster,range);
    eff_node=effected_node(list);
    eff+=eff_node;
}
printf("%d\t%f\t%f\t%f\t%f\t%f\n\n",range,avg_cluster_size/100
,(float)avg_bridges/100,(float)total_repeaters/100,(float)total_order/100,(float)eff/100);
fflush(stdout);
range+=5;
}while(range<60);
n+=10;
}while(n<31);
}

/*****FUNCTION RANDOMLY PLACES N NODES IN 100*100 AREA*****/
pnode random_deploy(int n)
{
    pnode list,k,p;
    int i;

    p=(pnode) getnode();
    list=p;
    k=p;
    k->no =n;
    for(i=0;i<n;i++){
        p=(pnode) getnode();
        p->no =i;
        k->next =p;
        k=p;
        k->x=un_rand(idum)*100;
        k->y=un_rand(idum)*100;
    }

    return(list);
}

```

```
/******FUNCTION FINDS ONE -HOP NEIGHBORS OF ALL NODES******/
```

```
void one_hop_neighbour(pnode list,float range)
{
    pnode k,t;
    float a,b,d;
    pnode p,q;
    int count;
    k=list->next;
    do{
        count=0;
        t=k;
        a=k->x;
        b=k->y;
        q=NULL;
        k=list->next;

        do{
            d= sqrt((k->x-a)*(k->x-a)+(k->y-b)*(k->y-b));
            if((d<range)&&(d!=0))
            {
                p=(pnode) getone_hop();
                if(q==NULL)
                    t->one =p;
                else
                    q->next=p;
                q=p;
                q->member=k;

                q->no=k->no;
                count+=1;
            }

            k=k->next;
        }while(k!=NULL);

        k=t;

        k->one_hop_nei=count;
        k=k->next;
    }while(k!=NULL);
}
```

```

/*FUNCTION FINDS THE NEAREST NODE TO THE CENTRE NODE*****
***** OUT OF REMINING NODES*****/

```

```

find_min(pnode t)
{
    int    a,b;
    pnode  k,q;
    float  distance,min,d;
    pone   p,l;

    l=t->one;
    a=t->x;
    b=t->y;
    while((l->member)->cluster_member==YES)
    l=l->next;
    q=l->member;
    min=sqrt((q->x-a)*(q->x-a)+(q->y-b)*(q->y-b));
    p=l;

    do{
        l=l->next;
        if(l!=NULL)
        {
            while((l!=NULL)&&(l->member)->cluster_member==YES)
            l=l->next;
            if(l!=NULL)
            {
                q=l->member;
                d=sqrt((q->x-a)*(q->x-a)+(q->y-b)*(q->y-b));
                if(d<min)
                {
                    min=d;
                    p=l;
                }
            }
        }
    } while(l!=NULL);
    (p->member)->cluster_member=YES;
}

```

```

pone find_cluster_member(pnode t,int q)
{
    pone   y,k,l,m,a,b,templist,d,e,r;
    int    p,count,n;
    pnode  c;

```

```

k=NULL;
count=NULL;
p=q;
y=t->one;
t->cluster_member=YES;

do{
    find_min(t);
    q-=1;
    }while(q!=NULL);
q=p;
templist=NULL;
do{
    while((y->member)->cluster_member!=YES)
        {
            a=getone_hop();
            a->no=y->no;
            a->member=y->member;
            if(templist==NULL)
                {
                    templist=a;
                    b=a;
                }
            else{
                b->next=a;
                b=a;
            }

            y=y->next;
        }
m=y;
count++;

if(k==NULL)
    {
        k=y;
        l=k;
    }
else
    {
        l->next=y;
        l=y;
    }
if(y!=NULL)
y=y->next;

```

```

}while((y!=NULL)&&(count!=q));
    while(y!=NULL){
        a=getone_hop();
        a->no=y->no;
        a->member=y->member;
        if(templist==NULL)
            {
                templist=a;
                b=a;
            }
        else{
            b->next=a;
            b=a;
        }

        y=y->next;
    }

    m->next=NULL;
    r=(pone) modify_cluster(k,t);
    while(templist!=NULL)
        {
            c=templist->member;
            d=c->one;
            e=(pone) modify_list2(r,d);
            c->one=e;
            if(e!=NULL)
                {
                    n=count_member(e);
                    c->one_hop_nei=count;
                }
            else
                c->one_hop_nei=NULL;
            templist=templist->next;
        }
    return(k);
}

/*****FUNCTION FORMS THE CLUSTER OF NEAREST NODES*****/
pcluster cluster_form(pnode list)
{
    int    e,max,n,q,max_cluster_size;
    pnode  k,t,s,c;
    pone   x,y,z,u,f,l;
    pcluster  p,list_of_cluster;

```



```

cluster_no=NULL;
p= (pcluster) getcluster();
list_of_cluster=p;
p->cluster_id=cluster_no;
m=p;
max_cluster_size=5;

do{
    max=NULL;
    k=list->next;
    while(k->mark==YES)
    k=k->next;
    if((k->mark==NO)&&(k->one_hop_nei==NULL))
    {
        k->mark=YES;
        list->no-=1;
        l=getone_hop();
        l->member=k;
        make_cluster((pone) l,(pcluster) list_of_cluster);
        l->no=k->no;
    }
    else{
        do{
            if((k->mark==NO)&&(k->one_hop_nei>max))
            {
                max=k->one_hop_nei;
                t=k;
            }
            k=k->next;
        }while(k!=NULL);
        e=count_cl_member(t->one);
        q=max_cluster_size-1;
        if(e<=q)
        {
            list->no=list->no-e-1;
            y=t->one;
        }
        else
        {
            y=t->one;
            y=find_cluster_member(t,q);
            list->no=list->no-q-1;
        }
        t->mark=YES;
        mark_list((pone) y);
        f = (pone) modify_cluster((pone) y,(pnode) t);
    }
}

```

```

make_cluster((pone) f,(pcluster) list_of_cluster);

do{
    s=y->member;
    x=s->one;
    if(x!=NULL){
        x=(pone) modify_list2((pone) f,(pone) x);
        z=x;
        s->one=x;

        while(z!=NULL)
        {
            c=z->member;
            u=c->one;
            u=(pone) modify_list2((pone) f,(pone) u);
            c->one=u;
            if(u!=NULL){
                n=count_member((pone) u);
                c->one_hop_nei=n;
            }
            else
            c->one_hop_nei=NULL;
            z=z->next;
        }
    }

    y=y->next;
}while(y!=NULL);
}while(list->no>NULL);
return(list_of_cluster);
}

```

```

void mark_list(pone y) {
    pnode o;
    do{
        o=y->member;
        o->mark=YES;
        y=y->next;
    }while(y!=NULL);
}

```

```

/*****FUNCTION ADD THE CENTRE NODE TO ITS ONE-HOP NEIGHBOR*****/
*****TO FORM THE CLUSTER*****/
pone modify_cluster(pone y,pnode t) {

```

```

    pone p;
    p=(pone) getone_hop();
    p->next=y;
    y=p;
    y->member=t;
    y->no=t->no;
    return(y);
}

```

```

/*****FUNCTION MODIFIES ONE - HOP LIST AFTER CLUSTER FORMATION*****/

```

```

pone modify_list2(pone pt1,pone pt2) {
    pone k,y,x;
    k=y=pt2;
    if(y!=NULL)
    {
        do{
            x=pt1;
            while((x!=NULL)&&(y->member!=x->member))
                x=x->next;
            if(x!=NULL)
            {
                if(y==pt2){
                    pt2=pt2->next;
                    y=pt2;
                }
                else{
                    k->next=y->next;
                    y=y->next;
                }
            }
        }
        else{
            k=y;
            y=y->next;
        }
    }while(y!=NULL);
}
return(pt2);
}

```

```

make_cluster(pone f,pcluster list_of_cluster)
{
    pcluster p;
    p=getcluster();
    list_of_cluster->total_member+=1;
    m->next=p;
    m=p;
    m->another_cluster=f;
    ++cluster_no;
    m->cluster_size=(f->member)->one_hop_nei+1;
    p->cluster_id=cluster_no;
    put_cluster_no((pone) f);
}

```

```

count_member(pone u)
{
    int count;
    count=0;
    do{
        count+=1;
        u=u->next;
    }while(u!=NULL);
    return(count);
}

```

```

put_cluster_no(pone f)
{
    while(f!=NULL){
        (f->member)->cluster_id=cluster_no;
        f=f->next;
    }
}

```

*****FUNCTION RE-DEPLOY NODES AFTER THEIR RANDOM MOVEMENT*****/

```

random_re_deploy(pnode list,int n)
{
    pnode k;
    int i;

    k=list->next;
    for(i=1;i<=n;i++){
        k->x=k->x+(3*un_rand(idum))*cosf(2*3.1415*un_rand(idum));
        k->y=k->y+(3*un_rand(idum))*sinf(2*3.1415*un_rand(idum));
    }
}

```

```

        k=k->next;
    }
}

count_diff_cluster(pcluster list_of_cluster,int range)
{
    pcluster k;
    pone o,m,save_node;
    int save,new_nei;

    k=list_of_cluster->next;

do{

    if(k->cluster_size==1)
    {
        o=k->another_cluster;
        m_mark_node((pone) o);
    }

    else{
        o=k->another_cluster;
        m=o;
        new_nei=node_new_nei((pone) o,(pone) m,(int) range);

        if(new_nei==k->cluster_size)
            m_mark_node((pone) o);

        else{
            save=NULL;

            do{
                new_nei=node_new_nei((pone) o,(pone) m,(int) range);

                if(new_nei>save)
                {
                    save=new_nei;
                    save_node=o;
                }

                o=o->next;
            }while(o!=NULL);
            now_m_mark_node((pone) save_node,(pone) m,(int) range);
        }
    }
}

```

```

    }
    k=k->next;
  }while(k!=NULL);
}

```

/*FUNCTION MARK ALL THE WHICH GET EFFECTED DUE TO MOVEMENT **/

```

m_mark_node(pone p)
{
  pnode t;

  do{
    t=p->member;
    t->m_mark=YES;
    p=p->next;
    while(p!=NULL);
  }
}

```

```

node_new_nei(pone o,pone m,int range)
{
  pnode t,p;
  float a,b;
  int count;

  count=NULL;
  t=o->member;
  a=t->x;
  b=t->y;
  do{
    p=m->member;
    if(sqrt((p->x-a)*(p->x-a)+(p->y-b)*(p->y-b))<=(float) range) count++;
    m=m->next;
  }while(m!=NULL);
  return(count);
}

```

```

now_m_mark_node(pone o,pone m,int range)
{
  pnode t,p;
  float a,b;

  t=o->member;
  a=t->x;
  b=t->y;
  do{
    p=m->member;

```

```
        if(sqrt((p->x-a)*(p->x-a)+(p->y-b)*(p->y-b))<=(float) range) p->m_mark=YES;
        m=m->next;
    } while(m!=NULL);
}
```

```
    /*****FUNCTION COUNT THE EFFECTED NODES*****/
effectd_node(pnode list)
{
    pnode k;
    int count;
    count=NULL;

    k=list->next;

    do{
        if(k->m_mark==NO) count++;

        k=k->next;
    } while(k!=NULL);

    return(count);
}
```

APPENDIX D

/PROGRAM FOR REAL-TIME SESSION AND DATAGRAM THROUGHPUT****

```
#include<stdio.h>
#include<math.h>
#include<malloc.h>
#define NO 0
#define NULL 0
#include"un_rand.h"
#include"exp_rand.h"
#include"poi_rand.h"
#include"struct.h"
#define YES 1
#define FOUND 1
#define NOTFOUND 0
#define NOPATH 0
#define DISCARDED 0

main()
{
    int          time,discarded,real_packet_dispatched,cycle_time,connected,before_new,
    int          total_packet,range,eff_node,eff;
    int          a,b,n,c,q,nullcount,pathcount,totaldatagram,packet,data,ti,null,pah,poi_me
    float        data_gram_rate,x,data_gram_in_cycle,exp_mean;
    pnode        list;
    pcluster     list_of_cluster;
    p_rtraffic   rtraffic,save;
    p_rem_path   path_list;
    phop         path;
    pback        backlist;

    range=5;
    c=1;
    do{
        printf("c is %d\n",c);
        printf("Rejected\tAccepted\tDatagram\tTime\n\n");
        eff=NULL;
        null=NULL;
        data=NULL;
        ti=NULL;
        pah=NULL;
        for(q=0;q<10;q++){
```



```

time=NULL;
datagram=NULL;
nullcount=0;
pathcount=0;
n=20;
exp_mean=3.000;
poi_mean=1;
cycle_time=100*c;
data_gram_rate=(float)1/100;
x=data_gram_in_cycle=cycle_time*data_gram_rate;
path_list=NULL;
backlist=NULL;

list=(pnode) random_deploy(n);
one_hop_neighbour(list,range);
list_of_cluster=cluster_form(list);
one_hop_neighbour(list,range);
set_initial_bw(list_of_cluster,list,cycle_time);
rtraffic=(p_rtraffic);
real_traffic_generation(exp_mean,poi_mean,n,cycle_time);

save=rtraffic;
do{
do{
a=rtraffic->source;
b=rtraffic->desti;
connected=con_check(list,a,b,range);

if(connected!=1)
{
rtraffic->selection=DISCARDED;
before_new_sess=rtraffic->next_sess_packet;
packet=x*before_new_sess;
backlist=send_backlog(backlist,list);
backlist=send_datagram(packet,n,backlist,list);
time+=packet;
modify_path_list(path_list,before_new_sess,list);
rtraffic=rtraffic->next;
nullcount++;
}
else
{
path=find_path(a,b,list);
if(path==NULL)
{
rtraffic->selection=DISCARDED;

```

```

        before_new_sess=rtraffic->next_sess_packet;
        packet=x*before_new_sess;
        backlist=send_backlog(backlist,list);
        backlist=send_datagram(packet,n,backlist,list);
        time+=packet;
        rtraffic=rtraffic->next;
        nullcount++;
    }
else
{
    pathcount++;
    set_avai_bw(path,list);
}
}
} while(!(connected&&path)&&(rtraffic));
if(rtraffic!=NULL)
{
    before_new_sess=rtraffic->next_sess_packet;
    total_packet=rtraffic->no_of_packet;

    if(before_new_sess>total_packet)
    {
        free_bw(path,list);
        if(path_list!=NULL)
            modify_path_list(path_list,before_new_sess,list);
        packet=x*before_new_sess;
        backlist=send_backlog(backlist,list);
        backlist=send_datagram(packet,n,backlist,list);
        time+=packet;
    }
else
{
    packet=x*before_new_sess;
    backlist=send_backlog(backlist,list);
    backlist=send_datagram(packet,n,backlist,list);
    time+=packet;
    path_list=(p_rem_path) add_in_path_list(path,path_list,total_packet);
    path_list=(p_rem_path) modify_path_list(path_list,before_new_sess,l
}
    rtraffic=rtraffic->next;
}
} while(rtraffic!=NULL);
printf("%d\t%d\t%d\t%d\t\n",nullcount,pathcount,datagram,time);
null+=nullcount;
pah+=pathcount;
data+=datagram;

```

```

    ti+=time;
    random_re_deploy(list,n);
    count_diff_cluster(list_of_cluster,range);
    eff_node=effected_node(list);
    eff+=eff_node;
    fflush(stdout);
}
printf("\nAvg rejected  Avg accepted  Avg datagram  Avg time\n");
printf("%d\t\t%d\t\t%d\t\t%d\t\t\n",null/10,pah/10,data/10,ti/10);
printf("\n");
c=c+1;
}while(c<2);
}

```

/*****FUNCTION CHECKS WHETHER THE SOURCE*****
 ***** AND THE DESTINATION ARE CONNECTED*****/

```

con_check(pnode list,int first,int sec,int range)

```

```

{
    pnode k,t,savel;
    pcon p,c,l;
    float a,b,d;

    k=list->next;
    savel=k;
    t=k;

    do{
        if(t->no!=first)
            t=t->next;
        }while(t!=NULL&& t->no!=first);
    t->con_mark=YES;
    p=(pcon) getcon_node();
    c=p;
    l=c;
    p->pn=t;

    do{
        k=savel;
        t=c->pn;
        a=t->x;
        b=t->y;

    do{

```

```

while((k!=NULL)&&(k->con_mark==YES))
k=k->next;
if(k!=NULL)
{
    d=sqrt((k->x-a)*(k->x-a)+(k->y-b)*(k->y-b));

    if(d<range&&k->no==sec)
    {
        make_all_NO((pnode) save1);
        return(FOUND);
    }
    else if(d<range)
    {
        p=(pcon) getcon_node();
        l->next=p;
        p->pn=k;
        k->con_mark=YES;
        l=p;
    }
    k=k->next;
}
}while(k!=NULL);

c=c->next;
}while(c!=NULL);

if(c==NULL)
{
    make_all_NO((pnode) save1);
    return(NOTFOUND);
}

make_all_NO(pnode save1)
{
    while(save1!=NULL)
    {
        save1->con_mark=NO;
        save1=save1->next;
    }
}

```

/******FUNCTION PREPARE THE LIST OF REAL TIME SESSION******/

```

p_rtraffic real_traffic_generation(float exp_mean,int poi_mean,int n,int cycletime)
{

```

```

int      i,t,k;
p_rtraffic  p,rtraffic,l;

rtraffic=NULL;
for(i=0;i<100;i++)
{
    t=0;
    k=0;
    p=getrtraffic();

    if(rtraffic==NULL)
    {
        rtraffic=p;
        l=p;
    }
    else
    {
        l->next=p;
        l=p;
    }
    do{
        p->source=un_rand(idum)*(n-1);
        p->desti=un_rand(idum)*(n-1);
    } while(p->source==p->desti);
    p->no_of_packet=(60*1000.0/cycletime)*exp_rand(exp_mean,idum);
    while(t==0)
    {
        t=poi_rand(poi_mean,idum);
        if(t==0)
            k+=(1000.0/cycletime);
    }
    if(t!=0)
        p->next_sess_packet=k+(1000.0/cycletime)/t;
}
return(rtraffic);
}

```

/******FUNCTION RETURNS THE POINTER OF THE NUMBERD NODE*****/

```

pnode get_source_poi(pnode list,int source)
{
    pnode k;

```

```

k=list->next;
while(k->no!=source)
k=k->next;
return(k);
}

```

/**FUNCTION RETURNS ONE IF DESTINATION IS REACHED ELSE ZERO**/

```

check_path(phop l,int desti)
{
    while((l!=NULL)&&(l->nodeno!=desti))
    l=l->next;
    if(l!=NULL)
    return(1);
    else
    return(0);
}

```

/**FUNCTION RETURN THE NUMBER OF HOPS IN THE PATH***/

```

path_hops(phop hoplist,int desti)
{
    phop k;
    k=hoplist;
    while(k->nodeno!=desti)
    k=k->next;
    return(k->hopno);
}

```

phop make_path_list(phop hoplist,pnode k,int total_hop,phop t)

```

{
    pone a;
    phop x,p;

    a=k->one;
    do{
        x=hoplist;
        while((x!=NULL)&&(a->no!=x->nodeno))
        x=x->next;
        if((x!=NULL)&&(x->hopno==total_hop))
        {
            p=(phop) gethop();
            t->next=p;
            p->hopno=total_hop;
            p->nodeno=x->nodeno;
        }
    }
}

```

```

        t=p;
        return(t);
    }
    a=a->next;
    }while(a!=NULL);
return(t);
}

```

```

phop get_path(pnode list,phop hoplist,int desti)

```

```

{
    int    total_hop;
    phop  path,t,p;
    pnode k;

    path=NULL;

    total_hop=path_hops(hoplist,desti);
    p=(phop) gethop();

    if(path==NULL)
    {
        path=p;
        t=p;
        p->nodeno=desti;
        p->hopno=total_hop;
    }

    do{
        total_hop-=1;
        k=get_source_poi(list,t->nodeno);
        t=make_path_list(hoplist,k,total_hop,t);
    }while(total_hop!=NULL);

    return(path);
}

```

```

add_member(pone t,phop l,phop hoplist)

```

```

{
    int    hno;
    phop  p,m,i;

    hno=l->hopno;

    do{
        m=l;

```

```

        l=l->next;
    }while(l!=NULL);

do{
    i=hoplist;
    while((i!=NULL)&&(t->no!=i->nodeno))
        i=i->next;

    if((i==NULL)&&((t->member)->avai_bw!=NULL))
        {
            p=(phop) gethop();
            m->next=p;
            m=p;
            m->nodeno=t->no;
            m->hopno=hno+1;
        }
    t=t->next;
}while(t!=NULL);
}

```

/*******FUNCTION RETURNS THE POINTER TO THE PATH*****
 *****BETWEEN THE GIVEN SOURCE AND DESTINATION*****/

```

phop find_path(int source,int desti,pnode list)
{
    pnode k;
    pone t;
    phop p,hoplist,l,m,path;
    int get;

    hoplist=NULL;

    p=gethop();
    if(hoplist==NULL)
        {
            hoplist=p;
            l=p;
            p->nodeno=source;
            p->hopno=NULL;
        }
    k=(pnode) get_source_poi(list,source);
    if(k->avai_bw==NULL)
        return(NOPATH);
    t=k->one;

```



```

do{
    if(t!=NULL)
        {
            add_member(t,l,hoplist);
            get=check_path(l,desti);
            if(get==1)
                break;
            else
                l=l->next;
        }
    else
        l=l->next;

    if(l!=NULL)
        {
            k=get_source_poi(list,l->nodeno);
            t=k->one;
        }
}while(l!=NULL);

if(get==1)
{
    path=(phop) get_path(list,hoplist,desti);
    return(path);
}
else
return(NOPATH);
}

set_avai_bw(phop path ,pnode list)
{
    pnode k,t;
    k=list->next;

    t=k;

do{
    t=k;
    while(t->no!=path->nodeno)
        t=t->next;
    if(t->avai_bw>0)
        t->avai_bw -=1;
    path=path->next;
}while(path!=NULL);
}

```

```
/******FUNCTION REMOVES THE POINTER OF PATH FROM THE PENDING LIST ***  
*****IF ITS SESSION IS COMPLETED******/
```

```
p_rem_path modify_path_list(p_rem_path path_list,int no,pnode list)  
{  
    p_rem_path k,t;  
  
    t=k=path_list;  
  
    if(path_list!=NULL)  
    {  
        do{  
            k->no_of_pack-=no;  
            if((k!=NULL)&&(k->no_of_pack<1))  
            {  
                free_bw(k->ppath,list);  
  
                if(k==path_list)  
                {  
                    k=k->next;  
                    t=k;  
                    path_list=k;  
                }  
            }  
            else  
            {  
                t->next=k->next;  
                k=k->next;  
            }  
        }  
    }  
    else if(k!=NULL)  
    {  
        t=k;  
        k=k->next;  
    }  
    }while(k!=NULL);  
}  
return(path_list);  
}
```

```
/******FUNCTION ADD THE POINTER TO PATH IN PATHLIST*****  
***** IF ITS PACKETS ARE PENDING******/
```

```
p_rem_path add_in_path_list(phop path,p_rem_path path_list,int total_packet)  
{
```

```
p_rem_path p;

p=getrem_path();
p->ppath=path;
p->no_of_pack=total_packet;
```

```
if(path_list==NULL)
path_list=p;
else
{
p->next=path_list;
path_list=p;
}
return(path_list);
}
```

```
*****FUNCTION FREE BANDWIDTH ALONG ALL THE NODES OCCUPIED*****
***** WHERE SLOTS WERE RESERVED FOR THE JUST COMPLETED SESSION**/
```

```
free_bw(phop path,pnode list)
{
pnode k,t;
k=list->next;
t=k;
do{
t=k;
while((path->nodeno)!=(t->no))
t=t->next;
t->avai_bw+=1;
path=path->next;
}while(path!=NULL);
}
```

```
count_cl_member(pone o)
{
int count;
count= NULL;
while(o!=NULL)
{
count++;
o=o->next;
}
return(count);
}
```

```
/******FUNCTION ASSIGN THE INITIAL BANDWIDTH TO ALLTHE *****  
NODES DEPENDING ON THE CLUSTER'S NODE STRENGTH TO WHICH THEY BELONG/
```

```
set_initial_bw(pcluster list_of_cluster,pnode list,int cycle)
```

```
{  
    pone    o;  
    pcluster k;  
    pnode   t,p;  
    int     member,bw;  
  
    k=list_of_cluster->next;  
    t=list->next;  
    p=t;  
    do{  
        o=k->another_cluster;  
        member=count_cl_member(o);  
        bw=(cycle)/(5*member);  
        do{  
            t=p;  
            while(o->no!=t->no)  
                t=t->next;  
            t->avai_bw=bw;  
            o=o->next;  
        }while(o!=NULL);  
        k=k->next;  
    }while(k!=NULL);  
}
```

```
set_bw_original(pnode list)
```

```
{  
    pnode t;  
    t=list->next;  
    while(t!=NULL)  
    {  
        t->avai_bw=t->temp_bw;  
        t=t->next;  
    }  
}
```

```
set_bw_infinite(int n,pnode list)
```

```
{  
    pnode t;  
    t=list->next;  
    while(t!=NULL)  
    {  
        t->temp_bw=t->avai_bw;
```

```

    t->avai_bw=n;
    t=t->next;
}
}

```

/******FUNCTION GENERATE THE DATAGRAM OF FIXED SIZE******/

```

pdtraffic datagram_traffic_generation(int no,int n,pnode list)

```

```

{
    pdtraffic  dtraffic, p,l;
    int        i,source,desti;

    dtraffic=NULL;
    for(i=0;i<no;i++)
    {
        p=getdatagram();
        if(dtraffic==NULL)
        {
            dtraffic=p;
            l=p;
        }
        else
        {
            l->next=p;
            l=p;
        }
        do{
            source=un_rand(idum)*(n-1);
            desti=un_rand(idum)*(n-1);
        }while(source==desti);

        p->dpath=find_path(source,desti,list);
    }
    return(dtraffic);
}

```

```

is_bw_avai(phop k,pnode list)

```

```

{
    int    n;
    pnode t;

    n=k->nodeno;
    t=(pnode) get_source_poi(list,n);
    return(t->avai_bw);
}

```

```
*****FUNCTION SEND THE DATAGRAM WAITING IN THE QUEUE*****/
```

```
pback send_backlog(pback backlist,pnode list)
```

```
{  
    pback k,t;  
    phop m,l;  
    int n;  
  
    if(backlist!=NULL)  
    {  
        k=backlist;  
        t=k;  
        do  
        {  
            m=k->dpath;  
            do  
            {  
                n=is_bw_avai(m,list);  
  
                if(n>=1)  
                {  
                    l=m;  
                    m=m->next;  
                    free(l);  
                }  
                else  
                    break;  
            } while(m!=NULL);  
            if(m!=NULL)  
            {  
                t=k;  
                k->dpath=m;  
                k=k->next;  
            }  
            else  
            {  
                datagram++;  
                if(backlist==k)  
                {  
                    k=k->next;  
                    free(backlist);  
                    backlist=k;  
                    t=k;  
                }  
            }  
            else  
            {  

```

```

        t->next=k->next;
        free(k);
        k=t->next;
    }
}
}while(k!=NULL);
}
return(backlist);
}

```

/*****FUNCTION ADD THE DATAGRAM IN THE WAITING LIST SINCE NO *****/
 *****/BANDWIDTH IS AVAILABLE ALONG THE SHORTEST PATH*****/

```

pback add_in_backlist(pback p,pback backlist)
{
    if(backlist==NULL)
        backlist=p;
    else
    {
        p->next=backlist;
        backlist=p;
    }
    return(backlist);
}

```

```

pback send_datagram(int no,int n,pback backlist,pnode list)
{
    phop    k,t;
    int     q;
    pdtraffic l,dtraffic;
    pback   p;

    set_bw_infinite(no,list);
    dtraffic=datagram_traffic_generation(no,n,list);
    set_bw_original(list);

    l=dtraffic;
    while(l!=NULL)
    {
        dtraffic=l;
        k=dtraffic->dpath;
        do
        {
            q=is_bw_avai(k,list);
            if(q>=1)
            {

```

```
        t=k;
        k=k->next;
        free(t);
    }
    else
        break;
} while(k!=NULL);
if(k!=NULL)
{
    p=(pback) getbacklog();
    p->dpath=k;
    backlog=(pback) add_in_backlist(p,backlist);
}
else
    datagram++;
    l=l->next;
    free(dtraffic);
}
return(backlist);
}
```