实验报告《CC-AODV 一种有效的拥塞控制AODV》

一、实验目的

- 基于论文在移动自组织网络（MANET）实现拥塞控制自组织按需距离矢量（CC-AODV）路由算法，并进一步研究其在低速网络下的表现。
- 掌握NS3网络仿真模拟器的搭建环境、搭建配置和基本使用。
- 利用NS3仿真软件实现应用CC-AODV的MANET，并将其性能与应用了AODV的MANET进行性能的比较。

二、实验原理

2.1 移动自组织网络（MANET）

本次实验场景MANET（Mobile Ad-hoc Network）指的是一种自组织的、无中心结构的移动设备网络。在这种网络中，移动设备，如智能手机、平板电脑、笔记本电脑等可以通过无线方式直接进行通信。其键特点包括：

- 自组织：MANET中的节点能够自动建立和管理网络，无需依赖预先存在的网络基础设施或集中式管理。
- 动态拓扑结构：由于节点的移动性，MANET的网络拓扑结构经常自发变化。
- 多跳路由：在MANET中，数据可能需要通过多个节点转发才能到达目的地，因为不是所有节点都直接相连。
- 异构性：网络中的节点可能包括各种类型的设备，具有不同的处理能力、存储容量和能量资源。

MANET一般应用于灾区、战区、交通站点等场景，在这些场景下，通信发生时基础设施并不是固定的。它为未来的移动节点通信提供了高效的无线通信，因此自20 世纪 90 年代中期以来一直主导着路由研究主题。根据MANET设计的本质，网络通过将路由器紧密地容纳在一起来实现更好的性能。然而，在空间比较广阔的情况下，我们必须实施稳健的算法以获得满意的网络连接结果。

2.2 自组织按需距离矢量路由 (AODV)

AODV（Ad hoc On-Demand Distance Vector）是一种用于移动设备和无线网络的路由协议，特别适用于自组织的移动无线网络，也称为MANET（Mobile Ad Hoc Network）。AODV的关键特点是按需路由（on-demand routing），意味着它只在网络节点需要时才创建路由。这里进行了一个模拟，即源节点开始向其邻近节点发送RREQ，以初始化通信，如图2.1所示。

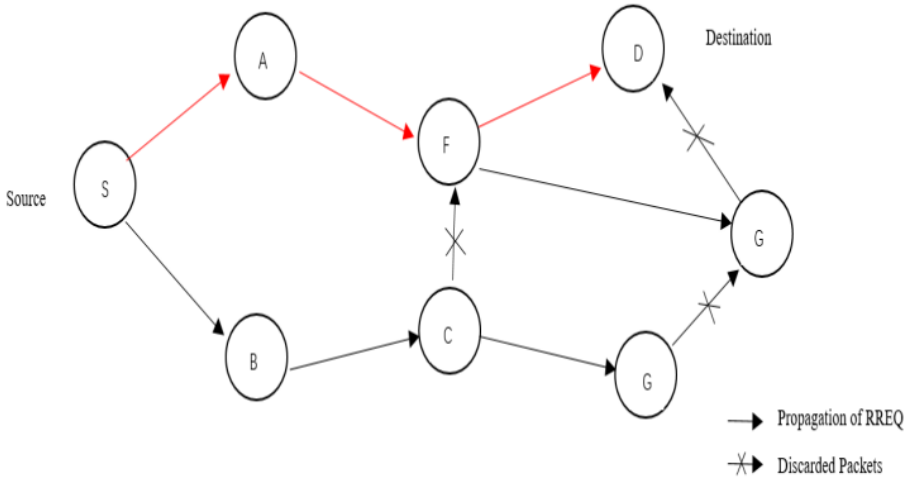


图2.1 AODV初始化

当一个中间节点接收到RREQ数据包时，路由表会添加路由信息。如果表中已经有了该条目，则路由器会比较序列号和跳数与表中现有信息。如果条件满足，表将更新路由信息。路由表有固定的条目，<目的IP地址，目的序列号，有效目的序列号标志，网络接口，跳数，下一跳，前驱列表，生存时间>。通过这些条目，节点可以确定自己是否是目的节点。此外，节点还可以检查是否之前已接收到带有相同ID的RREQ数据包。因此，如果节点接收到相同ID的数据包，则会判断是否需要更新表。节点检查两个主要条件：如果序列号大于现有路由表条目的序列号，如果跳数计数小于之前的跳数，那么表条目将通过携带信息的RREP来更新。当目的节点收到路由请求包（RREQ）后，它会生成路由回复包（RREP）。这个包会单播回其代表的源节点，并更新中间节点的路由表。因此，AODV建立了路由路径。

一旦链路失败或连接丢失，路由器错误包（RRER）就会生成并发送给源节点，源节点随后请求建立新的路由路径。当源节点收到RRER包时，它开始泛洪广播RREQ包以重新启动路由，允许AODV维护路由路径。虽然有时中间节点太忙而无法传输数据包，但仍然使用这些节点，因为它们处于通信的最短路径上。尽管如此，当使用这种方法时，其他可用的节点并没有得到充分利用，即使它们可能流量较低，导致带宽利用不足，传递数据包的延迟增加，传递的数据包数量减少，即性能出现减低。为了克服这一挑战，在论文中提出了拥塞控制CC-AODV。

2.3 拥塞控制自组织按需距离矢量路由 (CC-AODV)

拥塞控制自组织按需距离矢量路由在AODV的基础上加上了拥塞控制计算以进一步提高性能。CC-AODV通过检查表中当前节点的压力程度，使用拥塞计数器标签来确定数据的路径，一旦生成RREP包并通过节点传输，拥塞计数器就会在计数器上加一。图2.2中解释了建立路由的整个过程：首先，源节点在整个网络中执行洪泛广播RREQ包。当RREQ包到达中间节点时，路由器检查拥塞计数器是否小于某个预定值。如果比较结果小于计数器，路由表更新并转发到下一个路由器，如果大则路由器将丢弃RREQ包。一旦RREQ到达相应的目的地，路由器将生成RREP。

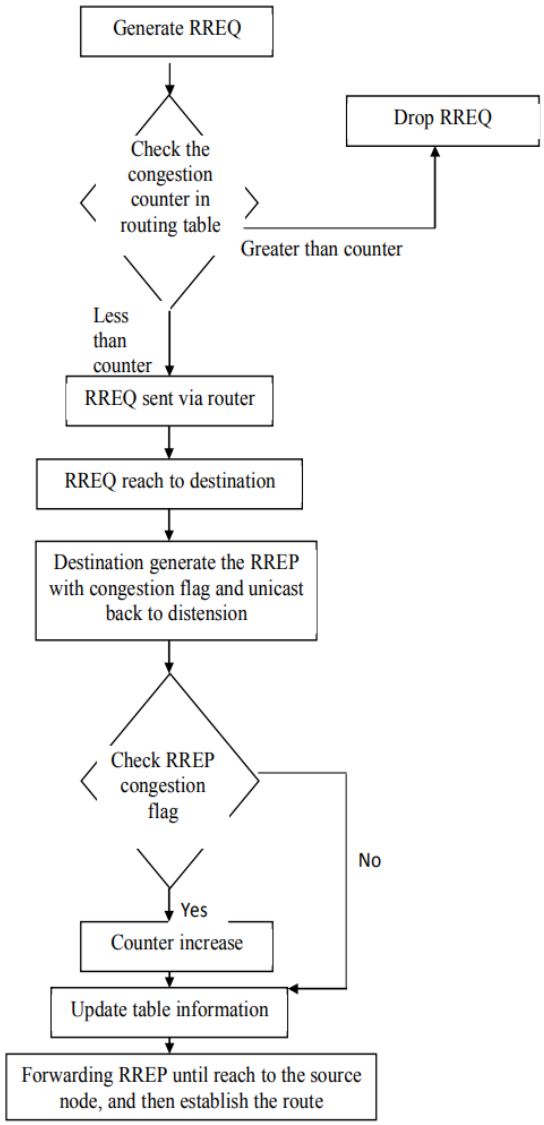


图2.2 CC-AODV路由建立流程图

在CC-AODV中，拥塞标志被添加到RREP头部。生成RREP以响应RREQ有两种情况。一种是来自源节点以建立路由，另一种是来自邻居节点以维护路由。当目的节点从源节点接收到RREQ时，它会生成带有设置为真的拥塞标志的RREP。当RREP单播回相应的源节点时，经过中间节点，路由器会检查拥塞标志。如果为真，则计数器增加，否则计数器保持不变。更新后的RREP包如下：

New-RREP:

Route Reply (RREP) Message Format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |R|A|    Reserved     |Prefix Sz|   Hop Count   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination IP address                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Destination Sequence Number                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Originator IP address                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Lifetime                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                     Congestion Flag                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

图2.3 CC-AODV中的RREP包

三、实验环境及场景

本实验探究的是在移动自组织网络（MANET）中CC-AODV对AODV的改进。MANET没有固定的结构，其拓扑结构会随着节点移动进行频繁更改。每个节点在将流量转发到网络中其他指定节点时都充当路由器。实验大致拓扑图如图3.1。

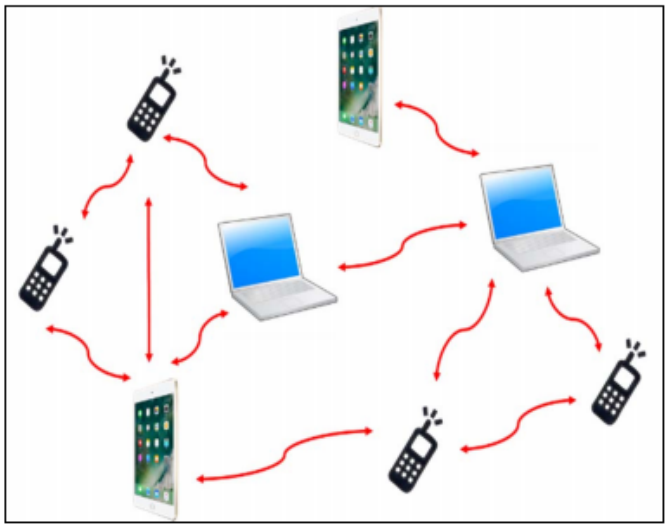


图3.1 MANET拓扑图样例

为了模拟MANET网络，搭建本实验的拓扑结构，我们使用的工具及仿真参数设置如下：

1. 仿真环境及工具：Ubuntu16.04LTS、ns-3.35、gnuplot。
2. 仿真参数：

参数		值	解释
信道类型		无线	
WiFi设置	非单播模式	DsssRate11Mbps	物理模式为 DSSS，速率为 11 Mbps
	信道传播延迟模型	恒定速度传播延迟模型	这个模型假设信号在传播过程中以恒定的速度传播，无论距离有多远，信号传播的速度都是固定的。它是一种简化的模型。
	传播损耗模型	Friis 传播损耗模型	这是一种基于距离的常用传播损耗模型。这个模型基于信号传播距离和频率计算信号的衰减。随着信号传播距离的增加，信号功率会按照距离的平方进行衰减。此外，信号的频率也会影响到信号的传播损耗。用于模拟理想的开放自由空间下的损耗。
	MAC 层类型	Adhoc 模式	虽然不能直接将 Wi-Fi MAC 层类型设置为 MANET，但可以通过选择适当的网络层协议（比如 AODV）以及配置适合 MANET 的路由和通信协议来模拟 MANET 环境。

	MAC协议	802.11	
仿真总时间		30s	具体设置为50~51s启动仿真，80s结束仿真
数据传输速率		2048 bps	模拟一个低速网络环境
节点最小移动速度		0	
节点最大移动速度		20 m/s	
数据包大小		512	
发送功率		7.5 dBm	发送功率的设置可以影响到节点的通信范围、传输距离和信号强度。增加发送功率可能会增加节点的通信范围，但也可能导致更多的能量消耗和干扰。
模拟区大小		500*500	使节点在一个 500x500 的矩形区域内随机分布
数据类型		UDP	
节点总数量（nWifis）		10 20 30 40 50	使用网络中不同数量的节点进行模拟，以象征无线网络的不同实际应用。例如，10 个节点象征着可用于农业设置的小型网络。30 个节点象征着可用于工业设置的中型网络和可用于军事基地的大型 50 个节点网络。
接收节点数量（nSinks）		5 10 15 20 25	所有节点中，前一半作为接收节点，后一半作为发送节点。第i+nSinks节点发送数据给第i个节点，0<i<nSinks-1。

实验自变量为节点总数量（nWifis）和接收节点数量（nSinks），其中接收节点数量恒为节点总数量的一半。第0~nSinks-1个节点作为接收节点，第nSinks~nWifis-1个节点作为发送节点。第i+nSinks节点向第i个节点发送数据，0<i<nSinks-1。

观察并记录丢包数量、吞吐量、包投递率、丢包率、端到端延迟随节点数量的变化。

四、实验步骤

4.1 修改ns3中AODV协议的实现代码

修改前先拷贝一份原AODV协议代码，以备进行AODV对照组的实验。

以下是具体修改的代码部分：

4.1.1 在RREP包头中添加拥塞标志位

修改 `aodv-packet.h`：

```
347       */
348       RrepHeader (uint8_t prefixSize = 0, uint8_t hopCount = 0, Ipv4Address dst =
349                   Ipv4Address (), uint32_t dstSeqNo = 0, Ipv4Address origin =
350                   Ipv4Address (), Time lifetime = MilliSeconds (0), int32_t congestionFlag = 0);
351       /**
352        * \brief Get the type ID.
353        * \return the object TypeId
354        */
```

```
436   Time GetLifeTime () const;
437
438   void SetCongestionFlag(int32_t congestionFlag);
439
440   int32_t GetCongestionFlag() const;
441
442   // Flags
443   /**
444    * \brief Set the ack required flag
```

```
479     private:
480       uint8_t        m_flags;                  ///< A - acknowledgment required flag
481       uint8_t        m_prefixSize;           ///< Prefix Size
482       uint8_t              m_hopCount;           ///< Hop Count
483       Ipv4Address    m_dst;                ///< Destination IP Address
484       uint32_t       m_dstSeqNo;           ///< Destination Sequence Number
485       Ipv4Address      m_origin;             ///< Source IP Address
486       uint32_t       m_lifeTime;           ///< Lifetime (in milliseconds)
487       uint32_t           m_congestionFlag;
488     };
```

修改 `aodv-packet.cc`：

```
299   RrepHeader::RrepHeader (uint8_t prefixSize, uint8_t hopCount, Ipv4Address dst,
300                           uint32_t dstSeqNo, Ipv4Address origin, Time lifeTime, int32_t congestionFlag)
301     : m_flags (0),
302       m_prefixSize (prefixSize),
303       m_hopCount (hopCount),
304       m_dst (dst),
305       m_dstSeqNo (dstSeqNo),
306       m_origin (origin)
307   {
308     m_lifeTime = uint32_t (lifeTime.GetMilliSeconds ());
309     m_congestionFlag = congestionFlag;
310   }
```

```
331   uint32_t
332   RrepHeader::GetSerializedSize () const
333   {
334     return 19+4;
335   }
```

```
337   void
338   RrepHeader::Serialize (Buffer::Iterator i) const
339   {
340     i.WriteU8 (m_flags);
341     i.WriteU8 (m_prefixSize);
342     i.WriteU8 (m_hopCount);
343     WriteTo (i, m_dst);
344     i.WriteHtonU32 (m_dstSeqNo);
345     WriteTo (i, m_origin);
346     i.WriteHtonU32 (m_lifeTime);
347     i.WriteHtonU32 (m_congestionFlag);
348   }
```

```
uint32_t
RrepHeader::Deserialize (Buffer::Iterator start)
{
  Buffer::Iterator i = start;

  m_flags = i.ReadU8 ();
  m_prefixSize = i.ReadU8 ();
  m_hopCount = i.ReadU8 ();
  ReadFrom (i, m_dst);
  m_dstSeqNo = i.ReadNtohU32 ();
  ReadFrom (i, m_origin);
  m_lifeTime = i.ReadNtohU32 ();
  m_congestionFlag = i.ReadNtohU32 ();
```

```
  os << " source ipv4 " << m_origin << " lifetime " << m_lifeTime << " congestion flag " << m_congestionFlag
     << " acknowledgment required flag " << (*this).GetAckRequired ();
}
```

```
void
RrepHeader::SetCongestionFlag(int32_t congestionFlag){
  m_congestionFlag = congestionFlag;
}

int32_t
RrepHeader::GetCongestionFlag() const
{
  return m_congestionFlag;
}
```

```
bool
RrepHeader::operator== (RrepHeader const & o) const
{
  return (m_flags == o.m_flags && m_prefixSize == o.m_prefixSize
          && m_hopCount == o.m_hopCount && m_dst == o.m_dst && m_dstSeqNo == o.m_dstSeqNo
          && m_origin == o.m_origin && m_lifeTime == o.m_lifeTime && m_congestionFlag == o.m_congestionFlag);
}
```

4.1.2 在路由表中添加拥塞标志位

修改 `aodv-rtable.h`：

```
RoutingTableEntry (Ptr<NetDevice> dev = 0, Ipv4Address dst = Ipv4Address (), bool vSeqNo = false, uint32_t seqNo = 0,
                   Ipv4InterfaceAddress iface = Ipv4InterfaceAddress (), uint16_t  hops = 0,
                   Ipv4Address nextHop = Ipv4Address (), Time lifetime = Simulator::Now (), int32_t congestionFlag = 0);
```

```
/// List of precursors
std::vector<Ipv4Address> m_precursorList;
/// When I can send another request
Time m_routeRequestTimout;
/// Number of route requests
uint8_t m_reqCount;
/// Indicate if this entry is in "blacklist"
bool m_blackListState;
/// Time for which the node is put into the black
Time m_blackListTimeout;

int32_t m_congestionFlag;
```

修改 `aodv-rtable.cc`：

```
RoutingTableEntry::RoutingTableEntry (Ptr<NetDevice> dev, Ipv4Address dst, bool vSeqNo, uint32_t seqNo,
                                      Ipv4InterfaceAddress iface, uint16_t hops, Ipv4Address nextHop, Time lifetime
    , int32_t congestionFlag)
```

4.1.3 添加拥塞计数器和拥塞阈值

修改 `aodv-routing-protocol.h`：

```
/// Routing table
RoutingTable m_routingTable;
/// A "drop-front" queue used by the routing layer to buffer packets to which it does not have a route.
RequestQueue m_queue;
/// Broadcast ID
uint32_t m_requestId;
/// Request sequence number
uint32_t m_seqNo;
/// Handle duplicated RREQ
IdCache m_rreqIdCache;
/// Handle duplicated broadcast/multicast packets
DuplicatePacketDetection m_dpd;
/// Handle neighbors
Neighbors m_nb;
/// Number of RREQs used for RREQ rate control
uint16_t m_rreqCount;
/// Number of RERRs used for RERR rate control
uint16_t m_rerrCount;

uint32_t m_congestionCounter;

```

修改 `aodv-routing-protocol.cc`：

```
#define MAX_CONGESTION_COUNT 4

    m_rerrCount (0),
    m_congestionCounter(0),
    m_htimer (Timer::CANCEL_ON_DESTROY),
```

```
    .AddAttribute ("MyRouteTimeout", "Value of lifetime field in RREP generating by
                   TimeValue (Seconds (11.2)),
                   MakeTimeAccessor (&RoutingProtocol::m_myRouteTimeout),
                   MakeTimeChecker ())
    .AddAttribute ("m_congestionCounter", "m_congestionCounter.",
                   UintegerValue (0),
                   MakeUintegerAccessor (&RoutingProtocol::m_congestionCounter),
                   MakeUintegerChecker<uint32_t> ())
    .AddAttribute ("BlackListTimeout", "Time for which the node is put into the bla
                   TimeValue (Seconds (5.6)),
```

4.1.4 拥塞计数超过阈值时丢弃RREQ包

修改 `aodv-routing-protocol.cc`：

```
  }

  if (m_congestionCounter > MAX_CONGESTION_COUNT){
    // std::cout<<"Congestion Counter reached Max Count"<<std::endl;
    NS_LOG_DEBUG ("Ignoring RREQ due to congestion max count reached");
    return;
  }

```

4.1.5 拥塞计数增减

修改 `aodv-routing-protocol.cc`：

```
1428  void
1429  RoutingProtocol::SendReply (RreqHeader const & rreqHeader, RoutingTableEntry const & toOrigin)
1430  {
1431    NS_LOG_FUNCTION (this << toOrigin.GetDestination ());
1432    /*
1433     * Destination node MUST increment its own sequence number by one if the sequence number in the RREQ packet is equal to that
1434     * incremented value. Otherwise, the destination does not change its sequence number before generating the  RREP message.
1435     */
1436    if (!rreqHeader.GetUnknownSeqno () && (rreqHeader.GetDstSeqno () == m_seqNo + 1))
1437      {
1438        m_seqNo++;
1439      }
1440    RrepHeader rrepHeader ( /*prefixSize=*/ 0, /*hops=*/ 0, /*dst=*/ rreqHeader.GetDst (),
1441                                          /*dstSeqNo=*/ m_seqNo, /*origin=*/ toOrigin.GetDestination (), /*lifeTime=*/ m_myRouteTimeout, 1);
1442    // std::cout<<"Set Congestion Flag to 1"<<std::endl;
1443    Ptr<Packet> packet = Create<Packet> ();
1444    SocketIpTtlTag tag;
1445    tag.SetTtl (toOrigin.GetHop ());
1446    packet->AddPacketTag (tag);
1447    packet->AddHeader (rrepHeader);
1448
1449    TypeHeader tHeader (AODVTYPE_RREP);
1450    packet->AddHeader (tHeader);
1451    Ptr<Socket> socket = FindSocketWithInterfaceAddress (toOrigin.GetInterface ());
1452    NS_ASSERT (socket);
1453    socket->SendTo (packet, 0, InetSocketAddress (toOrigin.GetNextHop (), AODV_PORT));
1454  }
```

```
1547
1548    if(rrepHeader.GetCongestionFlag()==1){
1549      m_congestionCounter++;
1550      // std::cout<<"Congestion Counter ++"<<std::endl;
1551    }
1552    /*
```

```
1561     */
1562    Ptr<NetDevice> dev = m_ipv4->GetNetDevice (m_ipv4->GetInterfaceForAddress (receiver));
1563    RoutingTableEntry newEntry (/*device=*/ dev, /*dst=*/ dst, /*validSeqNo=*/ true, /*seqno=*/ rrepHeader.GetDstSeqno (),
1564                                          /*iface=*/ m_ipv4->GetAddress (m_ipv4->GetInterfaceForAddress (receiver), 0),/*hop=*/ hop,
1565                                          /*nextHop=*/ sender, /*lifeTime=*/ rrepHeader.GetLifeTime (), rrepHeader.GetCongestionFlag());
1566    RoutingTableEntry toDst;
```

```
1651    if (tag.GetTtl () < 2)
1652      {
1653        NS_LOG_DEBUG ("TTL exceeded. Drop RREP destination " << dst << " origin " << rrepHeader.GetOrigin ());
1654        if(m_congestionCounter > 0){
1655          m_congestionCounter--;
1656          // std::cout<<"Congestion Counter -- (TTL exceeded)"<<std::endl;
1657        }
1658
1659        return;
1660      }
```

```
1780    m_routingTable.InvalidateRoutesWithDst (unreachable);
1781    if(m_congestionCounter > 0){
1782      m_congestionCounter--;
1783      // std::cout<<"Congestion Counter -- (RERR)"<<std::endl;
1784    }
```

4.2 编写实验拓扑搭建代码

根据实验环境及场景中的仿真参数，编写实验仿真代码，详见 `AODV_Topology.cc` 。将编写好的代码放到 `scratch` 文件夹下。

4.3 运行实验

4.3.1 编译运行

4.3.1.1 生成AODV协议下的实验结果

将 `AODV_Topology.cc` 如下代码段中文件保存名更改为 `aodv-data.txt` 。

```
393     std::ofstream myfile;
394     myfile.open ("DATA_AODV_TOPOLOGY.txt", std::ios::app);
395     myfile
396           <<nWifis<<" "
397           <<nSinks<<" "
398           <<lostPacketssum<<" "
399           <<((rxBytessum * 8.0) / timeDiff)/1024<<" "
400           <<(double)((rxPacketsum * 100.0) /txPacketsum)<<" "
401           <<(double)((lostPacketssum * 100.0) /txPacketsum)<<" "
402           <<Delaysum/rxPacketsum<<std::endl;
```

图3.1 结果文件对应代码段

在虚拟机中打开终端，进入 `ns-allinone-3.35/ns-3.35` 目录，依次输入如下命令。

```
./waf --run "scratch/AODV-Topology --nWifis=10 --nSinks=5"
./waf --run "scratch/AODV-Topology --nWifis=20 --nSinks=10"
./waf --run "scratch/AODV-Topology --nWifis=30 --nSinks=15"
./waf --run "scratch/AODV-Topology --nWifis=40 --nSinks=20"
./waf --run "scratch/AODV-Topology --nWifis=50 --nSinks=25"
```

运行成功后，会在 `ns-3.35` 目录下看到生成了 `aodv-data.txt` ，里面有5行数据，每一行分别表示：

节点总数量 接收节点数量 丢包数量 吞吐量 包投递率 丢包率 端到端延迟

4.3.1.2 生成CC-AODV协议下的实验结果

将图3.1中的文件名改为cc-aodv-data.txt。

在虚拟机中打开终端，进入ns-allinone-3.35/ns-3.35目录，输入命令同上。

运行成功后，会在ns-3.35目录下看到生成了cc-aodv-data.txt，里面也有5行数据。

4.3.2 结果可视化

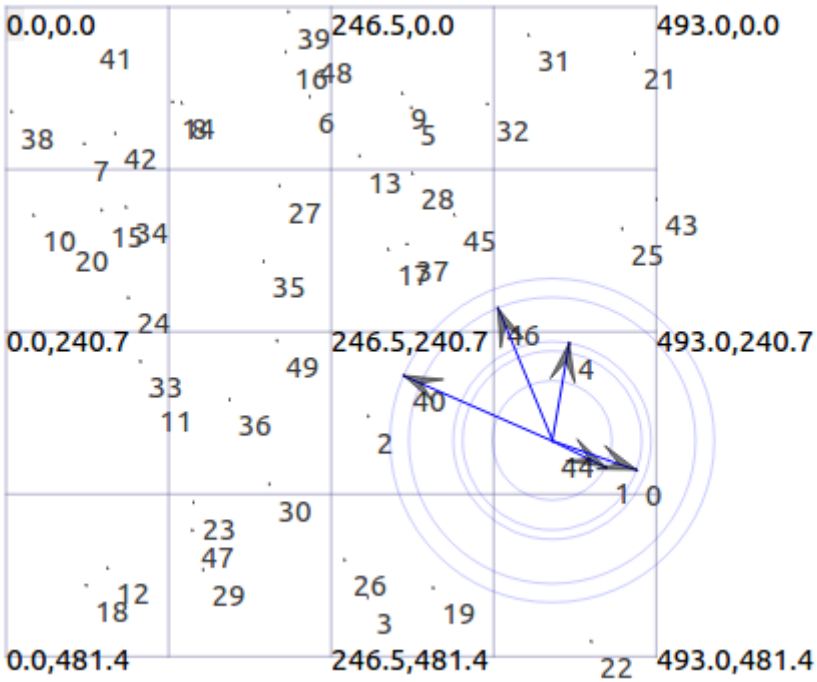
编写绘图指令，保存到 `TaskB_AODV_Code.plt` 文件中。

在虚拟机中打开终端，进入 `ns-allinone-3.35/ns-3.35` 目录，输入命令：

```
gnuplot TaskB_AODV_Plot_Code.plt
```

这一步之后就会得到 `TaskB_AODV_Plot.pdf` ，里面是5个图表，展示了丢包数量、吞吐量、包投递率、丢包率、端到端延迟在两种协议下随节点数量的变化。

同时，运行aodv_topology.cc后会生成xml文件，在NetAnim中打开此文件可看到节点传输情况。

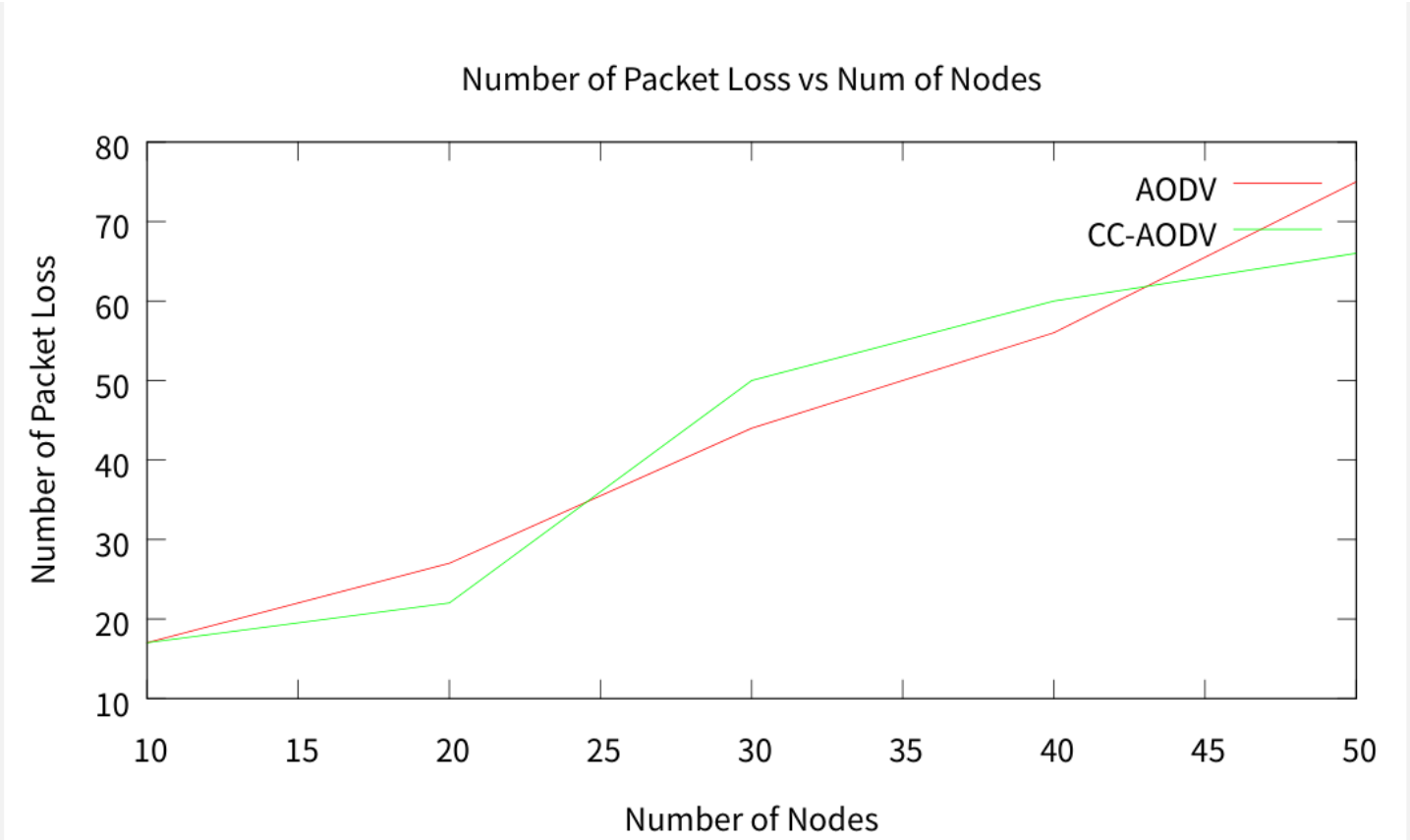


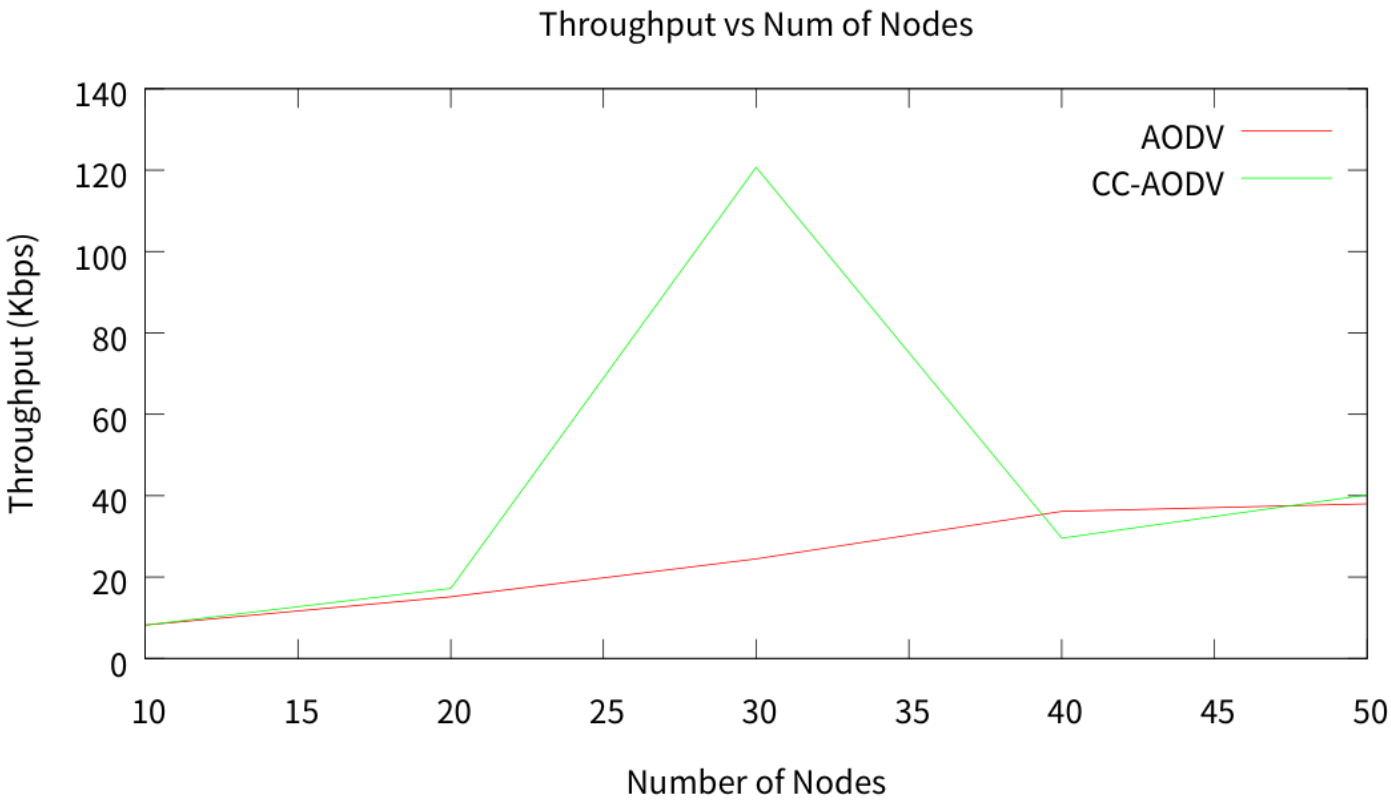
五、实验结果及分析

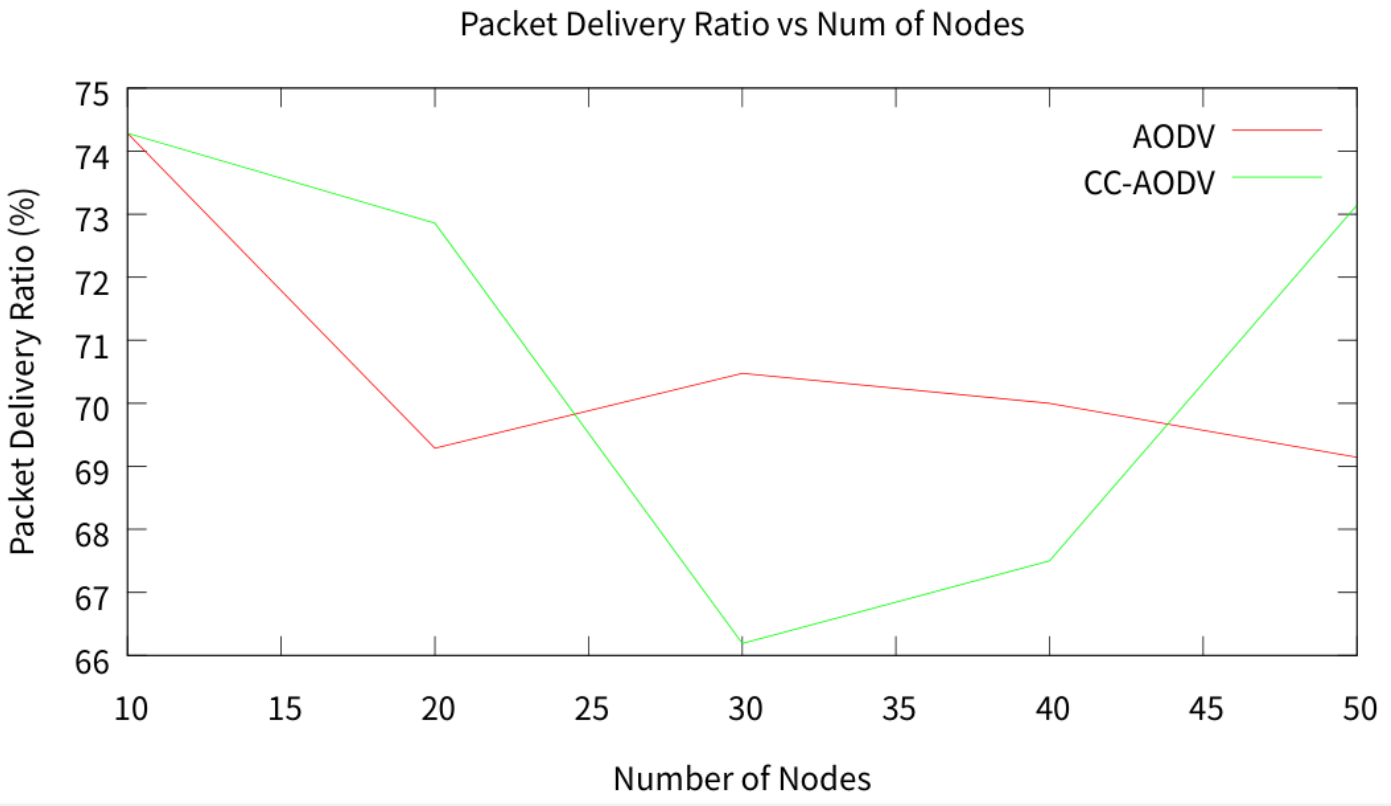
我们将节点移动速度分别设置成了0到4 m/s和4到10 m/s。得到以下结果：

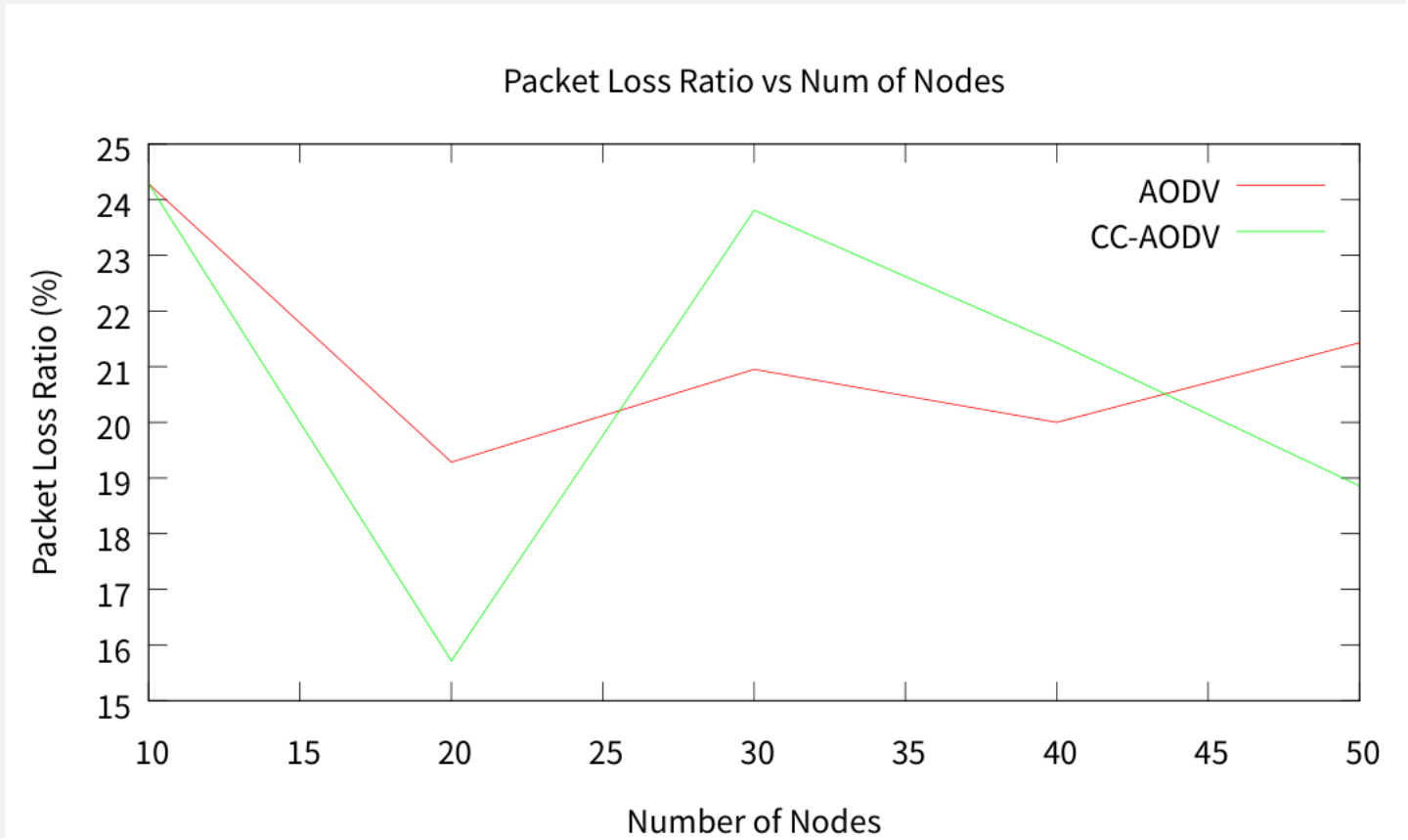
5.1 节点移动速度：4-10 m/s

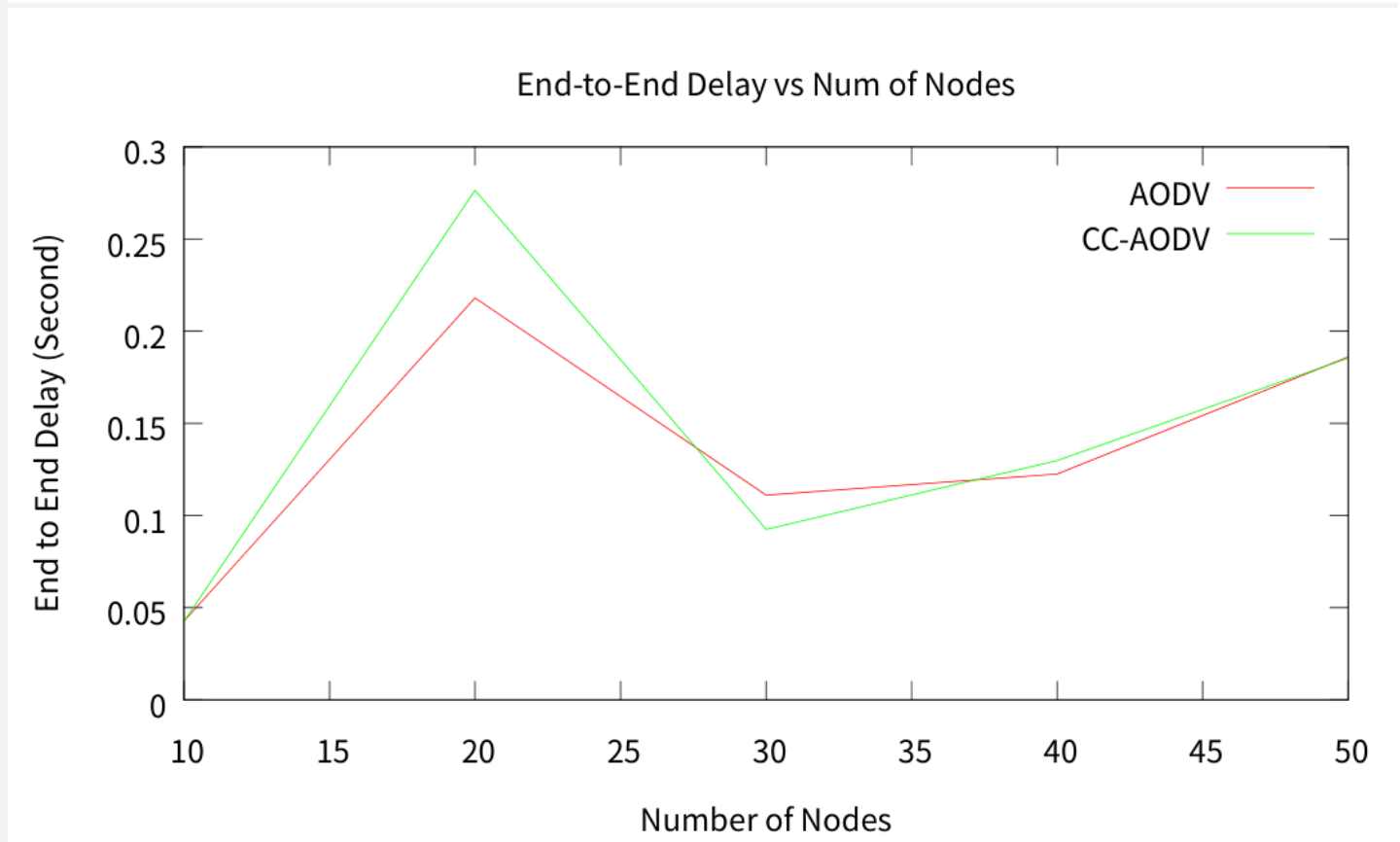
得到如下五个实验结果，进行分析：




Throughput vs Num of Nodes
Throughput (Kbps)
Number of Nodes
AODV
CC-AODV
0
20
40
60
80
100
120
140
10
15
20
25
30
35
40
45
50


Packet Delivery Ratio vs Num of Nodes
Packet Delivery Ratio (%)
Number of Nodes
AODV
CC-AODV
66
67
68
69
70
71
72
73
74
75
10
15
20
25
30
35
40
45
50



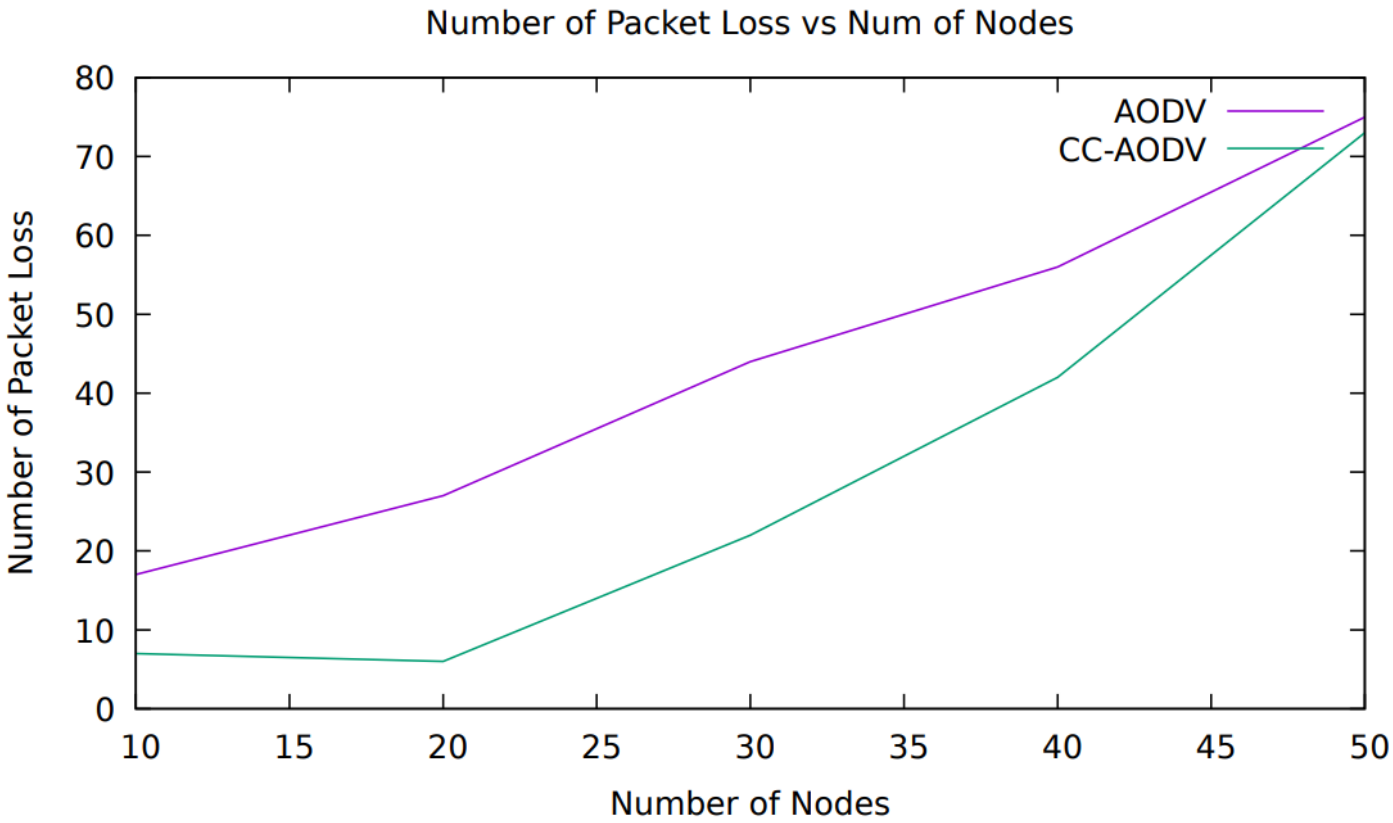


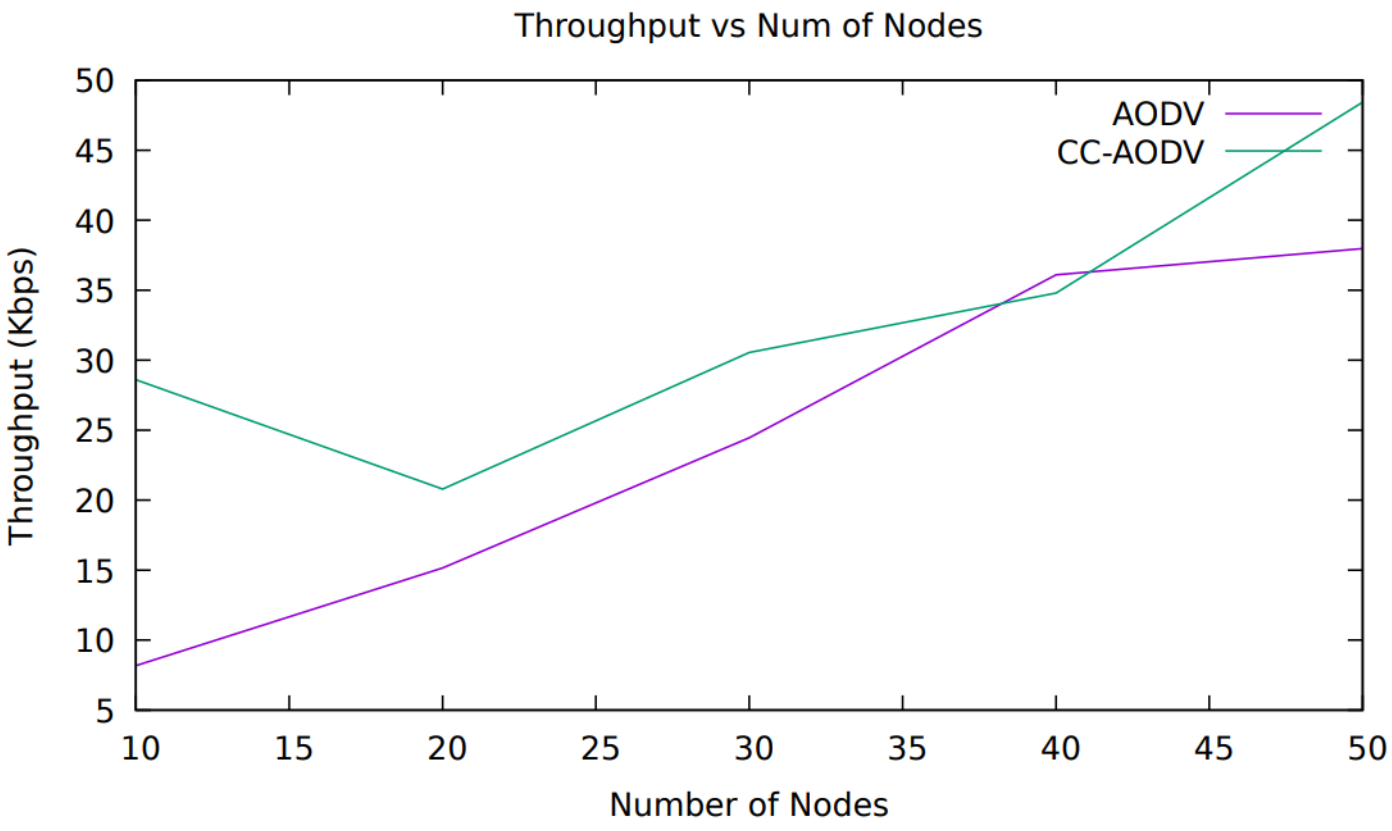
在进行实验结果的分析后，我们将其与原论文的实验结果进行对比，发现：

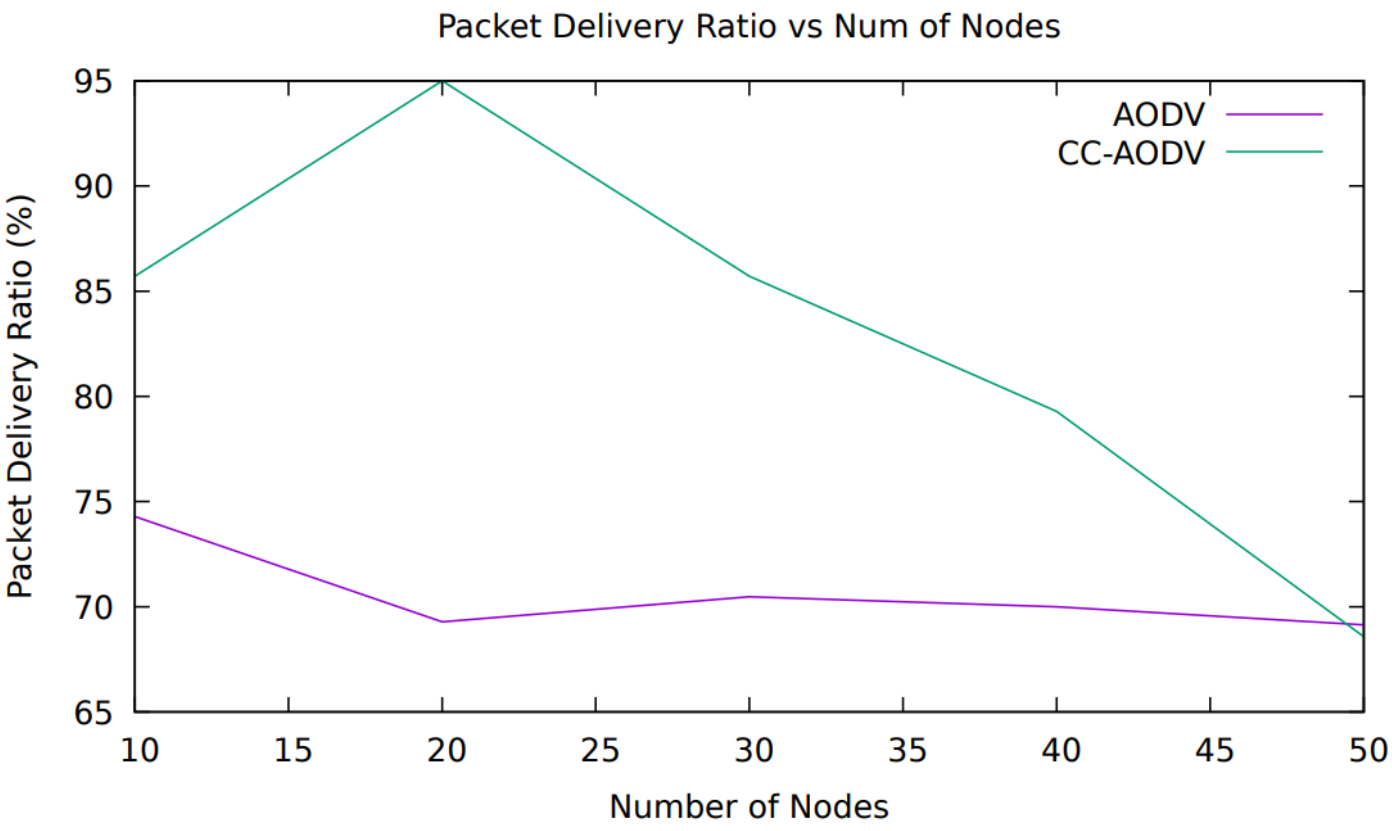
1. 总丢包量与论文完全符合。
2. 论文中的包投递率与节点数量成负相关。在本实验中，CC-AODV在节点数量为30到50时，节点数量与包投递率成正相关，当网络密度较大时，本实验实现的CC-AODV具有更好的性能。

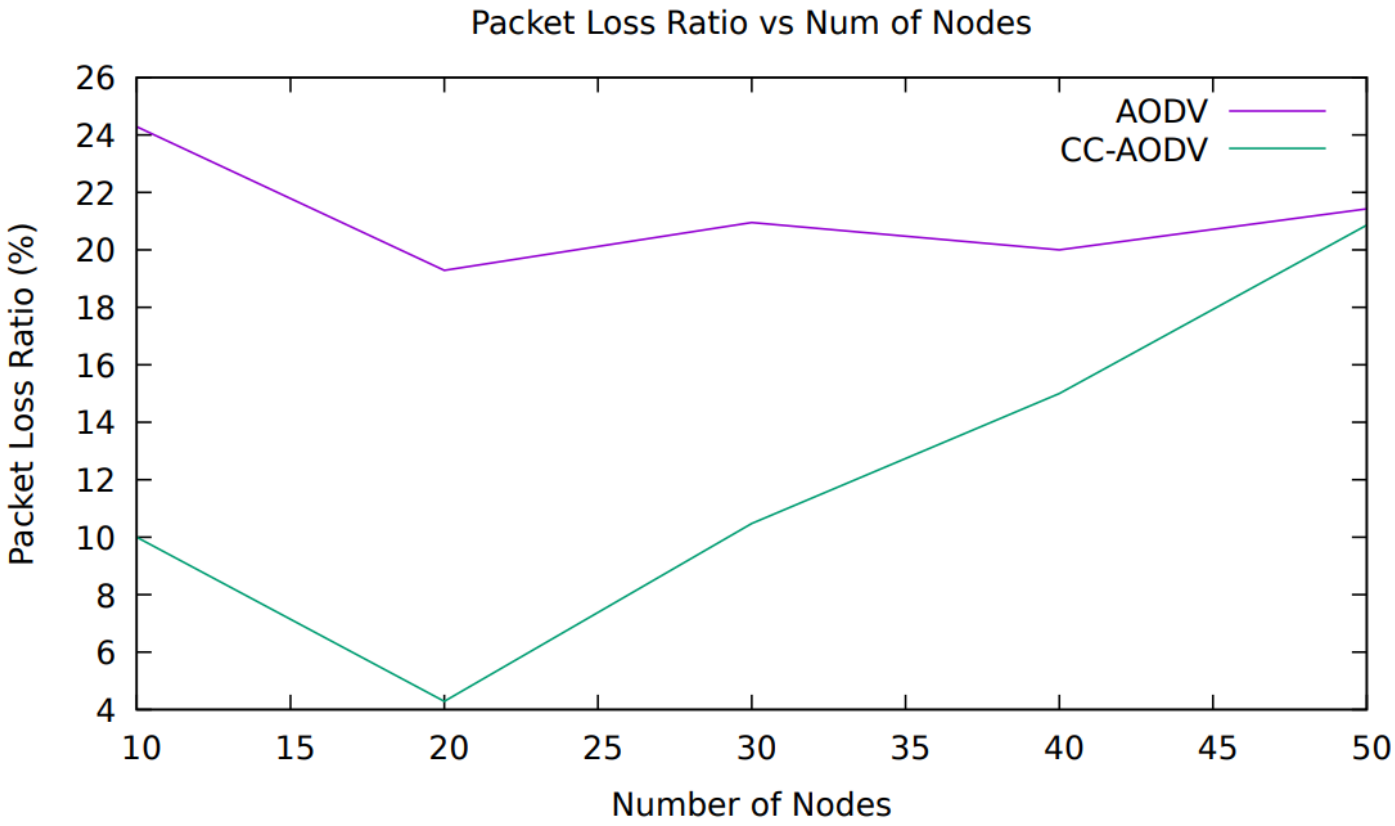
3. 吞吐量与论文部分符合。本实验中，在节点数量为30时，CC-AODV的吞吐量出现了一个高的峰值，在节点数量为20到40时，本实验实现的CC-AODV有着更高的吞吐量。
4. 论文中没有测试丢包率，只测量了丢包量。本实验进一步测试了丢包率，TTL对性能的影响很大，当网络密度较大时，CC-AODV具有更好的性能。
5. 端到端延迟与论文基本相符。

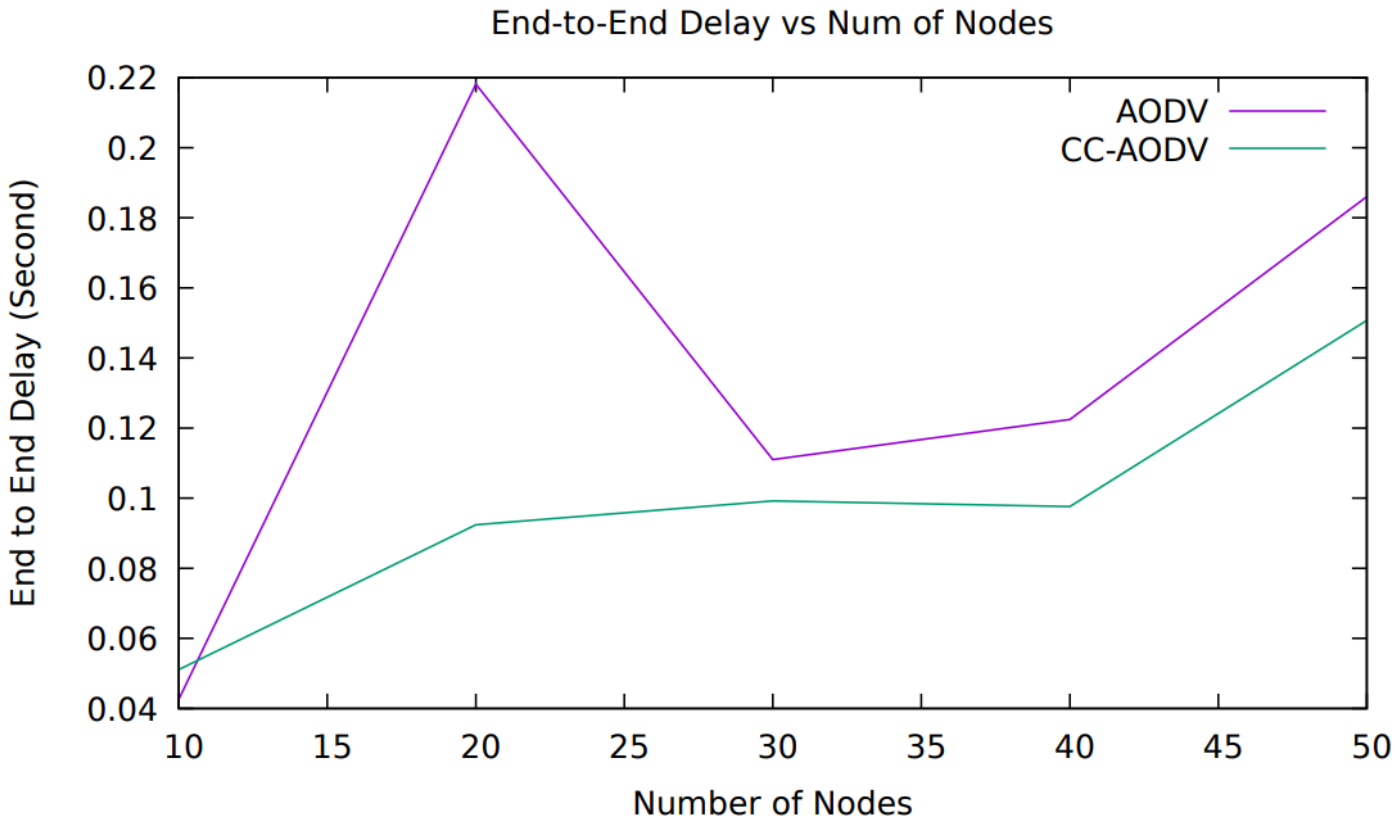
5.2 节点移动速度：0-4 m/s




Throughput vs Num of Nodes
Throughput (Kbps)
Number of Nodes
AODV
CC-AODV
50
45
40
35
30
25
20
15
10
5
10
15
20
25
30
35
40
45
50


Packet Delivery Ratio vs Num of Nodes
Packet Delivery Ratio (%)
Number of Nodes
AODV
CC-AODV
95
90
85
80
75
70
65
10
15
20
25
30
35
40
45
50





可以看到，当节点移动速度设置为0到4 m/s时，CC-AODV的丢包量、丢包率、端到端延迟均小于AODV，吞吐量和包投递率均大于AODV，效果比较稳定和明显。

5.3 结果分析

在一开始设置参数时我们采用了较低的传输速度，为了研究在网络连接不够通畅的情况下拥塞控制的效果。但是若是不改变其他参数，会导致实验结果的大幅度改变。

我们思考了在节点速度设置不同时，实验结果产生差异的原因，后来发现速度为主要影响因素。在我们参考的论文中，节点速度设置为4-10 m/s。然而，本实验将节点速度设置为4-10 m/s时并没有得到很好的结果，于是我们猜测可能是网络原因。由于我们的网络相比论文中的网络速度更低，因此我们模拟的网络中传输一次的所需时间相对来说更长。在同样的4-10 m/s的移动速度下，链路断裂，路由表更新的可能性是相同的，而由于本实验中传输一次时间长，则在过程中发生问题的概率更大，相较于论文中实验的情况更加不稳定。因此我们等比例降低了我们的移动速度，降为0-4m/s，最后结果趋于稳定，与论文结果基本相符。

在这样的情况下，我们进行了猜想验证，着手实施了链路断裂的监听，在头文件的错误处理函数中加上了消息输出，结果如下：

4-10m/s情况：

```
Route break: neighbor 10.1.1.10 loss
Route break: neighbor 10.1.1.8 loss
Route break: neighbor 10.1.1.1 loss
Route break: neighbor 10.1.1.8 loss
Route break: neighbor 10.1.1.1 loss
Route break: neighbor 10.1.1.1 loss
Route break: neighbor 10.1.1.10 loss
Route break: neighbor 10.1.1.4 loss
Route break: neighbor 10.1.1.5 loss
Route break: neighbor 10.1.1.5 loss
Route break: neighbor 10.1.1.5 loss
Route break: neighbor 10.1.1.5 loss
Route break: neighbor 10.1.1.2 loss


Total Tx Packets: 70
Total Rx Packets: 52
Total Packets Lost: 17
Throughput: 8.17111 Kbps
```

0-4m/s情况：

```
Waf: Leaving directory `/home/gg/tarba
d'
Build commands will be stored in build
'build' finished successfully (11.228s
Route break: neighbor 10.1.1.2 loss
Route break: neighbor 10.1.1.8 loss
Route break: neighbor 10.1.1.8 loss
Route break: neighbor 10.1.1.9 loss


Total Tx Packets: 70
Total Rx Packets: 60
Total Packets Lost: 7
Throughput: 28.6065 Kbps
Packets Delivery Ratio: 85.7143%
```

可以发现实验中确实出现了链路断裂的情况，并且在0-4m/s时出现次数比4-10m/s时少了很多，佐证了我们的推测。

六、实验中遇到的问题

1. 最初我们实验生成的结果中，端到端延迟随节点数量递减，与论文和实际相反。我们检查发现是绘图代码绘图顺序与结果保存到文本中的顺序不符，改正绘图代码后，我们得到了正确的图表。
2. 一开始我们节点移动速度设为0~20m/s，发现结果中各观测量都会出现异常峰值，曲线表现很不稳定。后来我们对比原论文，将移动速度设为与原论文相同的4~10m/s，表现和之前一样不稳定。我们无意中将速度设为0~4m/s，发现效果较理想。后来分析原因时我们想起代码中传速率设置为2048bps，属于低速网络，而论文中未提及网络速率，应为正常网络（传输速率比我们高），因而4~10m/s的移动速度对于原论文场景来说不会产生较大影响，而对于我们的低速网络场景来说，4~10m/s属于相对较大的移动速度，拓扑结构频繁剧烈变化，从而导致各方面性能表现都不稳定。

七、总结

在本次实验中，我们成功实现了在低速网络中CC-AODV算法的运用，并将其与AODV算法进行了比较，得知了在移速较为缓慢的情况下，此算法可以对MANET网络性能起到较大的提升作用，并且从链路断裂频率的角度出发对实验背景条件进行了一定程度的分析，实验结果基本符合预期。

未来，我们预计在这个实验的基础上进行实验场景的进一步细化，使之更加符合现实环境，并且将会加入除了主动路由以外的几种被动路由算法，进行深入比较，以得出更加全面的结果。

```
#include <fstream>
#include <iostream>
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/mobility-module.h"
#include "ns3/aodv-module.h"
#include "ns3/olsr-module.h"
#include "ns3/dsdv-module.h"
#include "ns3/dsr-module.h"
#include "ns3/applications-module.h"
#include "ns3/yans-wifi-helper.h"
#include "ns3/flow-monitor-helper.h"
#include "ns3/flow-monitor-module.h"

using namespace ns3;
using namespace dsr;

NS_LOG_COMPONENT_DEFINE ("manet-routing-compare");//定义名为manet-routing-compare的日志组件

class RoutingExperiment
{//包含了实验所需的方法和变量
public:
```

```
  RoutingExperiment ();
  //负责运行实验，接受三个参数：nWifis-WiFi节点数量，nSinks-sink节点数量，txp-传输功率
  void Run (int nWifis, int nSinks, double txp);

  void CommandSetup (int argc, char **argv);//处理命令行参数

private:
  //用于设置节点接收数据的socket。两个参数：addr是节点的IPv4地址，node是节点的指针
  //函数中创建了一个UDP的socket，并设置了回调函数，用于接收数据。
  Ptr<Socket> SetupPacketReceive (Ipv4Address addr, Ptr<Node> node);
  //作为回调函数被用于处理收到的数据包。接受一个 Socket 类型的指针作为参数。
  //当socket接收到数据时，会调用这个函数来处理接收到的数据包。
  void ReceivePacket (Ptr<Socket> socket);

  uint32_t port;//socket的端口号
  uint32_t bytesTotal;//总共传输的字节数量
  uint32_t packetsReceived;//收到的数据包数量

  std::string m_protocolName;//当前使用的路由协议的名称
  bool m_traceMobility;//是否启用了节点移动性的追踪
  uint32_t m_protocol;// 使用哪种路由协议进行仿真实验
};

RoutingExperiment::RoutingExperiment ()
  : port (9),
    bytesTotal (0),
    packetsReceived (0),
    m_traceMobility (false),
    m_protocol (2) // AODV
{
}

//生成接收到数据包的日志信息
static inline std::string
PrintReceivedPacket (Ptr<Socket> socket, Ptr<Packet> packet, Address
senderAddress)
{
  std::ostringstream oss;

  oss << Simulator::Now ().GetSeconds () << " " << socket->GetNode ()->GetId
();

  if (InetSocketAddress::IsMatchingType (senderAddress))
    {
      InetSocketAddress addr = InetSocketAddress::ConvertFrom (senderAddress);
      oss << " received one packet from " << addr.GetIpv4 ();
    }
```

```
69    else
70      {
71        oss << " received one packet!";
72      }
73    return oss.str ();
74  }
75
76  void
77  RoutingExperiment::ReceivePacket (Ptr<Socket> socket)
78  {
79    Ptr<Packet> packet;
80    Address senderAddress;
81    while ((packet = socket->RecvFrom (senderAddress)))
82      {
83        bytesTotal += packet->GetSize ();
84        packetsReceived += 1;
85        //NS_LOG_UNCOND (PrintReceivedPacket (socket, packet, senderAddress));
86      }
87  }
88
89  //设置一个接收数据包的Socket，并绑定到特定的地址和端口上
90  Ptr<Socket>
91  RoutingExperiment::SetupPacketReceive (Ipv4Address addr, Ptr<Node> node)
92  {
93  //根据字符串 "ns3::UdpSocketFactory" 获取相应的 TypeId，用于创建一个UDP类型的Socket
94    TypeId tid = TypeId::LookupByName ("ns3::UdpSocketFactory");
95    Ptr<Socket> sink = Socket::CreateSocket (node, tid);//创建了一个Socket对象
    sink，使用了之前获取的 UDP Socket 的 TypeId，将其绑定到指定的节点 node 上
96    InetSocketAddress local = InetSocketAddress (addr, port);//创建了一个
    InetSocketAddress 对象 local，表示要绑定的地址和端口
97    sink->Bind (local);// 将创建的Socket sink 绑定到指定的地址和端口上，以便它可以接收发
    送到这个地址和端口的数据包
98    sink->SetRecvCallback (MakeCallback (&RoutingExperiment::ReceivePacket,
    this));//设置了当Socket接收到数据包时的回调函数
99
100   return sink;//返回创建的Socket对象 sink，以便在其他地方使用它接收数据包
101 }
102
103 void
104 RoutingExperiment::CommandSetup (int argc, char **argv)
105 {
106   CommandLine cmd (__FILE__);
107   cmd.AddValue ("traceMobility", "Enable mobility tracing", m_traceMobility);
108   cmd.AddValue ("protocol", "1=OLSR;2=AODV;3=DSDV;4=DSR", m_protocol);
109   cmd.Parse (argc, argv);
110 }
111
```

```
112 int
113 main (int argc, char *argv[])
114 {
115   RoutingExperiment experiment;
116 
117   double txp = 7.5;
118   int nWifis = 65;
119   int nSinks = 30;
120 
121   CommandLine cmd (__FILE__);
122   cmd.AddValue ("nWifis", "nWifis", nWifis);
123   cmd.AddValue ("nSinks", "nSinks", nSinks);
124   cmd.Parse (argc, argv);
125   experiment.Run (nWifis, nSinks, txp);
126 }
127 
128 void
129 RoutingExperiment::Run (int nWifis, int nSinks, double txp)
130 {
131   Packet::EnablePrinting ();//启用数据包的打印功能
132   double TotalTime = 80.0;//仿真总时间
133   std::string rate ("2048bps");//数据传输速率,设置为2048可能是为了模拟一个低速网络环
    境
134   std::string phyMode ("DsssRate11Mbps");//物理模式为 DSSS，速率为 11 Mbps
135   std::string tr_name ("AODV_Topology_Trace");//追踪文件名
136   int nodeSpeed = 4; //节点移动速度m/s
137   int nodePause = 0; //暂停时间s
138   m_protocolName = "protocol";
139 
140   Config::SetDefault  ("ns3::OnOffApplication::PacketSize",StringValue
    ("512"));//设置OnOffApplication 应用程序的数据包大小为 512 字节
141   Config::SetDefault ("ns3::OnOffApplication::DataRate",  StringValue
    (rate));//设置 OnOff 应用的数据传输速率为之前定义的 rate
142 
143   //Set Non-unicastMode rate to unicast mode
144   Config::SetDefault
    ("ns3::WifiRemoteStationManager::NonUnicastMode",StringValue (phyMode));//设置
    了 Wi-Fi 管理器的非单播模式为之前定义的 phyMode
145 
146   NodeContainer adhocNodes;
147   adhocNodes.Create (nWifis);//创建了 nWifis 个节点的节点容器
148 
149   // setting up wifi phy and channel using helpers
150   WifiHelper wifi;
151   wifi.SetStandard (WIFI_STANDARD_80211b);//创建了一个名为 wifi 的 WifiHelper 对
    象，并设置了 Wi-Fi 标准为 802.11b
152 
```

```
153   YansWifiPhyHelper wifiPhy;//YANS 是 NS-3 中用于模拟无线网络的物理层模型之一，用于模拟 WiFi 设备的物理层行为。
154   YansWifiChannelHelper wifiChannel;//创建了 Wi-Fi 的物理层和信道属性的帮助器对象
155   wifiChannel.SetPropagationDelay
("ns3::ConstantSpeedPropagationDelayModel");//设置信道传播延迟模型为恒定速度传播延迟模型
156   wifiChannel.AddPropagationLoss ("ns3::FriisPropagationLossModel");//添加了 Friis 传播损耗模型，这是一种基于距离的常用传播损耗模型
157   wifiPhy.SetChannel (wifiChannel.Create ());//将之前设置好的信道应用到 Wi-Fi 的物理层中
158
159   // Add a mac and disable rate control
160   WifiMacHelper wifiMac;
161   wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
162                                 "DataMode",StringValue (phyMode),
163                                 "ControlMode",StringValue (phyMode));
164
165   wifiPhy.Set ("TxPowerStart",DoubleValue (txp));
166   wifiPhy.Set ("TxPowerEnd", DoubleValue (txp));//设置 Wi-Fi 的发送功率起始值和结束值为 txp
167
168   wifiMac.SetType ("ns3::AdhocWifiMac");//将 Wi-Fi MAC 层类型设置为 Adhoc 模式，表明节点将在自组网模式下进行通信
169   NetDeviceContainer adhocDevices = wifi.Install (wifiPhy, wifiMac,
adhocNodes);//安装 Wi-Fi 的物理层和 MAC 层到之前创建的节点容器 adhocNodes 中，并将创建的设备放入 adhocDevices 中
170
171   MobilityHelper mobilityAdhoc;//移动模型帮助器对象
172 //以确保在不同场景中获得一致的移动性。streamIndex作伪随机数生成器的种子。
173   //在相同的场景或条件下，如果多次运行模拟且种子相同，伪随机数生成器将以相同的顺序生成相同的随机数序列，进而导致相同的移动性模式。
174   //可以确保在多次运行模拟的情况下，在相同的条件下得到相同的移动性，以便进行一致的模拟和比较。
175   int64_t streamIndex = 0;
176
177   //使得节点在一个 500x500 的矩形区域内随机分布
178   ObjectFactory pos;
179   pos.SetTypeId ("ns3::RandomRectanglePositionAllocator");
180   pos.Set ("X", StringValue ("ns3::UniformRandomVariable[Min=0.0|Max=500.0]"));
181   pos.Set ("Y", StringValue ("ns3::UniformRandomVariable[Min=0.0|Max=500.0]"));
182
183   Ptr<PositionAllocator> taPositionAlloc = pos.Create ()-
>GetObject<PositionAllocator> ();//位置分配器
184   streamIndex += taPositionAlloc->AssignStreams (streamIndex);//为位置分配器分配了随机数流，这些随机数流用于生成节点的位置和移动参数中的随机变量。
185
186   //两个字符串流 ssSpeed 和 ssPause，分别用于生成节点的移动速度和暂停时间的随机变量
```

```
187   std::stringstream ssSpeed;
188   ssSpeed << "ns3::UniformRandomVariable[Min=0.0|Max=" << nodeSpeed << "]";//最小值0，最大值nodeSpeed=20m/s
189   std::stringstream ssPause;
190   ssPause << "ns3::ConstantRandomVariable[Constant=" << nodePause << "]";//恒为nodePause=0
191   //设置节点的移动模型为随机航点移动模型，并传入了速度、暂停时间以及之前创建的位置分配器。
192   mobilityAdhoc.SetMobilityModel ("ns3::RandomWaypointMobilityModel",
193                                   "Speed", StringValue (ssSpeed.str ()),
194                                   "Pause", StringValue (ssPause.str ()),
195                                   "PositionAllocator", PointerValue (taPositionAlloc));
196   //为移动模型设置位置分配器
197   mobilityAdhoc.SetPositionAllocator (taPositionAlloc);
198   //将移动模型安装到之前创建的无线节点容器 adhocNodes 中
199   mobilityAdhoc.Install (adhocNodes);
200   streamIndex += mobilityAdhoc.AssignStreams (adhocNodes, streamIndex);//为移动模型分配了随机数流
201   NS_UNUSED (streamIndex); //从这一点开始，streamIndex 不再被使用
202
203   //一系列路由协议的帮助器对象以及 Internet 协议栈的帮助器对象
204   AodvHelper aodv;
205   OlsrHelper olsr;
206   DsdvHelper dsdv;
207   DsrHelper dsr;
208   DsrMainHelper dsrMain;
209   Ipv4ListRoutingHelper list;
210   InternetStackHelper internet;
211
212   switch (m_protocol)
213     {
214     case 1:
215       list.Add (olsr, 100);
216       m_protocolName = "OLSR";
217       break;
218     case 2:
219       list.Add (aodv, 100);//优先级设为100
220       m_protocolName = "AODV";
221       break;
222     case 3:
223       list.Add (dsdv, 100);
224       m_protocolName = "DSDV";
225       break;
226     case 4:
227       m_protocolName = "DSR";
228       break;
229     default:
```

```
230       NS_FATAL_ERROR ("No such protocol:" << m_protocol);
231     }
232
233   if (m_protocol < 4)
234     {
235       internet.SetRoutingHelper (list);//设置路由辅助程序
236       internet.Install (adhocNodes);//安装该协议栈到 adhocNodes 上
237     }
238   else if (m_protocol == 4)
239     {
240       internet.Install (adhocNodes);
241       dsrMain.Install (dsr, adhocNodes);
242     }
243
244   NS_LOG_INFO ("assigning ip address");
245
246   Ipv4AddressHelper addressAdhoc;
247   addressAdhoc.SetBase ("10.1.1.0", "255.255.255.0");//一个子网的起始地址，掩码
248   Ipv4InterfaceContainer adhocInterfaces;
249   adhocInterfaces = addressAdhoc.Assign (adhocDevices);//将这些地址分配给
    adhocDevices 中的设备，并将分配的地址存储在adhocInterfaces 中
250
251   //用于配置 UDP 连接的 on/off 应用程序，该应用程序以恒定间隔 1.0 的频率发送数据，并在发
    送完数据后立即再次启动发送。
252   OnOffHelper onoff1 ("ns3::UdpSocketFactory",Address ());
253   onoff1.SetAttribute ("OnTime", StringValue
    ("ns3::ConstantRandomVariable[Constant=1.0]"));
254   onoff1.SetAttribute ("OffTime", StringValue
    ("ns3::ConstantRandomVariable[Constant=0.0]"));//应用程序没有关闭时间，持续发送
255
256   //设置接收节点，发送节点，第i+nSinks节点发送给第i个节点。i<nSinks
257   for (int i = 0; i < nSinks; i++)
258   {
259         //设置特定节点作为数据包的接收者（sink），返回一个指向 Socket 对象的指针，该
    对象用于接收从指定地址发送的数据包
260       Ptr<Socket> sink = SetupPacketReceive (adhocInterfaces.GetAddress (i),
    adhocNodes.Get (i));
261
262         //构造一个远程地址，设置为目标地址
263       AddressValue remoteAddress (InetSocketAddress
    (adhocInterfaces.GetAddress (i), port));//port初始化为9
264       onoff1.SetAttribute ("Remote", remoteAddress);
265
266         //设置发送节点
267       Ptr<UniformRandomVariable> var = CreateObject<UniformRandomVariable> ();
268       ApplicationContainer temp = onoff1.Install (adhocNodes.Get (i +
    nSinks));//安装了 On/Off 应用程序到指定的节点，用于模拟数据的发送
```

```
269         //设置应用程序的启动和停止时间
270             temp.Start (Seconds (var->GetValue (50.0,51.0)));//使用随机变量来确定应用程序的启动时间，启动时间在 50 到 51 秒之间。
271         temp.Stop (Seconds (TotalTime));//仿真总时间80s
272     }
273
274     std::stringstream ss;
275     ss << nWifis;
276     std::string nodes = ss.str ();
277
278     std::stringstream ss2;
279     ss2 << nodeSpeed;
280     std::string sNodeSpeed = ss2.str ();
281
282     std::stringstream ss3;
283     ss3 << nodePause;
284     std::string sNodePause = ss3.str ();
285
286     std::stringstream ss4;
287     ss4 << rate;
288     std::string sRate = ss4.str ();
289
290     //NS_LOG_INFO ("Configure Tracing.");
291     //tr_name = tr_name + "_" + m_protocolName +"_" + nodes + "nodes_" + sNodeSpeed + "speed_" + sNodePause + "pause_" + sRate + "rate";
292
293     // AsciiTraceHelper ascii;
294     // Ptr<OutputStreamWrapper> osw = ascii.CreateFileStream ( (tr_name + ".tr").c_str());
295     // wifiPhy.EnableAsciiAll (osw);
296
297     // AsciiTraceHelper ascii;
298     // MobilityHelper::EnableAsciiAll (ascii.CreateFileStream (tr_name + ".mob"));
299
300     uint32_t rxPacketsum = 0;
301     double Delaysum = 0;
302     uint32_t txPacketsum = 0;
303     uint32_t txBytessum = 0;
304     uint32_t rxBytessum = 0;
305     double txTimeFirst = 0;
306     double rxTimeLast = 0;
307     uint32_t lostPacketssum = 0;
308
309     //使用 FlowMonitor 来监视仿真过程中的流量情况
310     FlowMonitorHelper flowmon;
311     Ptr<FlowMonitor> monitor = flowmon.InstallAll();
```

```
312
313
314   NS_LOG_INFO ("Run Simulation.");
315
316   Simulator::Stop (Seconds (TotalTime));//设置仿真总时间
317   Simulator::Run ();//开始仿真运行
318
319   Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier>
    (flowmon.GetClassifier ());
320   std::map<FlowId, FlowMonitor::FlowStats> stats = monitor->GetFlowStats ();//
    获取流量监视器的统计信息
321
322   for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator i = stats.begin
     (); i != stats.end (); ++i)
323   {//遍历流量监控器（FlowMonitor）中的每个流，i->first 表示流的标识（FlowId），i-
    >second 则包含了与该流相关的统计信息
324     Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow (i->first);//根据
    flowid找到与该流相关联的五元组信息（源 IP 地址、目标 IP 地址、源端口、目标端口、协议类
    型）
325     if(t.sourcePort==654){//源端口是否为 654，如果是，则跳过
326       continue;
327     }
328     //累加接收和发送的数据包数量、数据字节数、延迟总和以及丢失的数据包数量
329     rxPacketsum += i->second.rxPackets;
330     txPacketsum += i->second.txPackets;
331     txBytessum += i->second.txBytes;
332     rxBytessum += i->second.rxBytes;
333     Delaysum += i->second.delaySum.GetSeconds();
334     lostPacketssum += i->second.lostPackets;
335     //记录首个发送数据包的时间和最后一个接收数据包的时间
336     if(txTimeFirst == 0)
337     {
338       txTimeFirst = i->second.timeFirstTxPacket.GetSeconds();
339     }
340
341     rxTimeLast = i->second.timeLastRxPacket.GetSeconds();
342   }
343   //将 FlowMonitor 的数据序列化为 XML 文件，并将其保存到名为 tr_name + ".flowmon" 的
    文件中
344   monitor->SerializeToXmlFile ((tr_name + ".flowmon").c_str(), false, false);
345
346   double timeDiff = (rxTimeLast - txTimeFirst);
347
348   std::cout << "\n\n";
349   std::cout << "Total Tx Packets: " << txPacketsum << "\n";
350   std::cout << "Total Rx Packets: " << rxPacketsum << "\n";
351   std::cout << "Total Packets Lost: " << lostPacketssum << "\n";
```

```
352   std::cout << "Throughput: " << ((rxBytessum * 8.0) / timeDiff)/1024<<" Kbps"
    <<"\n";
353   std::cout << "Packets Delivery Ratio: " << (double)((rxPacketsum * 100.0)
    /txPacketsum) << "%" << "\n";
354   std::cout << "Packets Loss Ratio: " << (double)((lostPacketssum * 100.0)
    /txPacketsum) << "%" << "\n";
355   std::cout << "Avg End to End Delay: " << Delaysum/rxPacketsum << "\n";
356
357   std::ofstream myfile;
358   myfile.open ("DATA_AODV_TOPOLOGY.txt", std::ios::app);
359   myfile
360         <<nWifis<<" "
361         <<nSinks<<" "
362         <<lostPacketssum<<" "
363         <<((rxBytessum * 8.0) / timeDiff)/1024<<" "
364         <<(double)((rxPacketsum * 100.0) /txPacketsum)<<" "
365         <<(double)((lostPacketssum * 100.0) /txPacketsum)<<" "
366         <<Delaysum/rxPacketsum<<std::endl;
367   myfile.close();
368
369   Simulator::Destroy ();
370 }
371
372
```