**Lab Practical 6**

**Aim:** Write a Program for simulation of following routing protocols: DSR, DSDV, and AODV. Compare the performance of these protocols on various performance metrics.

**DSDV Protocol:**

DSDV (Destination-Sequenced Distance-Vector Routing

Destination-Sequenced Distance-Vector Routing (DSDV) is a table-driven

routing scheme for ad hoc mobile networks based on the Bellman–Ford

algorithm. It was developed by C. Perkins and P.Bhagwat in 1994. The main

The contribution of the algorithm was to solve the routing loop problem. Each

entry in the routing table contains a sequence number, the sequence

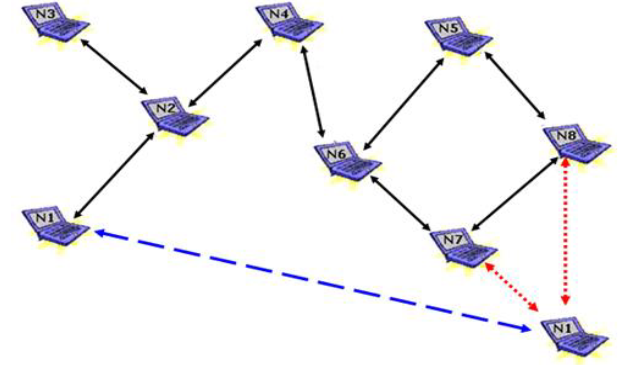
numbers are generally even if a link is present; else, an odd number is used.

The number is generated by the destination, and the emitter needs to send

out the next update with this number. Routing information is distributed

between nodes by sending full dumps infrequently and smaller incremental

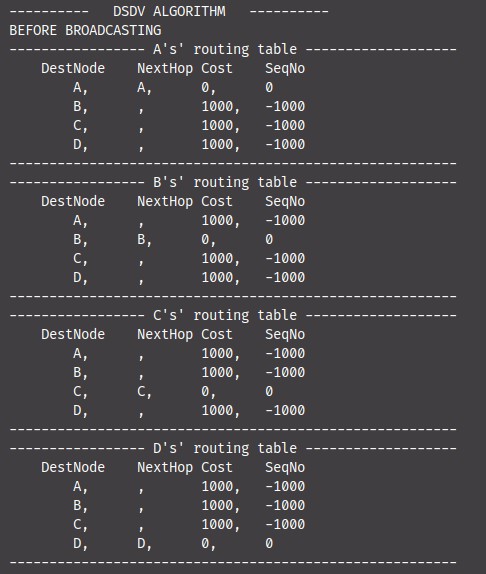
updates more frequently**.**

****

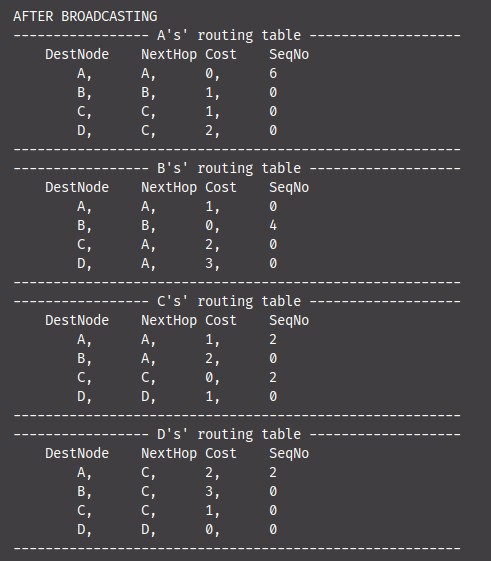
**Code:**

|  |
| --- |
| class Entry:  def \_\_init\_\_(self, dest, next\_hop, cost, seq\_no):  self.dest = dest  self.next\_hop = next\_hop  self.seq\_no = seq\_no  self.cost = cost  def \_\_str\_\_(self):  return f'\t{self.dest},\t{self.next\_hop},\t{self.cost},\t{self.seq\_no}\t'  class Node:  def \_\_init\_\_(self, name):  self.name = name  self.seq\_no = 0  self.neighbors = []  self.routing\_table = {}  def add\_neighbor(self, node):  self.neighbors.append(node)  node.neighbors.append(self)  def \_\_str\_\_(self):  return f'{self.name}'  def print\_routing\_table(self):  print(f"----------------- {self.name}'s' routing table -------------------")  print(' DestNode\tNextHop\tCost\tSeqNo')  for node\_entry in self.routing\_table:  print(f'{self.routing\_table[node\_entry]}')  print(f'--------------------------------------------------------')  def broadcast\_table(self):  self.routing\_table[self.name].seq\_no = self.seq\_no  self.seq\_no += 2  for node in self.neighbors:  node.recieve\_table(self.name, self.routing\_table)  def recieve\_table(self, sender\_name, sender\_routing\_table):  made\_update = False    for key in self.routing\_table.keys():  sender\_rountin\_entry = sender\_routing\_table[key]  own\_routing\_entry = self.routing\_table[key]  if(own\_routing\_entry.seq\_no >= sender\_rountin\_entry.seq\_no):  continue  # update the next entry  if(own\_routing\_entry.cost > (sender\_rountin\_entry.cost + 1)):  # make the update  own\_routing\_entry.seq\_no = sender\_rountin\_entry.seq\_no  own\_routing\_entry.dest = sender\_rountin\_entry.dest  own\_routing\_entry.next\_hop = sender\_name  own\_routing\_entry.cost = sender\_rountin\_entry.cost + 1  made\_update = True  if made\_update == True:  self.broadcast\_table()  INF = 1000  def print\_routing\_tables(network):  for node in network:  node.print\_routing\_table()  def main():  A = Node("A")  B = Node("B")  C = Node("C")  D = Node("D")  A.add\_neighbor(B)  A.add\_neighbor(C)  C.add\_neighbor(D)  network = [A, B, C, D]  print('---------- DSDV ALGORITHM ----------')  for node in network:  # update the routing table of node to accomodate its own entry  for other\_node in network:  if node == other\_node:  own\_entry = Entry(node, node, 0, node.seq\_no)  node.routing\_table[node.name] = own\_entry  else:  init\_entry = Entry(other\_node, "", INF, -INF)  node.routing\_table[other\_node.name] = init\_entry  print("BEFORE BROADCASTING")  print\_routing\_tables(network)  # A will start the broadcast  A.broadcast\_table()  print("AFTER BROADCASTING")  print\_routing\_tables(network)  if \_\_name\_\_ == '\_\_main\_\_':  main() |

**Output:**

****

**Before Broadcasting**

****

**After Broadcasting**

**Advantages**

The availability of paths to all destinations in network always shows that less

delay is required in the path set up process

The method of incremental update with sequence number labels, marks the

existing wired network protocols adaptable to Ad-hoc wireless networks.

Therefore, all available wired network protocol can be useful to ad hoc

wireless networks with less modi cation

**Disadvantages**

DSDV requires a regular updates of its routing tables, which uses up battery

power and a small amount of bandwidth even when the network is idle

Whenever the topology of the network changes, a new sequence number is

necessary before the network re-converges; thus, DSDV is not suitable for

highly dynamic or large scale networks. (As in all distance-vector protocols,

this does not perturb traf c in regions of the network that are not concerned

by the topology change.

**2. DSR (Dynamic Source Routing)**

The Dynamic Source Routing protocol (DSR) is a simple and efficient routing

protocol designed speci cally for use in multi-hop wireless ad hoc networks of

mobile nodes. DSR allows the network to be completely self-organizing and

self-con guring, without the need for any existing network infrastructure or

administration. It is a reactive protocol and all aspects of the protocol operate

entirely on-demand basis. It works on the concept of source routing. Source

routing is a routing technique in which the sender of a packet determines the

complete sequence of nodes through which, the packets are forwarded. The

advantage of source routing is : intermediate nodes do not need to maintain

up to date routing information in order to route the packets they forward. The

protocol is composed of the two main mechanisms of "Route Discovery" and

"Route Maintenance". DSR requires each node to maintain a route – cache of

all known self – to – destination pairs. If a node has a packet to send, it

attempts to use this cache to deliver the packet. If the destination does not

exist in the cache, then a route discovery phase is initiated to discover a

route to destination, by sending a route request. This request includes the

destination address, source address and a unique identi cation

number.

If a route is available from the route – cache, but is not valid any

more, a route maintenance procedure may be initiated. A node processes

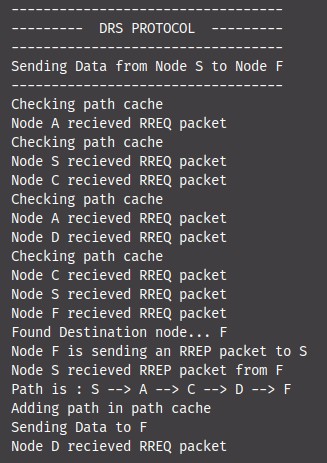
the route request packet only if it has not previously processes the packet

and its address is not present in the route cache.

**Code:**

|  |
| --- |
| class Packet:  def \_\_init\_\_(self, type, id, nodes, source, dest):  self.type = type  self.id = id  self.path = nodes  self.source = source  self.dest = dest  def get\_path(self):  path = [str(node) for node in self.path]  return ' --> '.join(path)      class Node:  def \_\_init\_\_(self, name):  self.name = name  self.neighbours = []  self.recieved\_packets = []  self.path\_cache = {}  def add\_neighbour(self, node):  self.neighbours.append(node)  node.neighbours.append(self)  def is\_packet\_already\_recieved(self, packet):  return (packet.id in self.recieved\_packets)  def recieve\_RREP(self, packet : Packet):  print(f'Node {self.name} recieved RREP packet from {packet.source}')  print(f'Path is : {packet.get\_path()}')  print('Adding path in path cache')  self.path\_cache[packet.dest] = packet.path  print(f'Sending Data to {packet.source}')  def send\_RREQ(self, packet):  print(f'Checking path cache')  if packet.dest in self.path\_cache.keys():  print('Path already in cache... no need to send RREQ')  return  for node in self.neighbours:  node.recieve\_RREQ(packet)  def recieve\_RREQ(self, packet : Packet):  print(f'Node {self} recieved RREQ packet')  # check if the dest is not current node  if packet.dest == self:  print(f"Found Destination node... {self}")  print(f'Node {self} is sending an RREP packet to {packet.source}')  packet.path.append(self)  packet.source.recieve\_RREP(Packet(2, packet.id, packet.path, self, packet.source))  return    if self.is\_packet\_already\_recieved(packet):  return  self.recieved\_packets.append(packet.id)  packet.path.append(self)  self.send\_RREQ(packet)    def \_\_str\_\_(self) -> str:  return f'{self.name}'  def main():  S = Node("S")  A = Node("A")  C = Node("C")  D = Node("D")  F = Node("F")  S.add\_neighbour(A)  A.add\_neighbour(C)  C.add\_neighbour(D)  S.add\_neighbour(D)  D.add\_neighbour(F)  print('----------------------------------')  print('--------- DRS PROTOCOL ---------')  print('----------------------------------')  print('Sending Data from S to F')  unique\_id = 1  S.recieved\_packets.append(1)  s\_RREQ = Packet(1, unique\_id, [S], S, F)  S.send\_RREQ(s\_RREQ)  if \_\_name\_\_ == "\_\_main\_\_":  main() |

**Output:**

****

**Advantages**

DSR uses a reactive approach which eliminates the need to periodically ood

the network with table update messages which are required in a table-driven

approach. The intermediate nodes also utilize the route cache information

efficiently to reduce the control overhead

**Disadvantages**

The disadvantage of DSR is that the route maintenance mechanism does not

locally repair a broken down link. The connection setup delay is higher than in

table- driven protocols. Even though the protocol performs well in static

and low-mobility environments, the performance degrades

rapidly with increasing mobility. Also, considerable routing overhead is involved due tothe source- routing mechanism employed in DSR. This routing overhead is

directly proportional to the path length.

**3. AODV Protocol:**

Ad hoc On-Demand Distance Vector (AODV) Routing is a routing

protocol for mobile ad hoc networks (MANETs) and other wireless ad hoc

networks. It was jointly developed on July 2003 in Nokia Research

Center, University of California, Santa Barbara and University of Cincinnati by

C. Perkins, E. Belding-Royer and S. Das

AODV is the routing protocol used in ZigBee – a low power, low data

rate wireless ad hoc network. There are various implementations of AODV

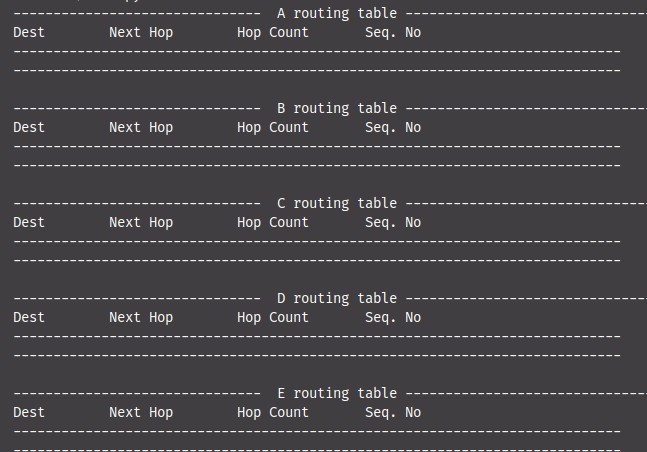
such as MAD-HOC, Kernel-AODV, AODV-UU, AODV-UCSB and AODV-

UIUC

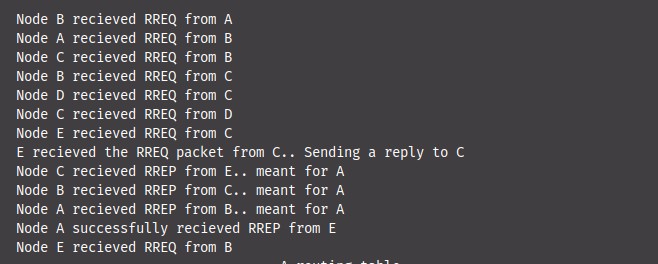
**Code:**

|  |
| --- |
| **class** **Packet**:  **def** **\_\_init\_\_**(self, source, source\_seq, source\_bcast\_id, dest):  self.source = source  self.source\_seq = source\_seq  self.source\_bcast\_id = source\_bcast\_id  self.dest = dest  self.dest\_seq = ""  self.hop\_cnt = **0**  **def** **copy**(self):  new\_packet = Packet(self.source, self.source\_seq, self.source\_bcast\_id, self.dest)  new\_packet.hop\_cnt = self.hop\_cnt + **1**  **return** new\_packet  **class** **Node**:  **def** **\_\_init\_\_**(self, name):  self.name = name  self.neighbors = []  self.routing\_table = {}  self.seq\_no = **0**  self.broadcast\_id = **0**  self.recieved\_packets = []  **def** **add\_neighbor**(self, node):  self.neighbors.append(node)  node.neighbors.append(self)  **def** **get\_neighbours**(self):  neighbours = [str(node) **for** node **in** self.neighbors]  **return** neighbours  **def** **broadcase\_RREQ**(self, packet):  **for** node **in** self.neighbors:  node.recieve\_RREQ(self, packet)  **def** **print\_routing\_table**(self):  print(f'------------------------------- {self} routing --------------------------------')  print(f'Dest**\t** Next Hop**\t** Hop Count**\t** Seq. No')  print(f'----------------------------------------------------------------------------')  **for** key **in** self.routing\_table.keys():  entry = self.routing\_table[key]  nextRow = '**\t\t**'.join([str(node) **for** node **in** entry])  print(f'{nextRow}')  print(f'----------------------------------------------------------------------------')    **def** **entry\_in\_routing\_table**(self, dest):  **return** str(dest) **in** self.routing\_table.keys()  **def** **already\_recieved\_packet**(self, packet):  **return** packet.source\_bcast\_id **in** self.recieved\_packets  **def** **add\_routing\_entry**(self, entry, sender):  self.routing\_table[entry.source.name] = [entry.source, sender, entry.hop\_cnt, entry.source\_seq]  **def** **get\_entry\_from\_routing\_table**(self, node):  **return** self.routing\_table[node.name]  **def** **recieve\_RREQ**(self, sender, packet : Packet):  print(f'Node {self} recieved RREQ from {sender}')  **if** self.already\_recieved\_packet(packet):  **return**  self.recieved\_packets.append(packet.source\_bcast\_id)  new\_packet = packet.copy()    self.add\_routing\_entry(new\_packet, sender)  **if** self.entry\_in\_routing\_table(packet.dest):  print(f'Node {self} recieve RREQ from {sender}... Sending RREP to {packet.source}')  **return**  **if** new\_packet.dest == self:  print(f'{self} recieved the RREQ packet from {sender}.. Sending a reply to {sender}')  RREP\_Packet = Packet(self, self.seq\_no, self.broadcast\_id, packet.source)  self.send\_RREP(packet.source, RREP\_Packet)  **return**  self.broadcase\_RREQ(new\_packet)  **def** **send\_RREP**(self, node, packet):  routing\_table\_entry = self.get\_entry\_from\_routing\_table(node)  next\_hop\_node = routing\_table\_entry[**1**]  next\_hop\_node.recieve\_RREP(self, packet)  **def** **recieve\_RREP**(self, sender, packet):  print(f'Node {self} recieved RREP from {sender}.. meant for {packet.dest}')  **if** self == packet.dest:  print(f'Node {self} successfully recieved RREP from {packet.source}')  **return**  ## add an entry in table  new\_entry = packet.copy()  self.add\_routing\_entry(new\_entry, sender)  self.send\_RREP(new\_entry.dest, new\_entry)  **def** **\_\_str\_\_**(self) -> str:  **return** f'{self.name}'  **def** **print\_routing\_tables**(network):  **for** node **in** network:  node.print\_routing\_table()  print("**\n**")  **def** **main**():  A = Node("A")  B = Node("B")  C = Node("C")  D = Node("D")  E = Node("E")  A.add\_neighbor(B)  B.add\_neighbor(C)  C.add\_neighbor(D)  E.add\_neighbor(B)  E.add\_neighbor(C)  network = [A, B, C, D, E]  # A will start RREQ broadcase  print\_routing\_tables(network)  RREQ\_Packet = Packet(A, A.seq\_no, A.broadcast\_id, E)  print\_routing\_tables(network)  A.add\_routing\_entry(RREQ\_Packet, A)      A.recieved\_packets.append(RREQ\_Packet.source\_bcast\_id)  A.broadcase\_RREQ(RREQ\_Packet)  print\_routing\_tables(network)  **if** \_\_name\_\_ == "\_\_main\_\_":  main() |

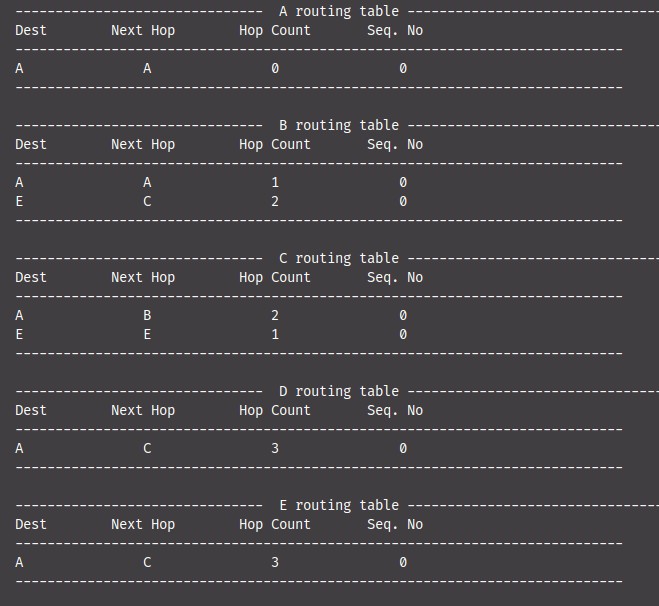
**Output: Before Routing**

****

**Routing Procedure**

****

**Final Routing Tables**

****

**Advantages:** This protocol is reliable for the wireless mesh networks. AODV is loop free

and does not require any cartelised system to handle routing process for

wireless mesh networks

**Disadvantages**

Shortest path may be lost due to traffic during the path discovery process.

AODV do not utilise any congestion control or avoidance mechanism to

balance traffic load. The delivery ratio of AODV drops dramatically from more

than 90% to about 28% when the number of connections increases from 10

to 50.