CC-AODV: An Effective Multiple Paths Congestion Control AODV

Kawshik Kumar Paul
Student ID : 1705043
Undergrad Student

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

February 25, 2022

Contents

1 Network Topologies Under Simulation

1.1 Topology for Task A - Wired

A wired topology is used which is look alike Fig 3 which is built with two LAN networks and a Point to Point network connecting the two LANs.
Packet is sent from one LAN network to another LAN network in simulation.

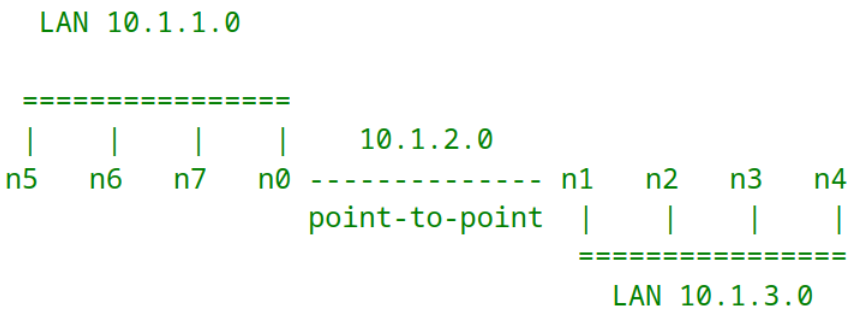


Figure 1: Wired Topology

.

1.2 Topology for Task A - Wireless Low Rate (Static)

IEEE standard 802.15.4 intends to offer the fundamental lower network layers of a type of wireless personal area network (WPAN) which focuses on low-cost, low-speed ubiquitous communication between devices. Here in this simulation, low rate wpan devices (lrwpan devices) are used.

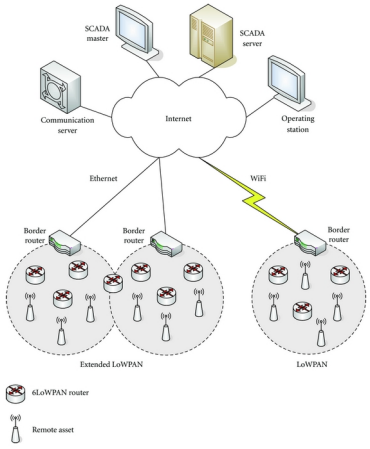


Figure 2: Wireless Low Rate Network Topology

.

1.3 Topology for Task B - MANET AODV

MANET is a dynamic wireless network that can be formed without the need for any pre-existing infrastructure. So, the network topology may be changed dynamically

in an unpredictable manner since nodes are free to move.

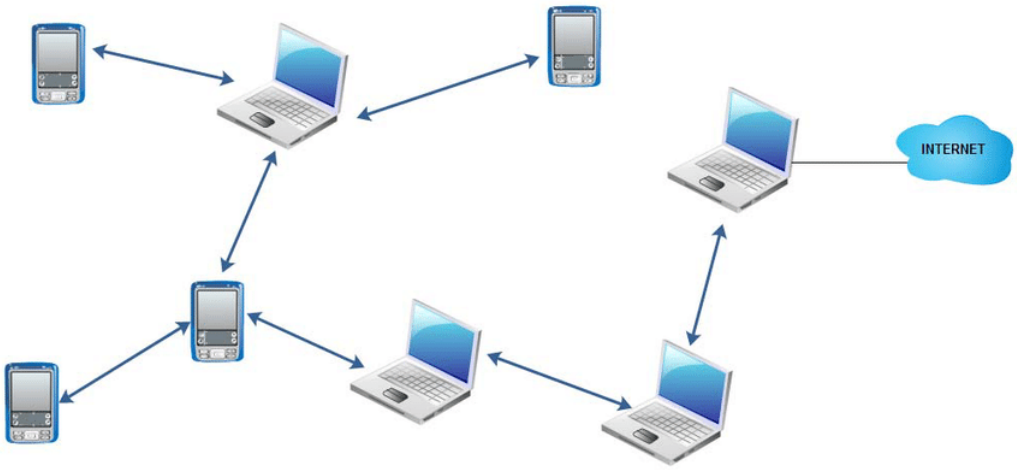


Figure 3: Adhoc Network Topology

.

2 Parameters Under Variation

2.1 Parameters for Task A - Wired

In this network topology, I had to simulate a wired topology. I have varied the following parameters,

1. The number of nodes varied as (20, 40, 60, 80, and 100)
2. The number of flows (10, 20, 30, 40, and 50)
3. The number of packets per second (100, 200, 300, 400, and 500)

2.2 Parameters for Task A - Wireless Low Rate (Static)

In this network topology, I had to simulate a wireless low rate (802.15.4) static network. I have varied the following parameters.

1. The number of nodes varied as (20, 40, 60, 80, and 100)
2. The number of flows (10, 20, 30, 40, and 50)
3. The number of packets per second (100, 200, 300, 400, and 500)
4. Coverage area (square coverage are varying one side as Tx_range, 2 x Tx_range, 3 x Tx_range, 4 x Tx_range, and 5 x Tx_range)

2.3 Parameters for Task B - MANET AODV

According to my selected paper, the number of nodes and sinks are varied to make the simulation.

3 Overview of the Proposed Algorithm

MANETs are characterized by wireless mobile nodes in a network that supports the functionality of self- configurable and independently movable nodes. These nodes in turn can be shape as hosts or clients to construct dynamic networks for package delivery from its source to their respective destinations via dynamic routing path. Regarding the network performance, routers perform a critical role of delivering the data to the appropriate destinations. Engineers have been implementing various routing algorithms to improve wireless network performance. Ad hoc On-Demand Distance Vector (AODV) routing is one of the famous routing algorithms. Tremendous amounts of research on this protocol have been done to improve the performance. In this paper, a new control scheme, named congestion control AODV (CC-AODV), is proposed to manage the described routing condition. With this table entry, the package delivery rates are significantly increased while the package drop rate is decreased, however its implementation causes package overhead. This paper uses NS3 (network simulator 3) for simulation.

3.1 What is AODV?

An Ad Hoc On-Demand Distance Vector (AODV) is a routing protocol designed for wireless and mobile ad hoc networks. This protocol establishes routes to destinations on demand and supports both unicast and multicast routing.

3.1.1 Route Discovery

- In AODV, the route is requested only when the source node wants to send data to the desired destination node. Hence, the source node starts to send RREQ to its neighboring nodes initiating communication.
- When an intermediate node received the RREQ packets, the routing table adds the routing information. If the table already has the entry, then the routers compare the sequence number and hop count with the existing information in the table. If the condition passes, the table will update the routing information in the table.
- After receiving RREQ, node determines whether it is the destination node or not. Moreover, the node can check whether it received the same RREQ packets with the same ID previously. As a result, if a node receives the same ID packets, then it determines whether it requires an update to the table or not.
- Once a destination node receives a RREQ packet, it generates the routing reply packets (RREP). This packet unicasts back to its represented source node and updates the intermediate node routing table. Thus, AODV establishes the routing path.

3.1.2 Route Maintenance

- Once a link has failed or the connection is lost, a router error (RRER) packet is generated and sent to the source node, which in turn requests to establish the new routing path.
- When the source node receives the RRER packets, it starts the flooding broadcast of RREQ packets to reinitiate the route again, allowing AODV to maintain the routing path.

3.2 Issues with AODV

Though sometimes intermediate nodes are too busy to transmit data packages, yet those nodes are used since they are on the shortest path of communication. Nonetheless, when using this approach other nodes that are available are not fully utilized even if they might have low traffic, leading to a lack of bandwidth utilization. As a result, the performance is degraded as the delays in delivering packets increase as well as the number of packets delivered is reduced.

3.3 Proposed CC-AODV Mechanism

To overcome the challenge of AODV, the congestion control CC-ADOV is proposed.

- The proposed CC-ADOV aims to lower the performance degradation caused by the packets congestion while the data is delivered using AODV.
- CC-AODV determines a path for the data by using the congestion counter label. This is achieved by checking how stressed the current node is in a table, and once the RREP package is generated and transmitted through the nodes, the congestion counter adds one to the counter.
- The process of CC-AODV flow chart explains how to establish the route in Fig 4.

3.3.1 Flowchart

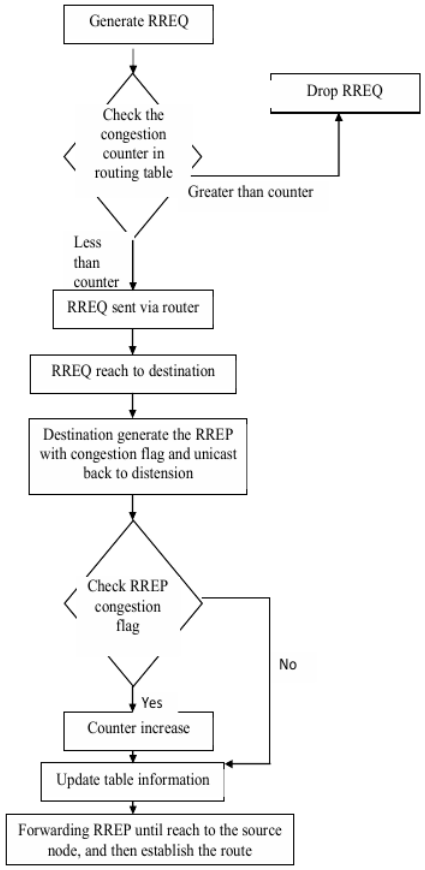


Figure 4: Process of CC-AODV flow chart

.

3.3.2 Algorithm

- First, the source node performs a flooding broadcast RREQ package in the entire network.
- When RREQ package arrives to the intermediate node, the router checks the congestion counter whether it is less than a certain predetermined value.
- If the comparison yields less than the counter, the routing table updates and forwarding to next router; otherwise, the router drops the RREQ package.
- Once the RREQ arrives to the corresponding destination, the RREP is generated by the router. In CC-AODV, the congestion flag is added to the RREP header.
- There are two cases of which a RREP is generated corresponding to a RREQ. One is from the source node to establish the route and the other is from the neighbor nodes to maintain the route.

- When the destination node receives the RREQ from the source node, it generates the RREP with the congestion flag set to true. While the RREP unicast back to the corresponding source node, passing by the intermediate node, the router checks the congestion flag. If it is true, the counter increases; otherwise, the counter keeps the same. Then, the router updates the routing information.

3.3.3 Implementation Guideline

- A 32-bits congestion control flag needs to be added to the RREP header shown in Fig 5
- Once the table is initialized, the congestion counter is generated and initialized to 0.
- Once the node receives the RREP package, the router checks the congestion flag, if the flag is true, then the counter is incremented by 1, otherwise the counter does not change.
- There is one entry in the table called life time. When the life time expires, the counter subtracts 1.
- When a node sends the RRER package back to the source node, the intermediate path between the source node and the destination node is broken from this node. Thus, the counter with this node is subtracted by 1.
- When the node is removed from the network, the congestion counter resets to 0.

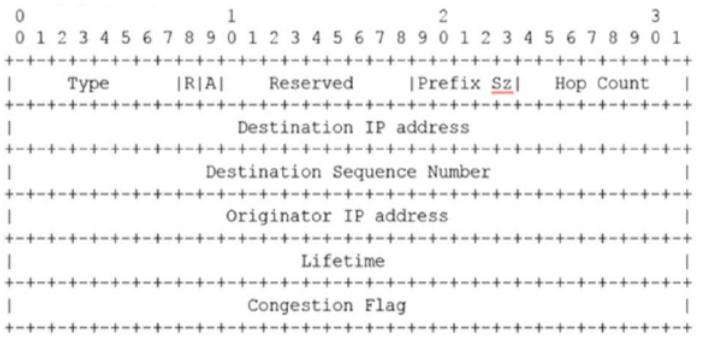


Figure 5: New-RREP Packets

.

4 Modifications Made in the Simulator

For this project, ns-3 network simulator is used and the version of this network simulator is ns-3.35
Modifications are made in 'src/aodv/model' folder. These are the files where AODV models are implemented.

4.1 Add Congestion Flag in RREP Packet Header

aodv-packet.h > {} ns3 > {} aodv > RrepHeader > m_congestionFlag

```
473   */
474   bool operator== (RrepHeader const & o) const;
475 private:
476   uint8_t       m_flags;                ///< A - acknowledgment
      required flag
477   uint8_t       m_prefixSize;         ///< Prefix Size
478   uint8_t             m_hopCount;         ///< Hop Count
479   Ipv4Address   m_dst;                ///< Destination IP Address
480   uint32_t      m_dstSeqNo;           ///< Destination Sequence Number
481   Ipv4Address     m_origin;           ///< Source IP Address
482   uint32_t      m_lifeTime;           ///< Lifetime (in milliseconds)
483 };
484
```

```
477   */
478   bool operator== (RrepHeader const & o) const;
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;       You, seconds ago • Uncommi
488 };
489
```

aodv-packet.h > {} ns3 > {} aodv > RrepHeader > GetCongestionFlag() const

```
433   * \brief Get the lifetime
434   * \return the lifetime
435   */
436   Time GetLifeTime () const;
437
438   // Flags
439   /**
440   * \brief Set the ack required flag
441   * \param f the ack required flag
```

```
433   * \brief Get the lifetime
434   * \return the lifetime
435   */
436   Time GetLifeTime () const;
437
438+  void SetCongestionFlag(int32_t congestionFlag);
439+
440+  int32_t GetCongestionFlag() const;      You, a minute a
441+
442   // Flags
443   /**
444   * \brief Set the ack required flag
445   * \param f the ack required flag
```

aodv-packet.h > {} ns3 > {} aodv > RrepHeader > RrepHeader(uint8_t, uint8_t, Ipv4Address, uint32_t, Ipv4Address, Time, int32_t)

```
344   * \param dstSeqNo the destination sequence number
345   * \param origin the origin IP address
346   * \param lifetime the lifetime
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));
351   /**
352   * \brief Get the type ID.
353   * \return the object TypeId
354   */
355   static TypeId GetTypeId ();
356   TypeId GetInstanceTypeId () const;
```

```
344   * \param dstSeqNo the destination sequence number
345   * \param origin the origin IP address
346   * \param lifetime the lifetime
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);      You, 2 minutes ago
351   /**
352   * \brief Get the type ID.
353   * \return the object TypeId
354   */
355   static TypeId GetTypeId ();
356   TypeId GetInstanceTypeId () const;
```

aodv-packet.cc > {} ns3 > {} aodv > RrepHeader(uint8_t, uint8_t, Ipv4Address, uint32_t, Ipv4Address, Time, int32_t)

```
298
299   RrepHeader::RrepHeader (uint8_t prefixSize, uint8_t hopCount,
      Ipv4Address dst,
300-                          uint32_t dstSeqNo, Ipv4Address origin, Time
                            lifeTime)
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   }
310
```

```
298
299   RrepHeader::RrepHeader (uint8_t prefixSize, uint8_t hopCount,
      Ipv4Address dst,
300+                          uint32_t dstSeqNo, Ipv4Address origin, Time
+                           lifeTime, int32_t congestionFlag)      You,
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   }
311
```

aodv-packet.cc > {} ns3 > {} aodv > RrepHeader(uint8_t, uint8_t, Ipv4Address, uint32_t, Ipv4Address, Time, int32_t)

```
}

uint32_t
RrepHeader::GetSerializedSize () const
{
  return 19;
}

void
RrepHeader::Serialize (Buffer::Iterator i) const
{
  i.WriteU8 (m_flags);
  i.WriteU8 (m_prefixSize);
  i.WriteU8 (m_hopCount);
  WriteTo (i, m_dst);
  i.WriteHtonU32 (m_dstSeqNo);
  WriteTo (i, m_origin);
  i.WriteHtonU32 (m_lifeTime);
}

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 ();

  uint32_t dist = i.GetDistanceFrom (start);
```

```
}

uint32_t
RrepHeader::GetSerializedSize () const
{
  return 19+4;
}

void
RrepHeader::Serialize (Buffer::Iterator i) const
{
  i.WriteU8 (m_flags);
  i.WriteU8 (m_prefixSize);
  i.WriteU8 (m_hopCount);
  WriteTo (i, m_dst);
  i.WriteHtonU32 (m_dstSeqNo);
  WriteTo (i, m_origin);
  i.WriteHtonU32 (m_lifeTime);
  i.WriteHtonU32 (m_congestionFlag);
}

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 ();

  uint32_t dist = i.GetDistanceFrom (start);
```

aodv-packet.cc > {} ns3 > {} aodv > RrepHeader(uint8_t, uint8_t, Ipv4Address, uint32_t, Ipv4Address, Time, int32_t)

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

void
RrepHeader::SetLifeTime (Time t)
{
  m_lifeTime = t.GetMilliSeconds ();
}

Time
RrepHeader::GetLifeTime () const
{
  Time t (MilliSeconds (m_lifeTime));
  return t;
}

void
RrepHeader::SetAckRequired (bool f)
{
  if (f)
```

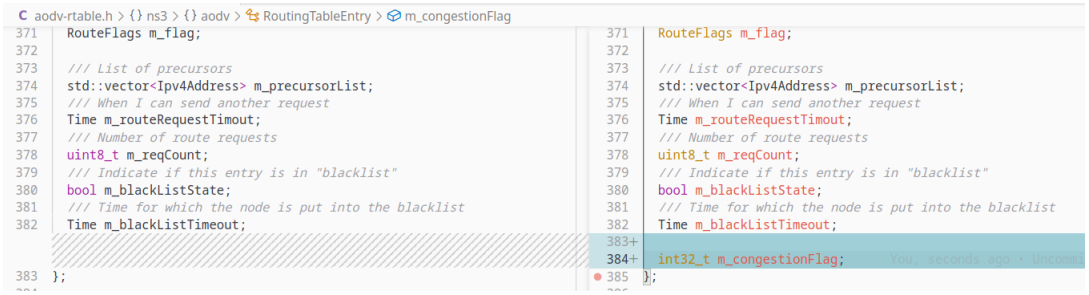
```
  }
  os << " source ipv4 " << m_origin << " lifetime " << m_lifeTime <<
  " congestion flag " << m_congestionFlag
     << " acknowledgment required flag " << (*this).GetAckRequired ()
     ;
}

void
RrepHeader::SetLifeTime (Time t)
{
  m_lifeTime = t.GetMilliSeconds ();
}

Time
RrepHeader::GetLifeTime () const
{
  Time t (MilliSeconds (m_lifeTime));
  return t;
}

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

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

void
RrepHeader::SetAckRequired (bool f)
{
  if (f)
```

aodv-packet.cc > {} ns3 > {} aodv > RrepHeader(uint8_t, uint8_t, Ipv4Address, uint32_t, Ipv4Address, Time, int32_t)

```
{
  return m_prefixSize;
}

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);
}

void
RrepHeader::SetHello (Ipv4Address origin, uint32_t srcSeqNo, Time
lifetime)
{
```

```
{
  return m_prefixSize;
}

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);
}

void
RrepHeader::SetHello (Ipv4Address origin, uint32_t srcSeqNo, Time
lifetime)
{
```

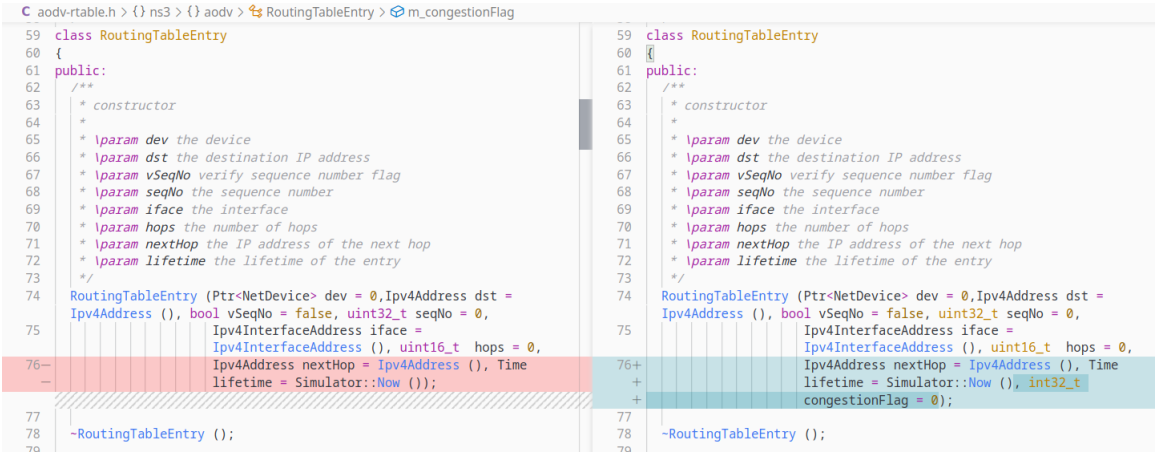
4.2 Congestion Flag in RoutingTableEntry



```
aodv-rtable.h > {} ns3 > {} aodv > RoutingTableEntry > m_congestionFlag
```

```
RouteFlags m_flag;

/// 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 blacklist
Time m_blackListTimeout;
```

```
+ int32_t m_congestionFlag;
};
```



```
aodv-rtable.h > {} ns3 > {} aodv > RoutingTableEntry > m_congestionFlag
```

```
class RoutingTableEntry
{
public:
  /**
   * constructor
   *
   * \param dev the device
   * \param dst the destination IP address
   * \param vSeqNo verify sequence number flag
   * \param seqNo the sequence number
   * \param iface the interface
   * \param hops the number of hops
   * \param nextHop the IP address of the next hop
   * \param lifetime the lifetime of the entry
   */
  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 ());

  ~RoutingTableEntry ();
```

```
+                    Ipv4Address nextHop = Ipv4Address (), Time
+                    lifetime = Simulator::Now (), int32_t
+                    congestionFlag = 0);

  ~RoutingTableEntry ();
```

```
aodv-rtable.cc > {} ns3 > {} aodv
```

```
RoutingTableEntry::RoutingTableEntry (Ptr<NetDevice> dev,
Ipv4Address dst, bool vSeqNo, uint32_t seqNo,
-                                  Ipv4InterfaceAddress iface,
-                                  uint16_t hops, Ipv4Address
-                                  nextHop, Time lifetime)
  : m_ackTimer (Timer::CANCEL_ON_DESTROY),
    m_validSeqNo (vSeqNo),
    m_seqNo (seqNo),
    m_hops (hops),
    m_lifeTime (lifetime + Simulator::Now ()),
    m_iface (iface),
    m_flag (VALID),
    m_reqCount (0),
    m_blackListState (false),
    m_blackListTimeout (Simulator::Now ())
{
  m_ipv4Route = Create<Ipv4Route> ();
  m_ipv4Route->SetDestination (dst);
```

```
+                                  Ipv4InterfaceAddress iface,
+                                  uint16_t hops, Ipv4Address
+                                  nextHop, Time lifetime,
+                                  int32_t congestionFlag)
  : m_ackTimer (Timer::CANCEL_ON_DESTROY),
    m_validSeqNo (vSeqNo),
    m_seqNo (seqNo),
    m_hops (hops),
    m_lifeTime (lifetime + Simulator::Now ()),
    m_iface (iface),
    m_flag (VALID),
    m_reqCount (0),
    m_blackListState (false),
    m_blackListTimeout (Simulator::Now ())
{
  m_ipv4Route = Create<Ipv4Route> ();
  m_ipv4Route->SetDestination (dst);
```

4.3 Add Congestion Counter and Max Count

```
aodv-routing-protocol.cc > MAX_CONGESTION_COUNT
```

```
#include <limits>

namespace ns3 {

NS_LOG_COMPONENT_DEFINE ("AodvRoutingProtocol");

namespace aodv {
NS_OBJECT_ENSURE_REGISTERED (RoutingProtocol);
```

```
#include <limits>

+#define MAX_CONGESTION_COUNT 4

namespace ns3 {

NS_LOG_COMPONENT_DEFINE ("AodvRoutingProtocol");

namespace aodv {
NS_OBJECT_ENSURE_REGISTERED (RoutingProtocol);
```

aodv-routing-protocol.h > {} ns3 > {} aodv > RoutingProtocol > m_congestionCounter

```
/// 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;
```

```
/// 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;
```

aodv-routing-protocol.cc > {} ns3 > {} aodv > RoutingProtocol()

```
m_rreqIdCache (m_pathDiscoveryTime),
m_dpd (m_pathDiscoveryTime),
m_nb (m_helloInterval),
m_rreqCount (0),
m_rerrCount (0),
m_htimer (Timer::CANCEL_ON_DESTROY),
m_rreqRateLimitTimer (Timer::CANCEL_ON_DESTROY),
m_rerrRateLimitTimer (Timer::CANCEL_ON_DESTROY),
m_lastBcastTime (Seconds (0))
{
m_nb.SetCallback (MakeCallback (&
RoutingProtocol::SendRerrWhenBreaksLinkToNextHop, this));
}
```

```
m_rreqIdCache (m_pathDiscoveryTime),
m_dpd (m_pathDiscoveryTime),
m_nb (m_helloInterval),
m_rreqCount (0),
m_rerrCount (0),
+ m_congestionCounter(0),
m_htimer (Timer::CANCEL_ON_DESTROY),
m_rreqRateLimitTimer (Timer::CANCEL_ON_DESTROY),
m_rerrRateLimitTimer (Timer::CANCEL_ON_DESTROY),
m_lastBcastTime (Seconds (0))
{
m_nb.SetCallback (MakeCallback (&
RoutingProtocol::SendRerrWhenBreaksLinkToNextHop, this));
+ // std::cout<<"Congestion Counter = "<<m_congestionCounter<<" and
+ Max Congestion Count set to "<<MAX_CONGESTION_COUNT<<std::endl;
}
```

aodv-routing-protocol.cc > {} ns3 > {} aodv > GetTypeId(void)

```
.AddAttribute ("MyRouteTimeout", "Value of lifetime field in
RREP generating by this node = 2 * max(ActiveRouteTimeout,
PathDiscoveryTime)",
                TimeValue (Seconds (11.2)),
                MakeTimeAccessor (&
                RoutingProtocol::m_myRouteTimeout),
                MakeTimeChecker ())
.AddAttribute ("BlackListTimeout", "Time for which the node is
put into the blacklist = RreqRetries * NetTraversalTime",
                TimeValue (Seconds (5.6)),
                MakeTimeAccessor (&
```

```
.AddAttribute ("MyRouteTimeout", "Value of lifetime field in
RREP generating by this node = 2 * max(ActiveRouteTimeout,
PathDiscoveryTime)",
                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 blacklist = RreqRetries * NetTraversalTime",
                TimeValue (Seconds (5.6)),
                MakeTimeAccessor (&
```

4.4 Drop RREQ Counter Greater than Max Count

aodv-routing-protocol.cc > {} ns3 > {} aodv > RecvRequest(Ptr<Packet>, Ipv4Address, Ipv4Address)

```
*  Node checks to determine whether it has received a RREQ with
the same Originator IP Address and RREQ ID.
*  If such a RREQ has been received, the node silently discards
the newly received RREQ.
*/
if (m_rreqIdCache.IsDuplicate (origin, id))
  {
    NS_LOG_DEBUG ("Ignoring RREQ due to duplicate");
    return;
  }
// Increment RREQ hop count
uint8_t hop = rreqHeader.GetHopCount () + 1;
```

```
*  Node checks to determine whether it has received a RREQ with
the same Originator IP Address and RREQ ID.
*  If such a RREQ has been received, the node silently discards
the newly received RREQ.
*/
if (m_rreqIdCache.IsDuplicate (origin, id))
  {
    NS_LOG_DEBUG ("Ignoring RREQ due to duplicate");
    return;
  }

+ 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;
+ }
// Increment RREQ hop count
uint8_t hop = rreqHeader.GetHopCount () + 1;
```

4.5 Counter Increment or Decrement and Flag Operations

aodv-routing-protocol.cc > {} ns3 > {} aodv > SendReply(RreqHeader const &, RoutingTableEntry const &)

```
1413
1414 void
1415 RoutingProtocol::SendReply (RreqHeader const & rreqHeader,
     RoutingTableEntry const & toOrigin)
1416 {
1417   NS_LOG_FUNCTION (this << toOrigin.GetDestination ());
1418   /*
1419    * Destination node MUST increment its own sequence number by one
      if the sequence number in the RREQ packet is equal to that
1420    * incremented value. Otherwise, the destination does not change
      its sequence number before generating the  RREP message.
1421    */
1422   if (!rreqHeader.GetUnknownSeqno () && (rreqHeader.GetDstSeqno ()
      == m_seqNo + 1))
1423     {
1424       m_seqNo++;
1425     }
1426   RrepHeader rrepHeader ( /*prefixSize=*/ 0, /*hops=*/ 0, /*dst=*/
      rreqHeader.GetDst (),
1427-                                          /*dstSeqNo=*/ m_seqNo, /
     -                                          *origin=*/ toOrigin.
     -                                          GetDestination (), /
     -                                          *lifeTime=*/
     -                                          m_myRouteTimeout);
```

```
1427
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;
```

aodv-routing-protocol.cc > {} ns3 > {} aodv > RecvReply(Ptr<Packet>, Ipv4Address, Ipv4Address)

```
1528     ProcessHello (rrepHeader, receiver);
1529     return;
1530   }
1531
1532   /*
1533    * If the route table entry to the destination is created or
      updated, then the following actions occur:
1534    * -  the route is marked as active,
1535    * -  the destination sequence number is marked as valid,
1536    * -  the next hop in the route entry is assigned to be the node
      from which the RREP is received,
1537    *    which is indicated by the source IP address field in the IP
      header,
1538    * -  the hop count is set to the value of the hop count from
      RREP message + 1
1539    * -  the expiry time is set to the current time plus the value
      of the Lifetime in the RREP message,
1540    * -  and the destination sequence number is the Destination
      Sequence Number in the RREP message.
1541    */
1542   Ptr<NetDevice> dev = m_ipv4->GetNetDevice
      (m_ipv4->GetInterfaceForAddress (receiver));
1543   RoutingTableEntry newEntry (/*device=*/ dev, /*dst=*/ dst, /
      *validSeqNo=*/ true, /*seqno=*/ rrepHeader.GetDstSeqno (),
1544                                          /*iface=*/
     -                                          m_ipv4->GetAddress
     -                                          (m_ipv4->GetInterfaceForAd
     -                                          dress (receiver), 0),/
     -                                          *hop=*/ hop,
1545-                                          /*nextHop=*/ sender, /
     -                                          *lifeTime=*/ rrepHeader.
     -                                          GetLifeTime ());
```

```
1544     ProcessHello (rrepHeader, receiver);
1545     return;
1546   }
1547
1548+ if(rrepHeader.GetCongestionFlag()==1){
1549+   m_congestionCounter++;
1550+   // std::cout<<"Congestion Counter ++"<<std::endl;
1551+ }
1552   /*
1553    * If the route table entry to the destination is created or
      updated, then the following actions occur:
1554    * -  the route is marked as active,
1555    * -  the destination sequence number is marked as valid,
1556    * -  the next hop in the route entry is assigned to be the node
      from which the RREP is received,
1557    *    which is indicated by the source IP address field in the IP
      header,
1558    * -  the hop count is set to the value of the hop count from
      RREP message + 1
1559    * -  the expiry time is set to the current time plus the value
      of the Lifetime in the RREP message,
1560    * -  and the destination sequence number is the Destination
      Sequence Number in the RREP message.
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->GetInterfaceForAd
     +                                          dress (receiver), 0),/
     +                                          *hop=*/ hop,
1565+                                          /*nextHop=*/ sender, /
     +                                          *lifeTime=*/ rrepHeader.
     +                                          GetLifeTime (),
     +                                          rrepHeader.
     +                                          GetCongestionFlag());
```

aodv-routing-protocol.cc > {} ns3 > {} aodv > RecvReply(Ptr<Packet>, Ipv4Address, Ipv4Address)

```
1625     m_routingTable.LookupRoute (toOrigin.GetNextHop (),
         toNextHopToOrigin);
1626     toNextHopToOrigin.InsertPrecursor (toDst.GetNextHop ());
1627     m_routingTable.Update (toNextHopToOrigin);
1628   }
1629   SocketIpTtlTag tag;
1630   p->RemovePacketTag (tag);
1631   if (tag.GetTtl () < 2)
1632     {
1633       NS_LOG_DEBUG ("TTL exceeded. Drop RREP destination " << dst
          << " origin " << rrepHeader.GetOrigin ());
1634       return;
1635     }
```

```
1645     m_routingTable.LookupRoute (toOrigin.GetNextHop (),
         toNextHopToOrigin);
1646     toNextHopToOrigin.InsertPrecursor (toDst.GetNextHop ());
1647     m_routingTable.Update (toNextHopToOrigin);
1648   }
1649   SocketIpTtlTag tag;
1650   p->RemovePacketTag (tag);
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+      }      You, 5 minutes ago • Uncommitted changes
1658+
1659       return;
1660     }
```

```
aodv-routing-protocol.cc > { } ns3 > { } aodv > RecvError(Ptr<Packet>, Ipv4Address)
1742          }
1743        }
1744      if (rerrHeader.GetDestCount () != 0)
1745        {
1746          TypeHeader typeHeader (AODVTYPE_RERR);
1747          Ptr<Packet> packet = Create<Packet> ();
1748          SocketIpTtlTag tag;
1749          tag.SetTtl (1);
1750          packet->AddPacketTag (tag);
1751          packet->AddHeader (rerrHeader);
1752          packet->AddHeader (typeHeader);
1753          SendRerrMessage (packet, precursors);
1754        }
1755      m_routingTable.InvalidateRoutesWithDst (unreachable);
1756  }
```

```
1767          }
1768        }
1769      if (rerrHeader.GetDestCount () != 0)
1770        {
1771          TypeHeader typeHeader (AODVTYPE_RERR);
1772          Ptr<Packet> packet = Create<Packet> ();
1773          SocketIpTtlTag tag;
1774          tag.SetTtl (1);
1775          packet->AddPacketTag (tag);
1776          packet->AddHeader (rerrHeader);
1777          packet->AddHeader (typeHeader);
1778          SendRerrMessage (packet, precursors);
1779        }
1780      m_routingTable.InvalidateRoutesWithDst (unreachable);
1781+     if(m_congestionCounter > 0){
1782+       m_congestionCounter--;
1783+       // std::cout<<"Congestion Counter -- (RERR)"<<std::endl;
1784+     }       You, 6 minutes ago • Uncommitted changes
1785  }
```

5 Paper Result vs Result of My Task B

5.1 Plot Graph

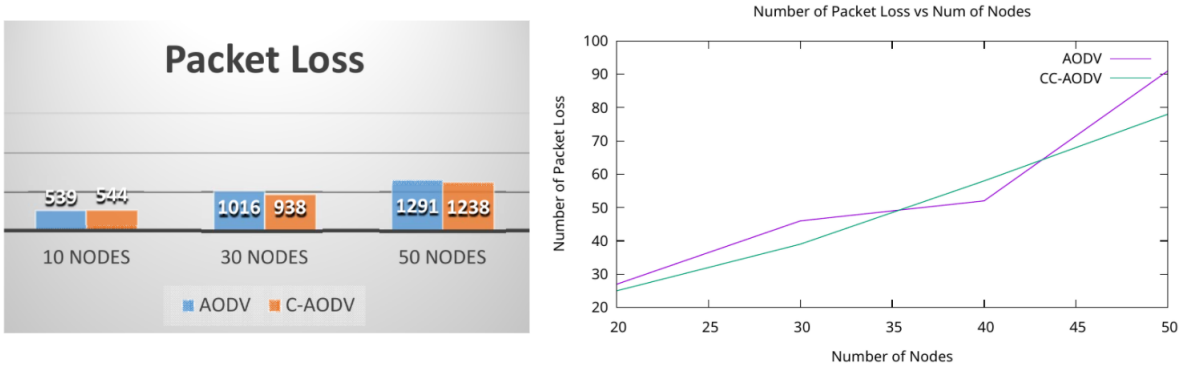


Figure 6: Left: Paper Result, Right: My Result

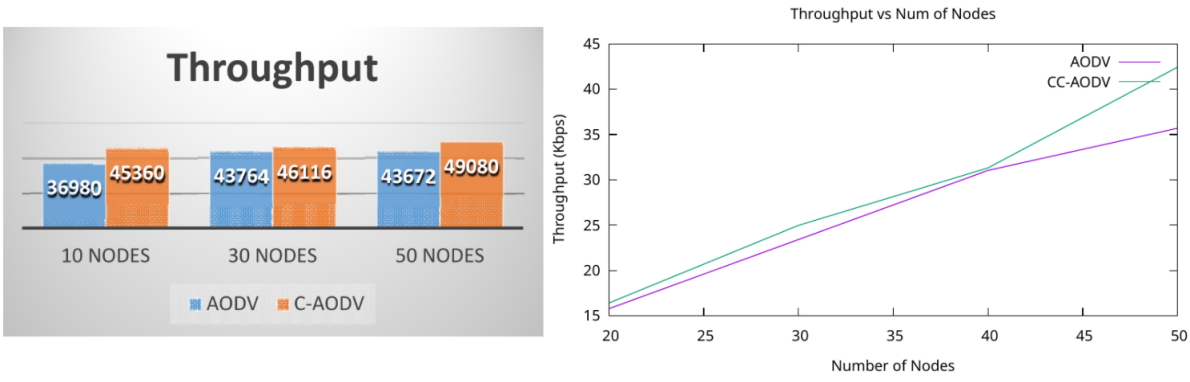


Figure 7: Left: Paper Result, Right: My Result

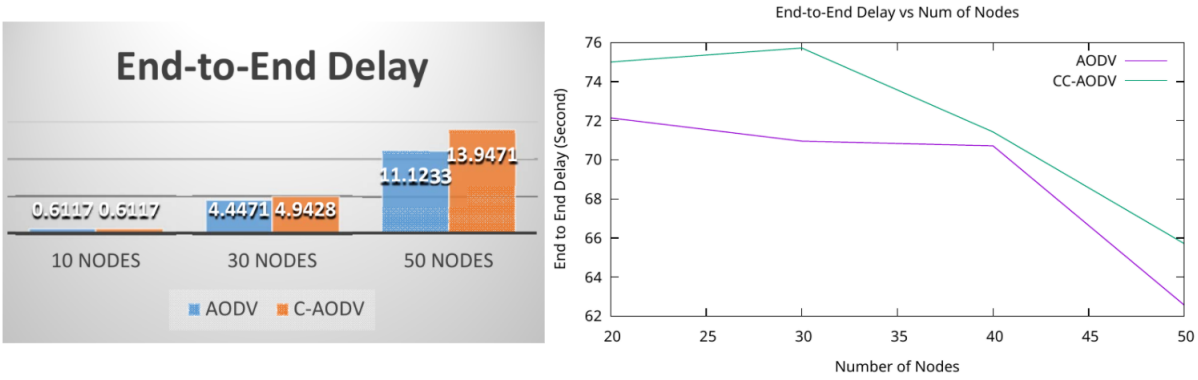


Figure 8: Left: Paper Result, Right: My Result

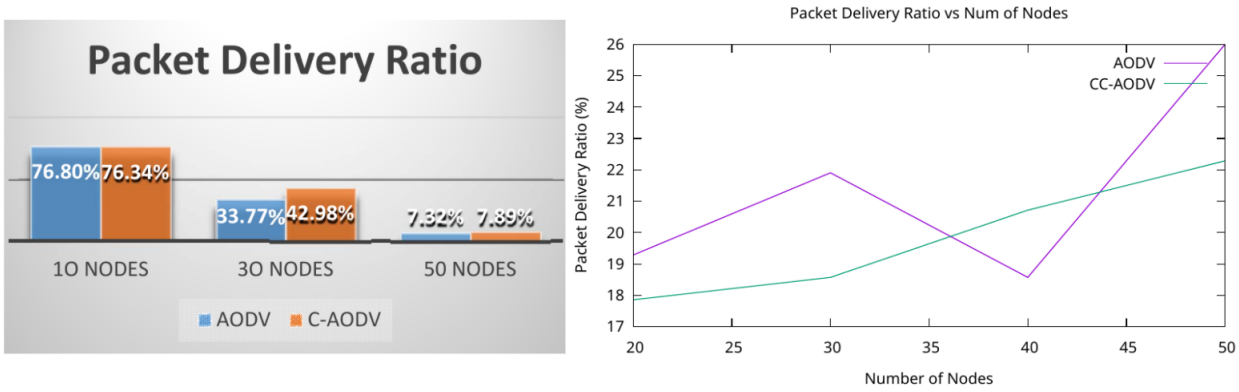


Figure 9: Left: Paper Result, Right: My Result

5.2 My Observation about the Result

- Here we can see that the packet loss graph almost followed the one of the paper. But basic AODV and CC-AODV are nearly same. Not much difference.
- We can notice a significant difference in the throughput graph. Throughput is significantly increased.
- End to End delay is greater in CC-AODV than basic AODV in both paper simulation and my simulation. In my simulation I have measured the average End to End delay, so the values are not matching with the numeric value range with the paper simulation.
- Packet delivery ratio didn't follow the paper very much, but if we look closely, when the number of nodes are near 30, the result are somewhat very near.
- In CC-AODV, there is a counter whose value determines whether RREQ packet is sent or dropped. A congestion counter level is used here. I have experimented with some congestion counter level values. I have observed that the congestion counter value influences the packet delivery rate or throughput

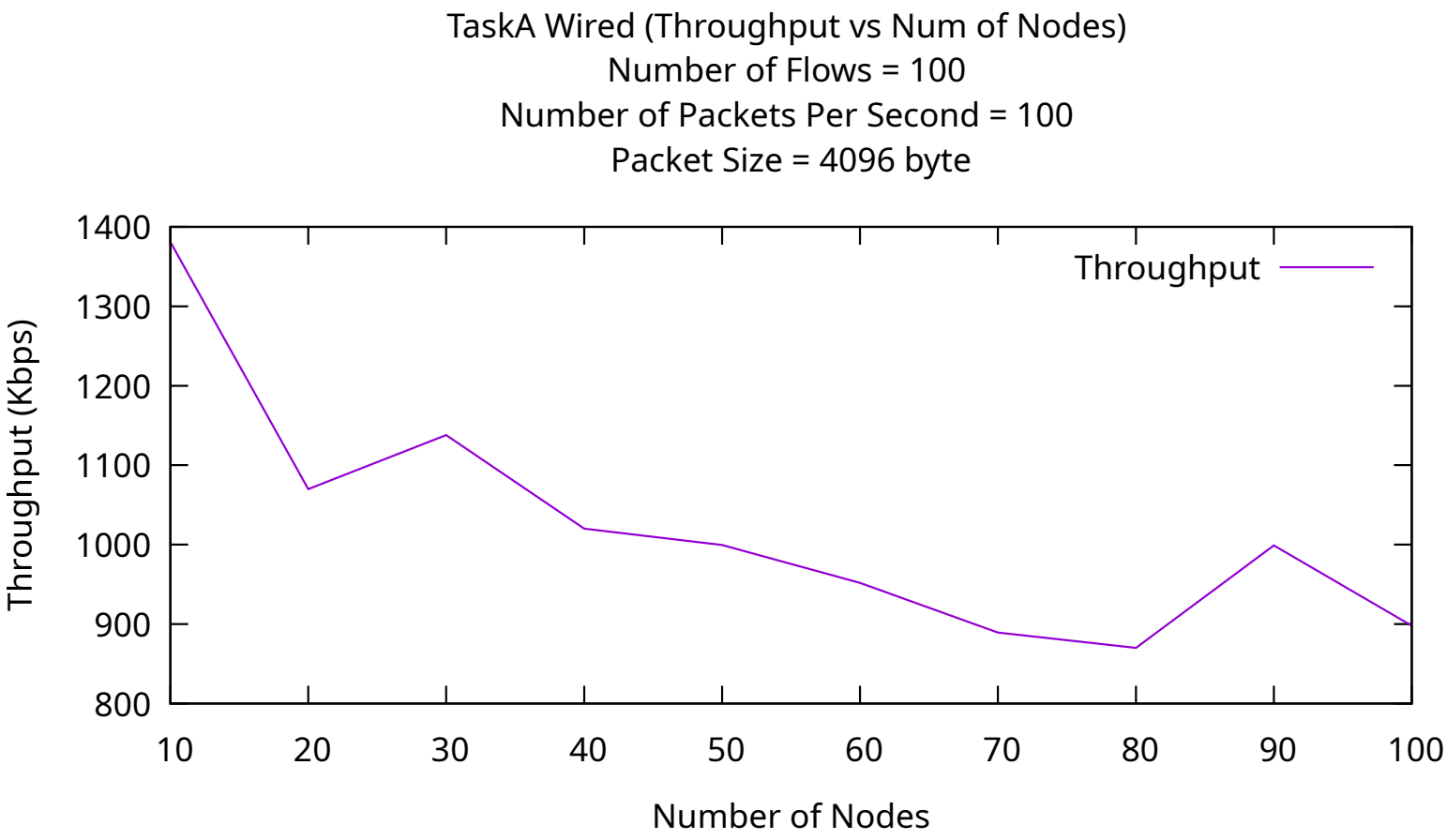
rate or others. If flow is near 30, and the counter level is set near 4, the result is better than the basic AODV. Increasing the level from 4 to higher will make the result of CC-AODV like the basic AODV. But if the value is lesser than 4, it will not come out with good result. So the tuning is very important here to increase the packet delivery rate or throughput rate or others.

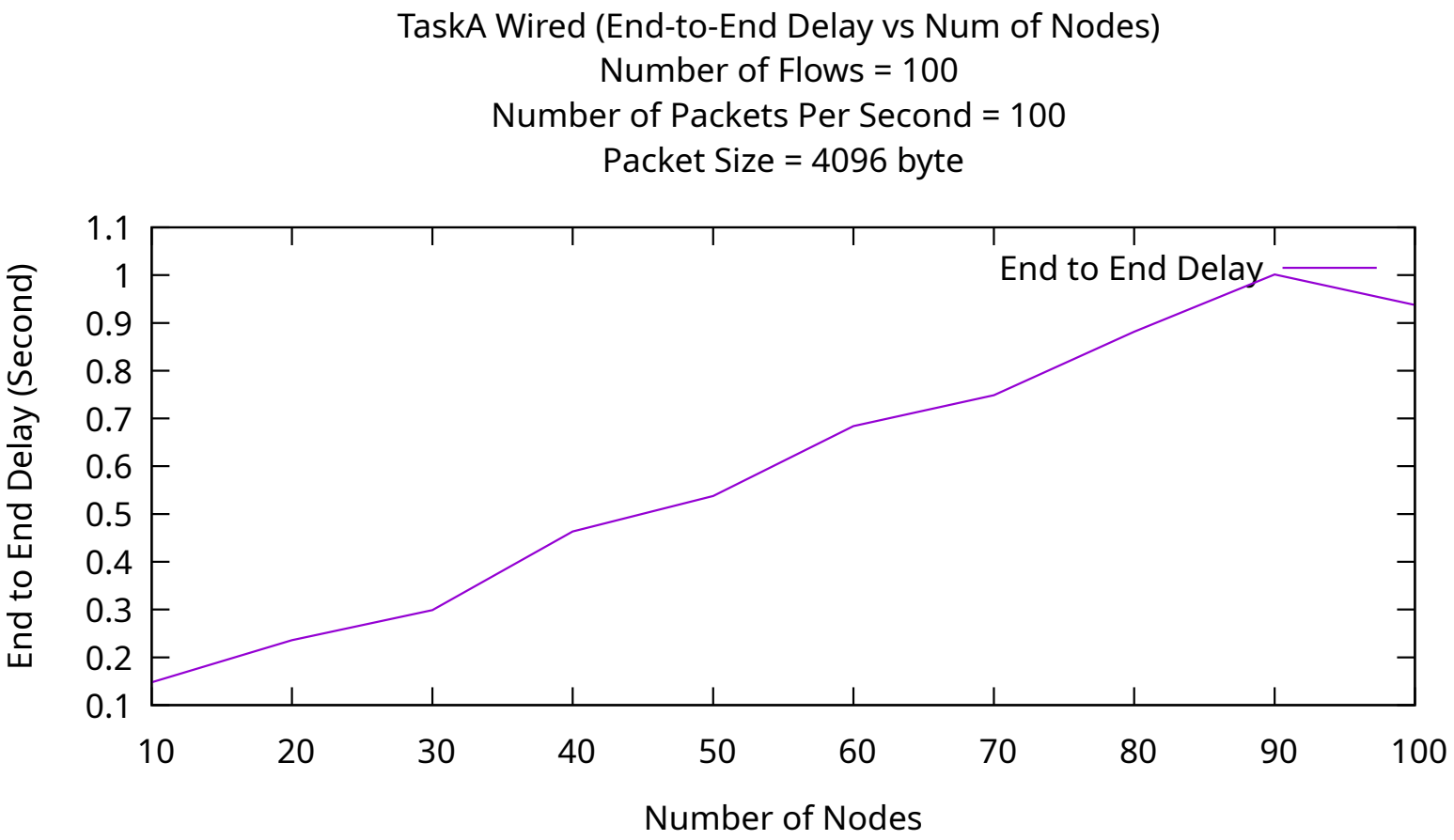
5.3 Minor Improvement Suggestion from my side

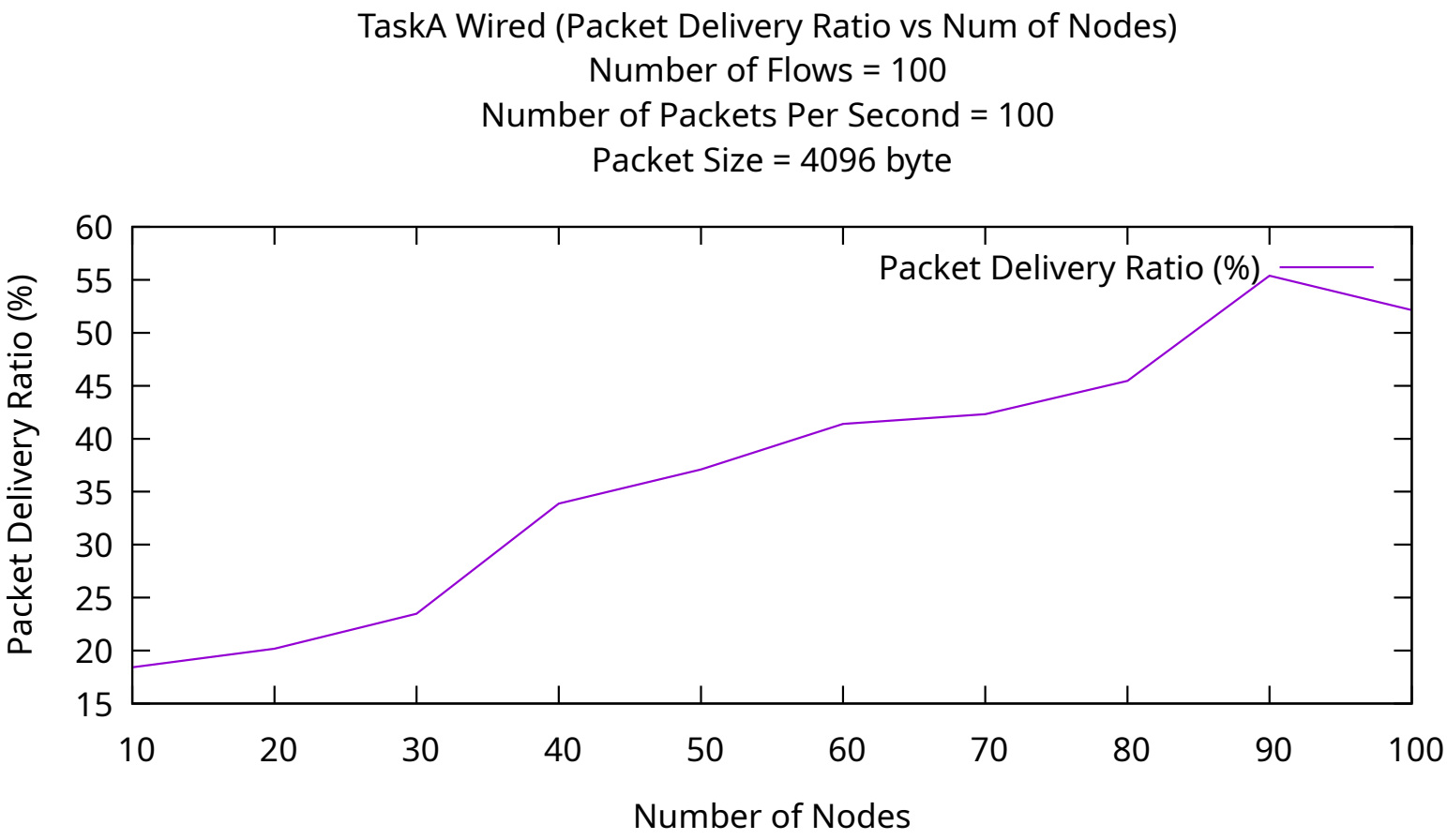
In this paper, they suggested to add a flag in RREP packet header. The flag is only to set true or false, nothing else. But they took a 32 bit flag unnecessarily which is a waste of memory. A boolean variable can be used here. It can be improved.

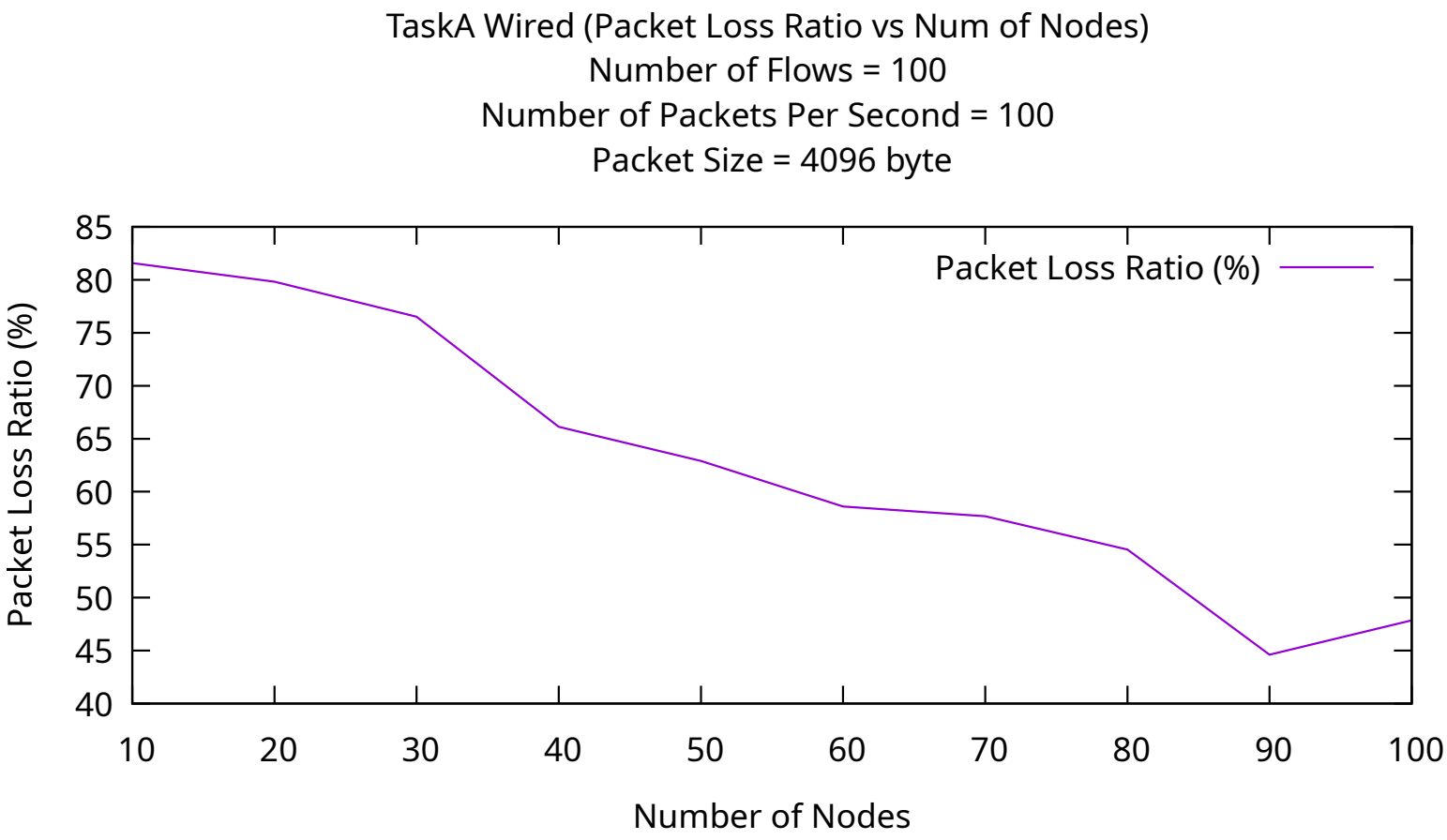
6 Results For Task A - Wired Topology

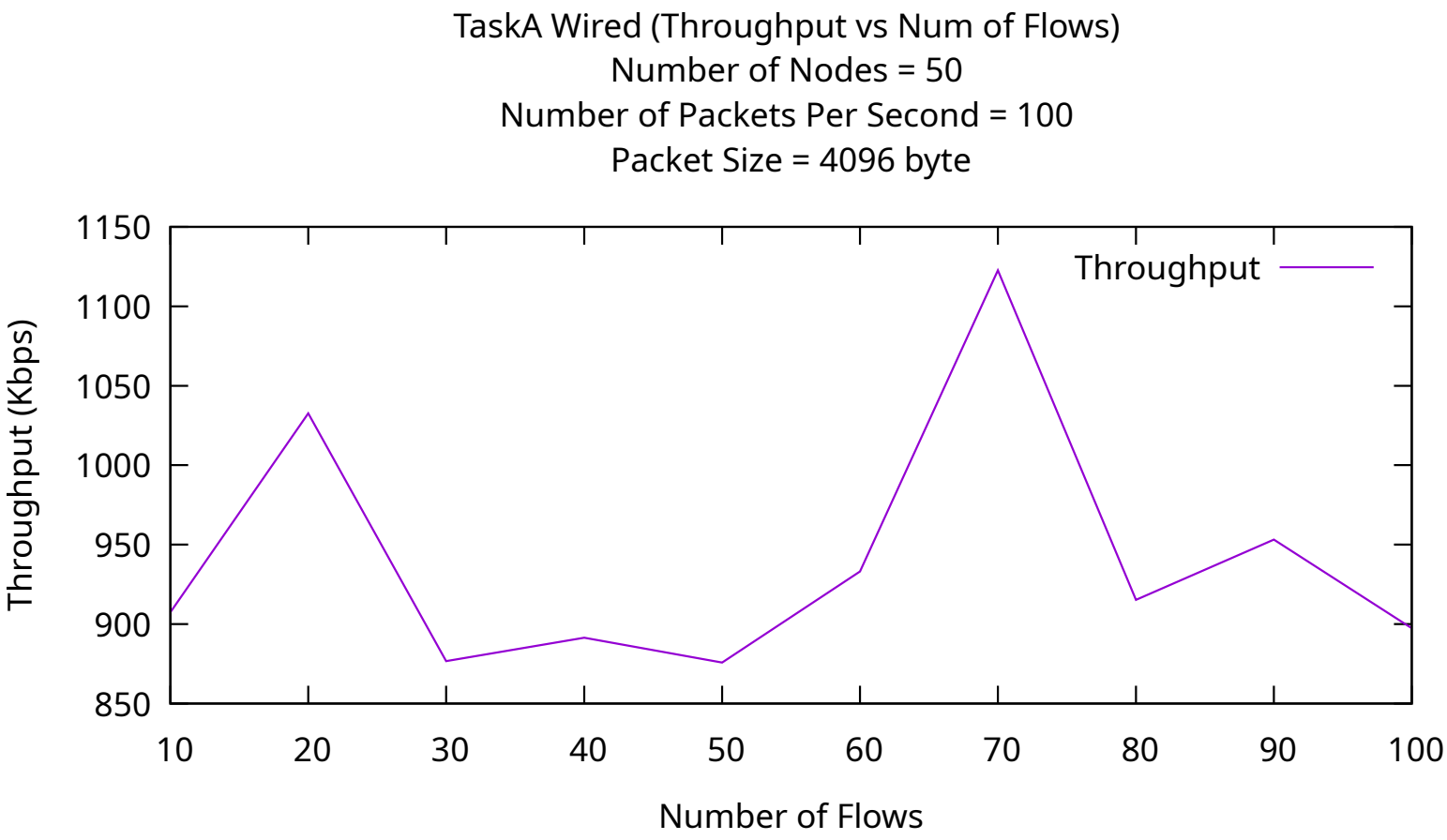
6.1 Plot Graph

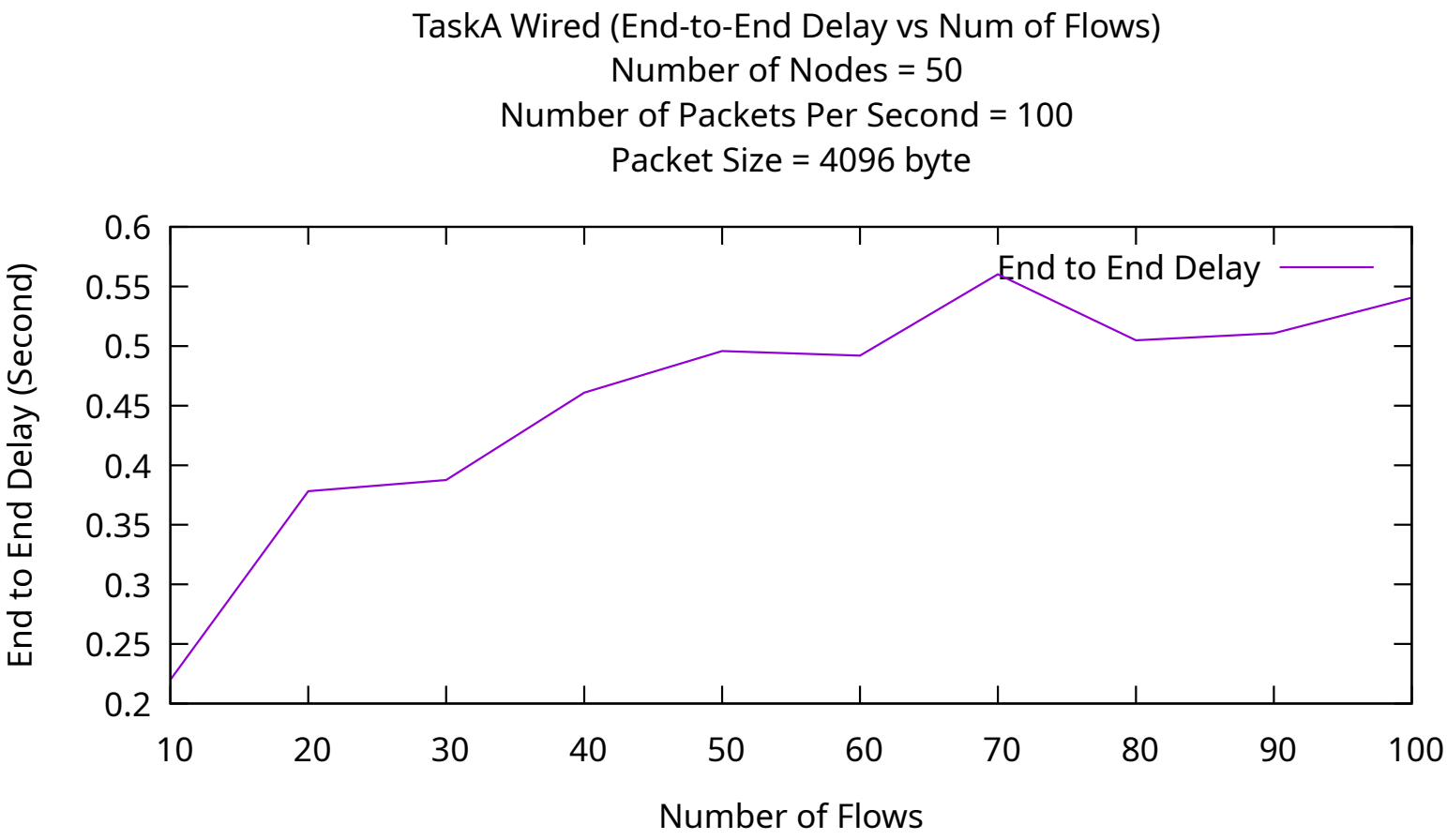

TaskA Wired (Throughput vs Num of Nodes)
Number of Flows = 100
Number of Packets Per Second = 100
Packet Size = 4096 byte
Throughput
Throughput (Kbps)
1400
1300
1200
1100
1000
900
800
10
20
30
40
50
60
70
80
90
100
Number of Nodes

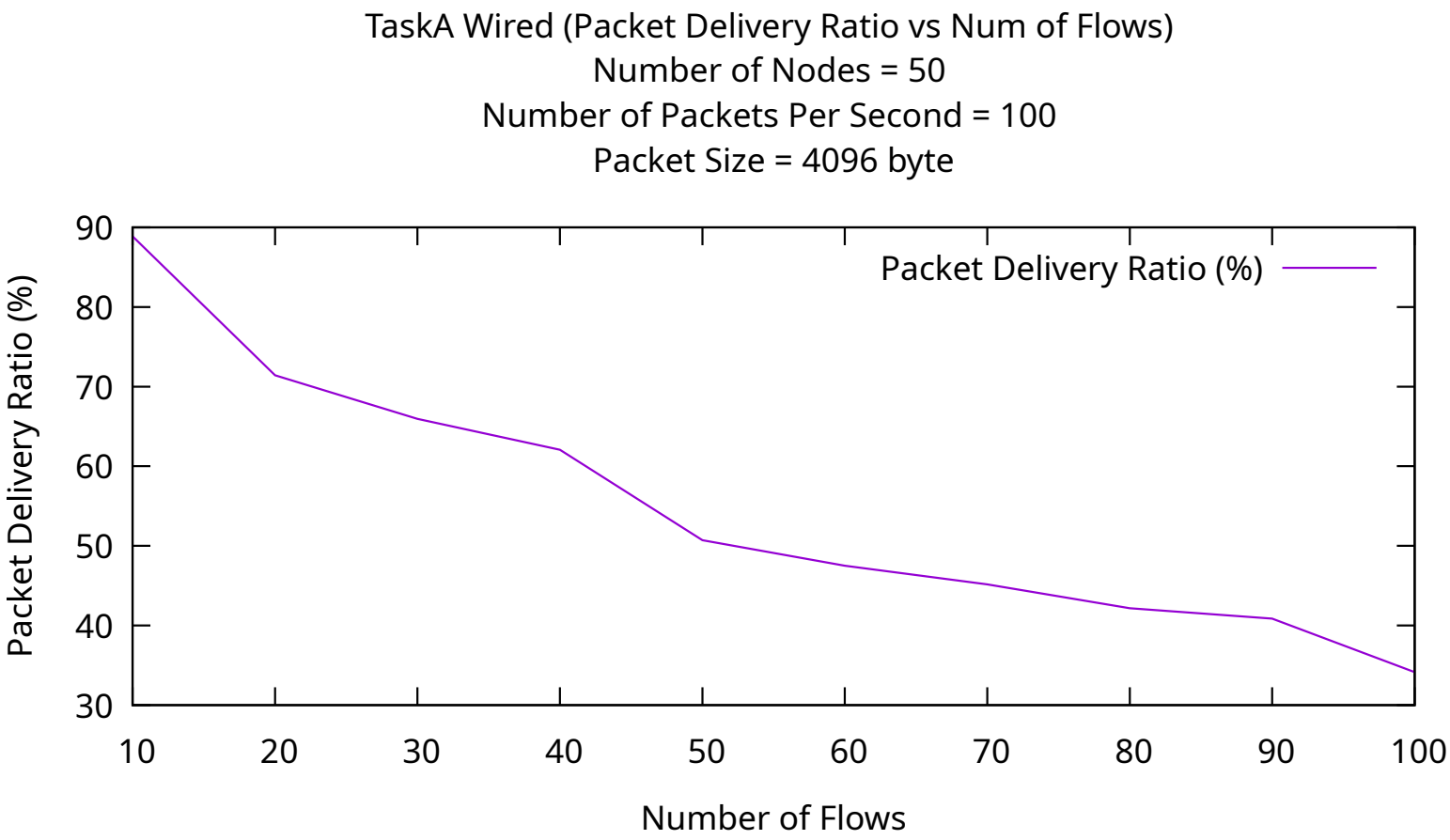

TaskA Wired (End-to-End Delay vs Num of Nodes)
Number of Flows = 100
Number of Packets Per Second = 100
Packet Size = 4096 byte
End to End Delay (Second)
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
10
20
30
40
50
60
70
80
90
100
Number of Nodes
End to End Delay

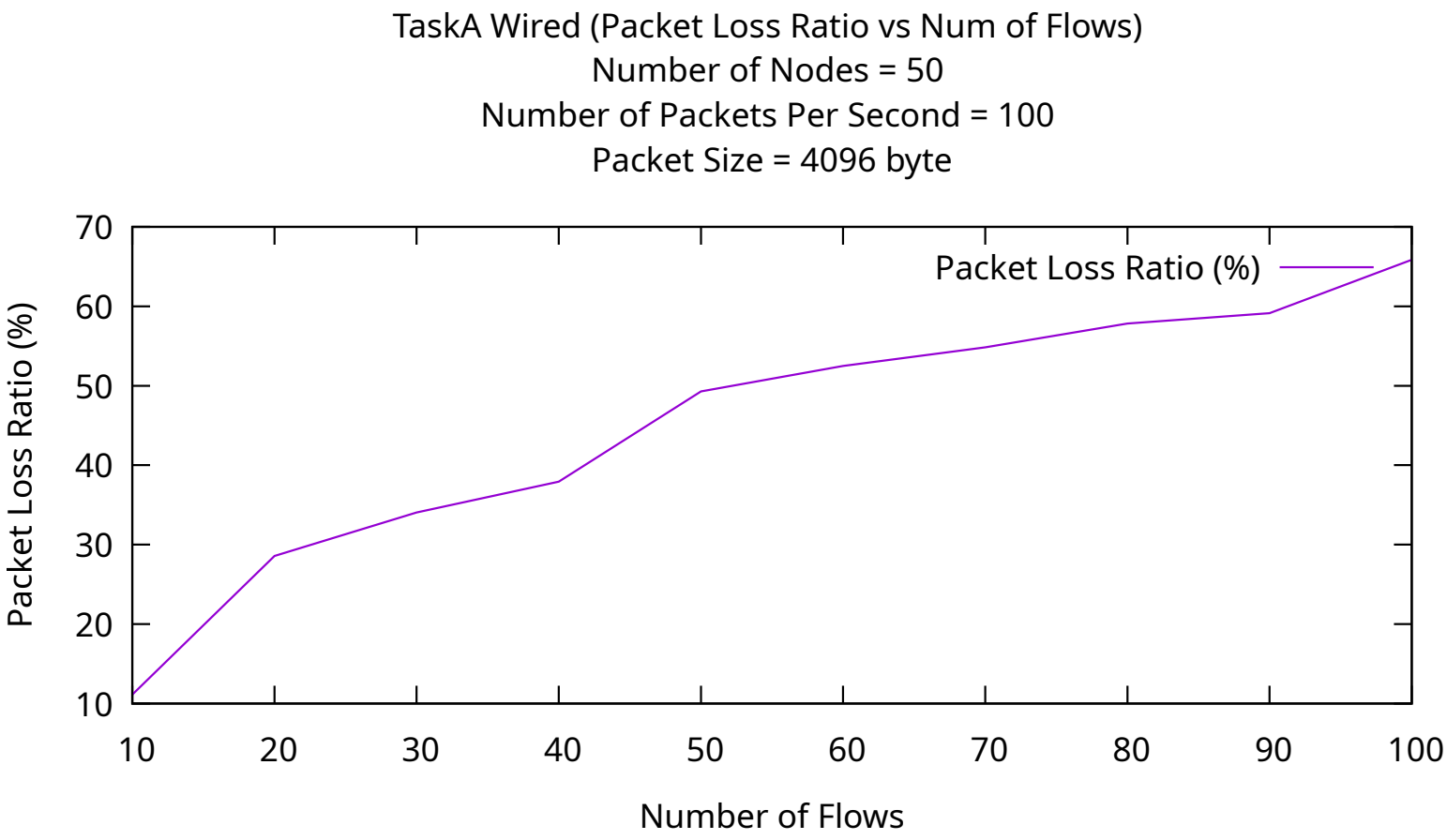

TaskA Wired (Packet Delivery Ratio vs Num of Nodes)
Number of Flows = 100
Number of Packets Per Second = 100
Packet Size = 4096 byte
Packet Delivery Ratio (%)
60
55
50
45
40
35
30
25
20
15
10
20
30
40
50
60
70
80
90
100
Number of Nodes
Packet Delivery Ratio (%)

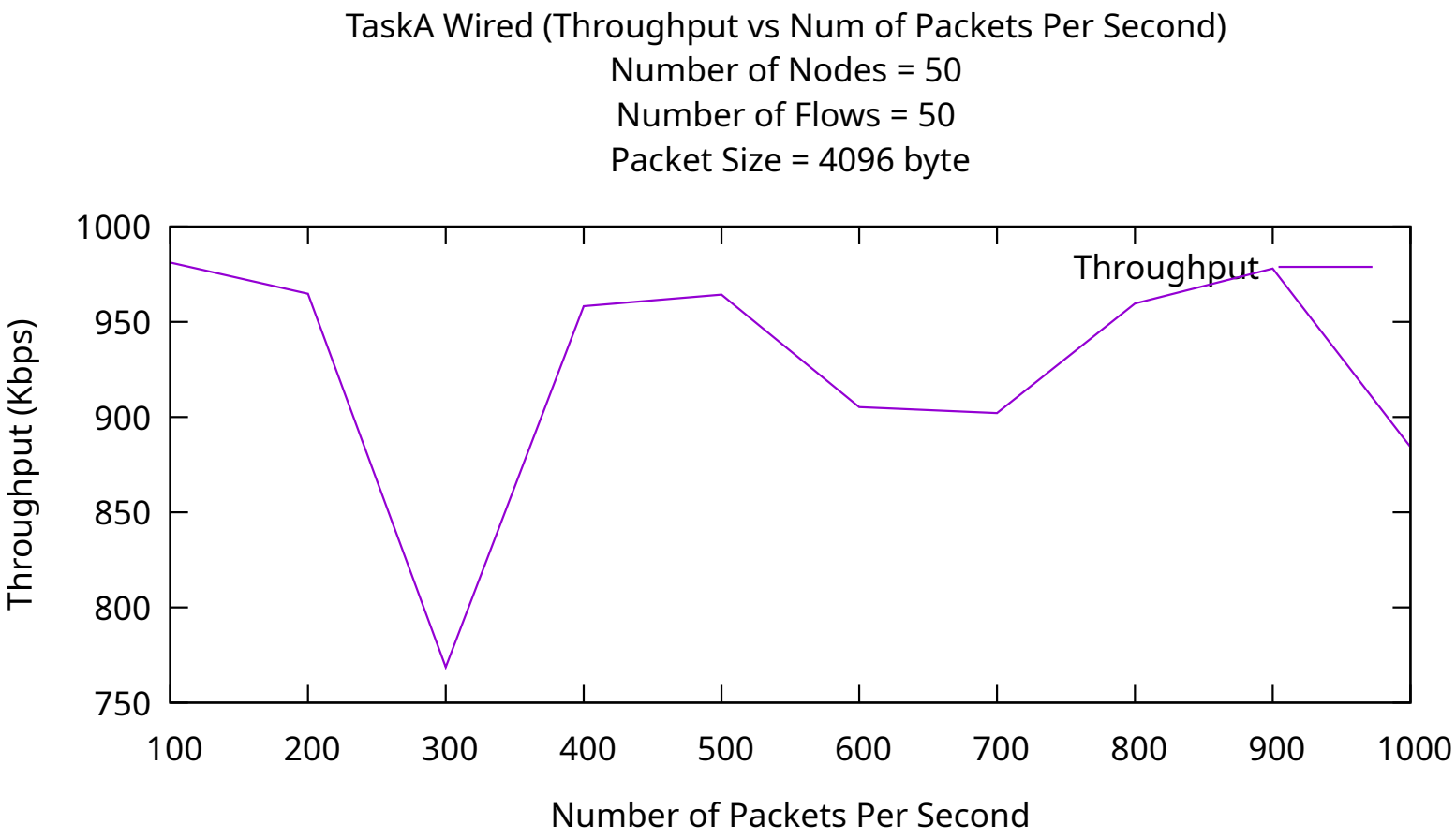

TaskA Wired (Packet Loss Ratio vs Num of Nodes)
Number of Flows = 100
Number of Packets Per Second = 100
Packet Size = 4096 byte
Packet Loss Ratio (%)
85
80
75
70
65
60
55
50
45
40
10
20
30
40
50
60
70
80
90
100
Number of Nodes
Packet Loss Ratio (%)

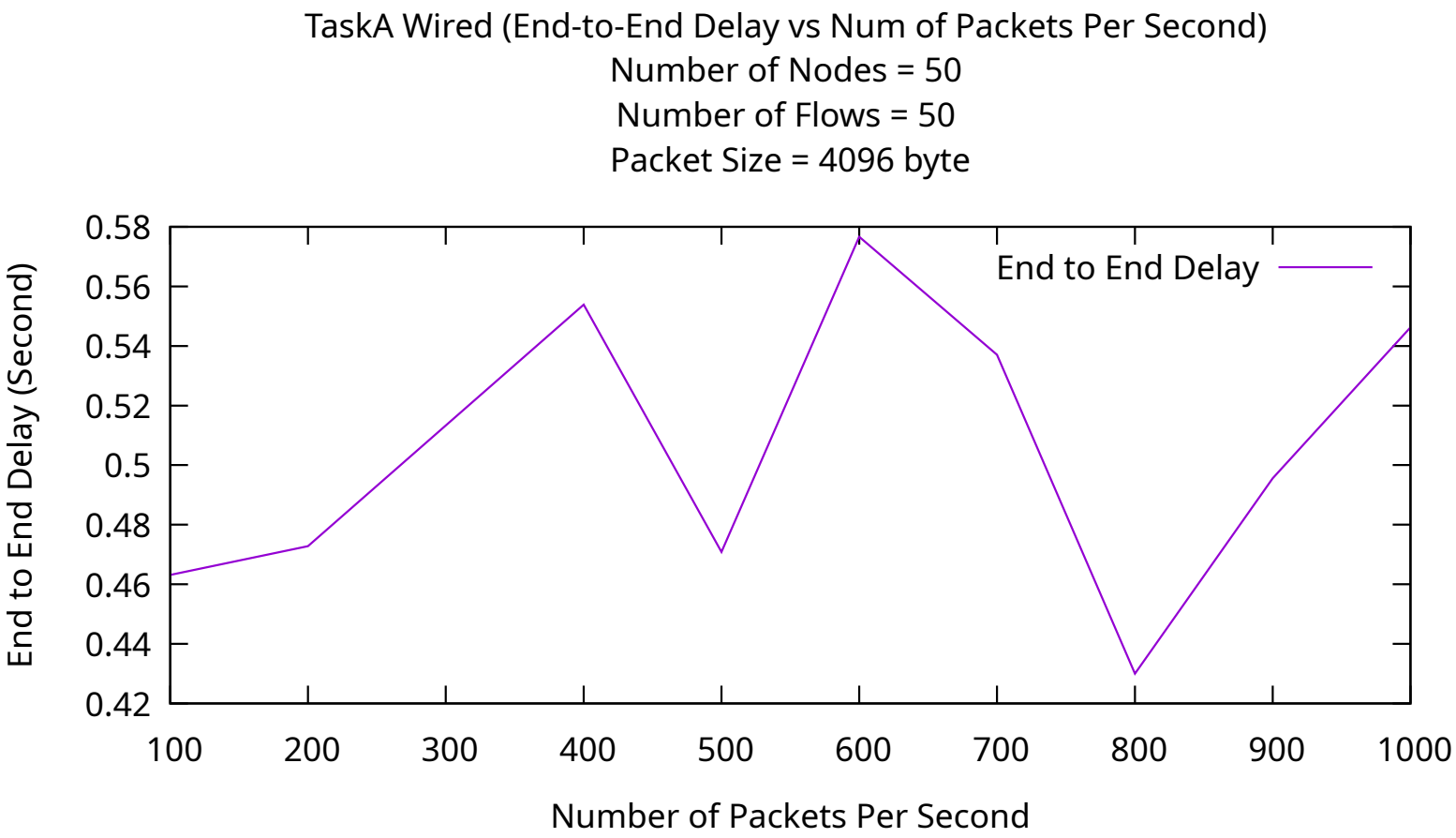

TaskA Wired (Throughput vs Num of Flows)
Number of Nodes = 50
Number of Packets Per Second = 100
Packet Size = 4096 byte
Throughput
Throughput (Kbps)
1150
1100
1050
1000
950
900
850
10
20
30
40
50
60
70
80
90
100
Number of Flows

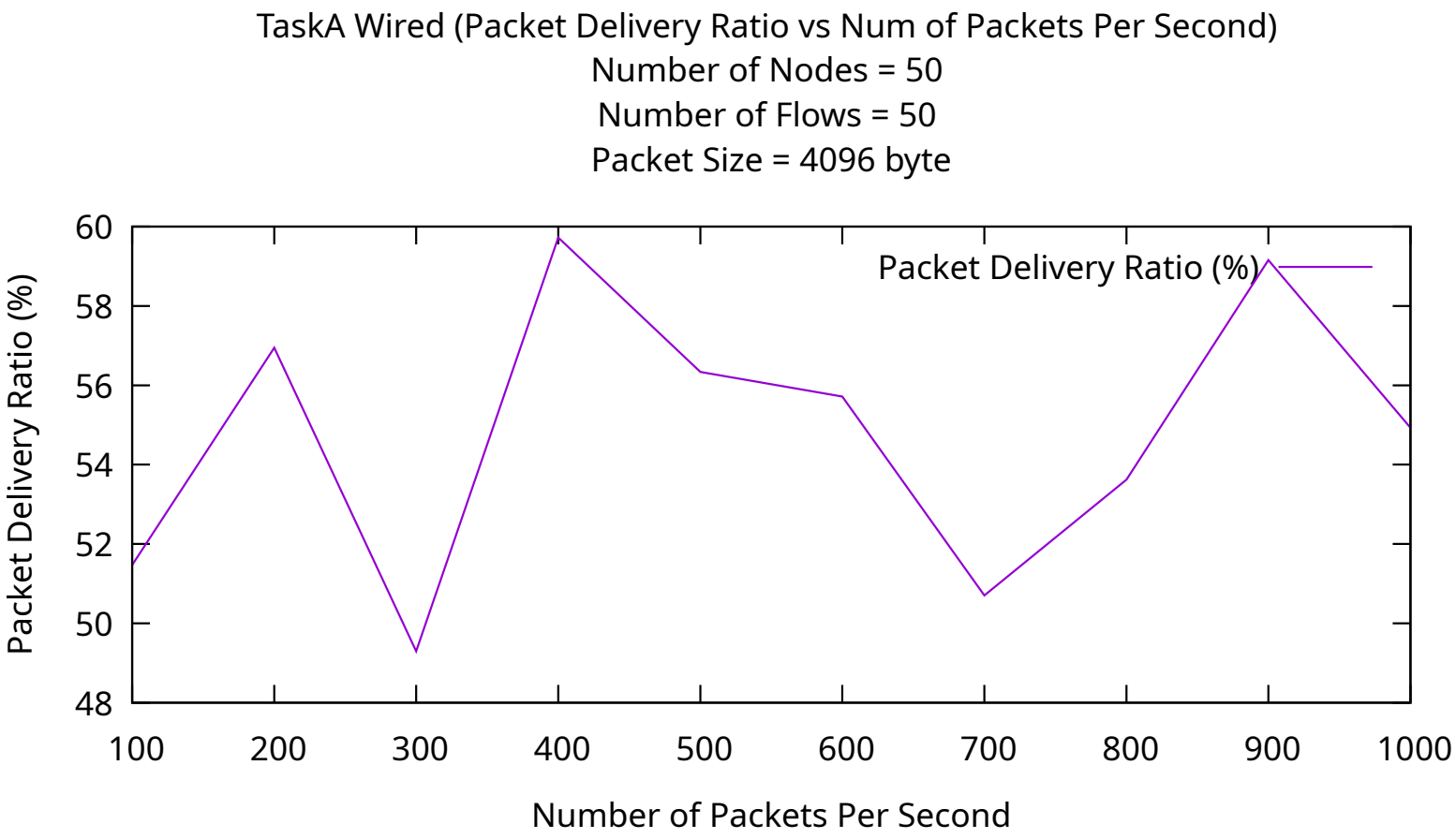

TaskA Wired (End-to-End Delay vs Num of Flows)
Number of Nodes = 50
Number of Packets Per Second = 100
Packet Size = 4096 byte
End to End Delay (Second)
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
10
20
30
40
50
60
70
80
90
100
Number of Flows
End to End Delay

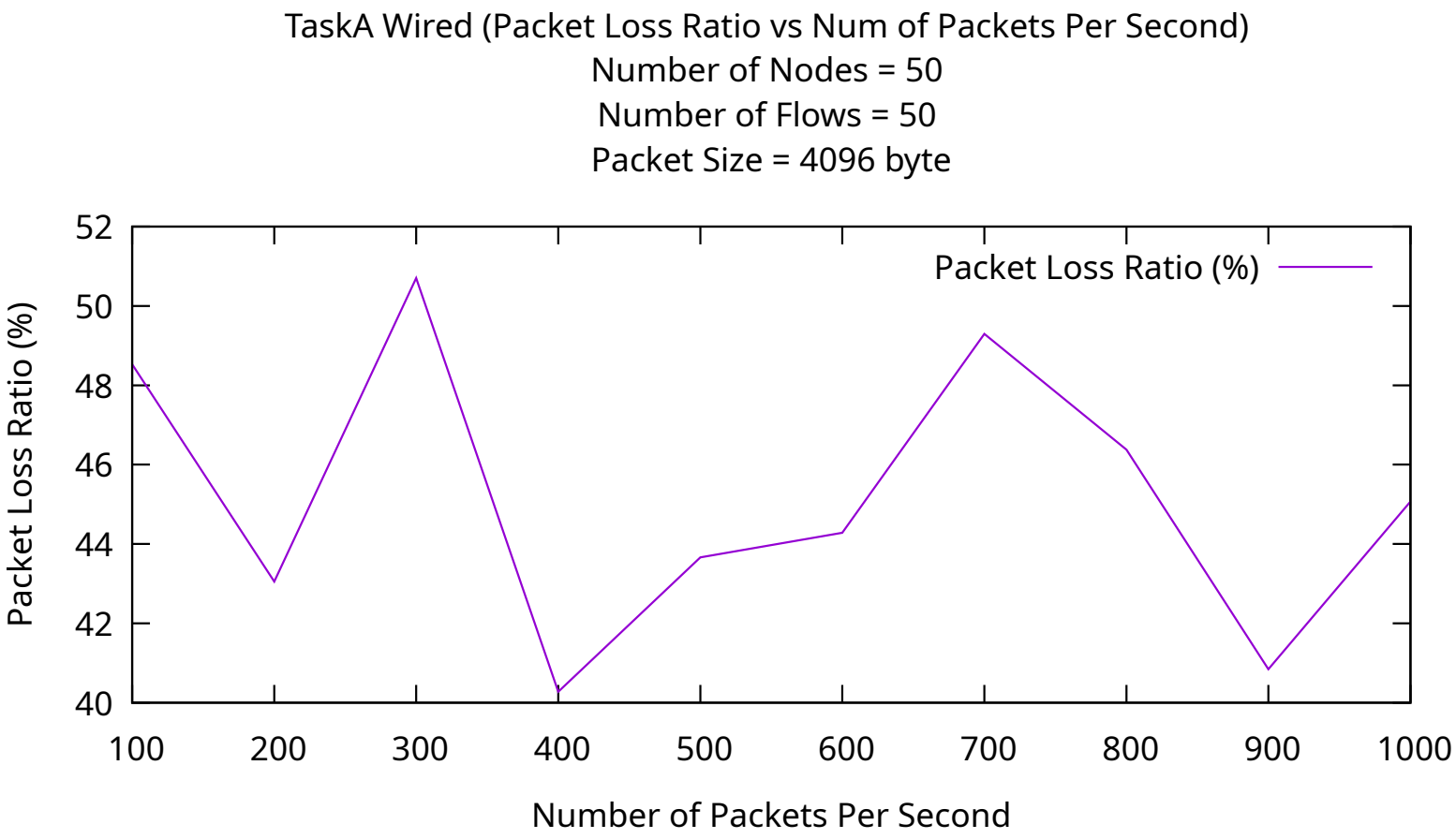

TaskA Wired (Packet Delivery Ratio vs Num of Flows)
Number of Nodes = 50
Number of Packets Per Second = 100
Packet Size = 4096 byte
Packet Delivery Ratio (%)
90
80
70
60
50
40
30
10
20
30
40
50
60
70
80
90
100
Packet Delivery Ratio (%)
Number of Flows


TaskA Wired (Packet Loss Ratio vs Num of Flows)
Number of Nodes = 50
Number of Packets Per Second = 100
Packet Size = 4096 byte
Packet Loss Ratio (%)
70
60
50
40
30
20
10
10
20
30
40
50
60
70
80
90
100
Number of Flows
Packet Loss Ratio (%)


TaskA Wired (Throughput vs Num of Packets Per Second)
Number of Nodes = 50
Number of Flows = 50
Packet Size = 4096 byte
Throughput (Kbps)
1000
950
900
850
800
750
100
200
300
400
500
600
700
800
900
1000
Number of Packets Per Second
Throughput


TaskA Wired (End-to-End Delay vs Num of Packets Per Second)
Number of Nodes = 50
Number of Flows = 50
Packet Size = 4096 byte
End to End Delay (Second)
0.58
0.56
0.54
0.52
0.5
0.48
0.46
0.44
0.42
100
200
300
400
500
600
700
800
900
1000
Number of Packets Per Second
End to End Delay


TaskA Wired (Packet Delivery Ratio vs Num of Packets Per Second)
Number of Nodes = 50
Number of Flows = 50
Packet Size = 4096 byte
Packet Delivery Ratio (%)
60
58
56
54
52
50
48
100
200
300
400
500
600
700
800
900
1000
Number of Packets Per Second


TaskA Wired (Packet Loss Ratio vs Num of Packets Per Second)
Number of Nodes = 50
Number of Flows = 50
Packet Size = 4096 byte
Packet Loss Ratio (%)
52
50
48
46
44
42
40
100
200
300
400
500
600
700
800
900
1000
Number of Packets Per Second

6.2 My Observation about the Result

6.2.1 Variation of Number of Nodes

- Increasing the number of nodes causes an increase in the probability that the packet may be dropped while it competes to access the wireless channel at each node to reach its destination.
- End to end delay increases with the number of nodes as to go to nodes which is far, it costs time.
- Number of nodes increases definitely increases the delivery ratio as number of sink increases. So packet loss ratio decreases.

6.2.2 Variation of Number of Flows

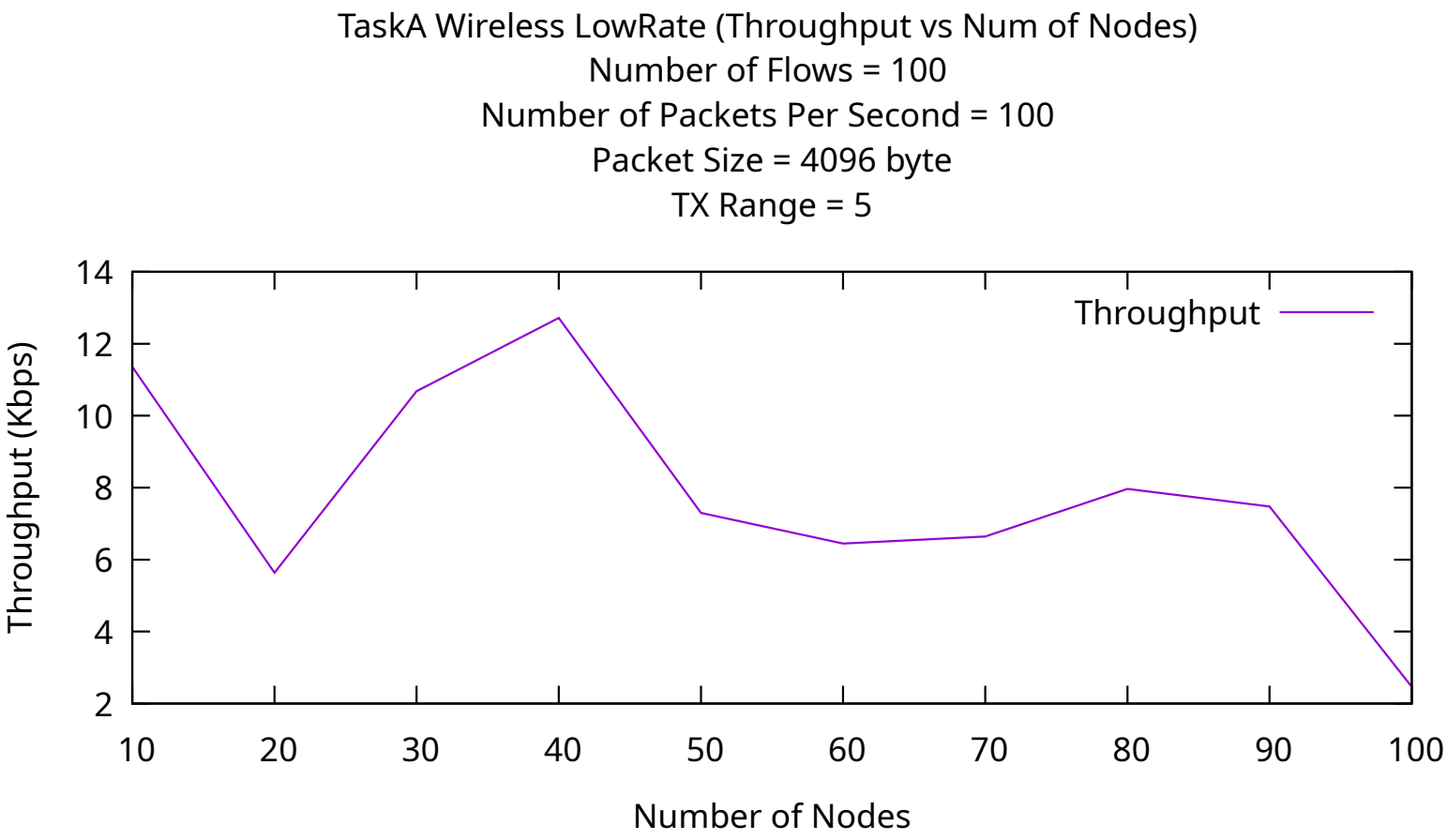
- Increasing number of flows will make race conditions between packets, so throughput may go down. But not always race condition happens.
- Increasing number of flows will make go a flow to far away nodes, it will increase the end to end delay.
- Number of flows to make many packets drop. So packet delivery ratio will decrease and packet drop ratio will increase.

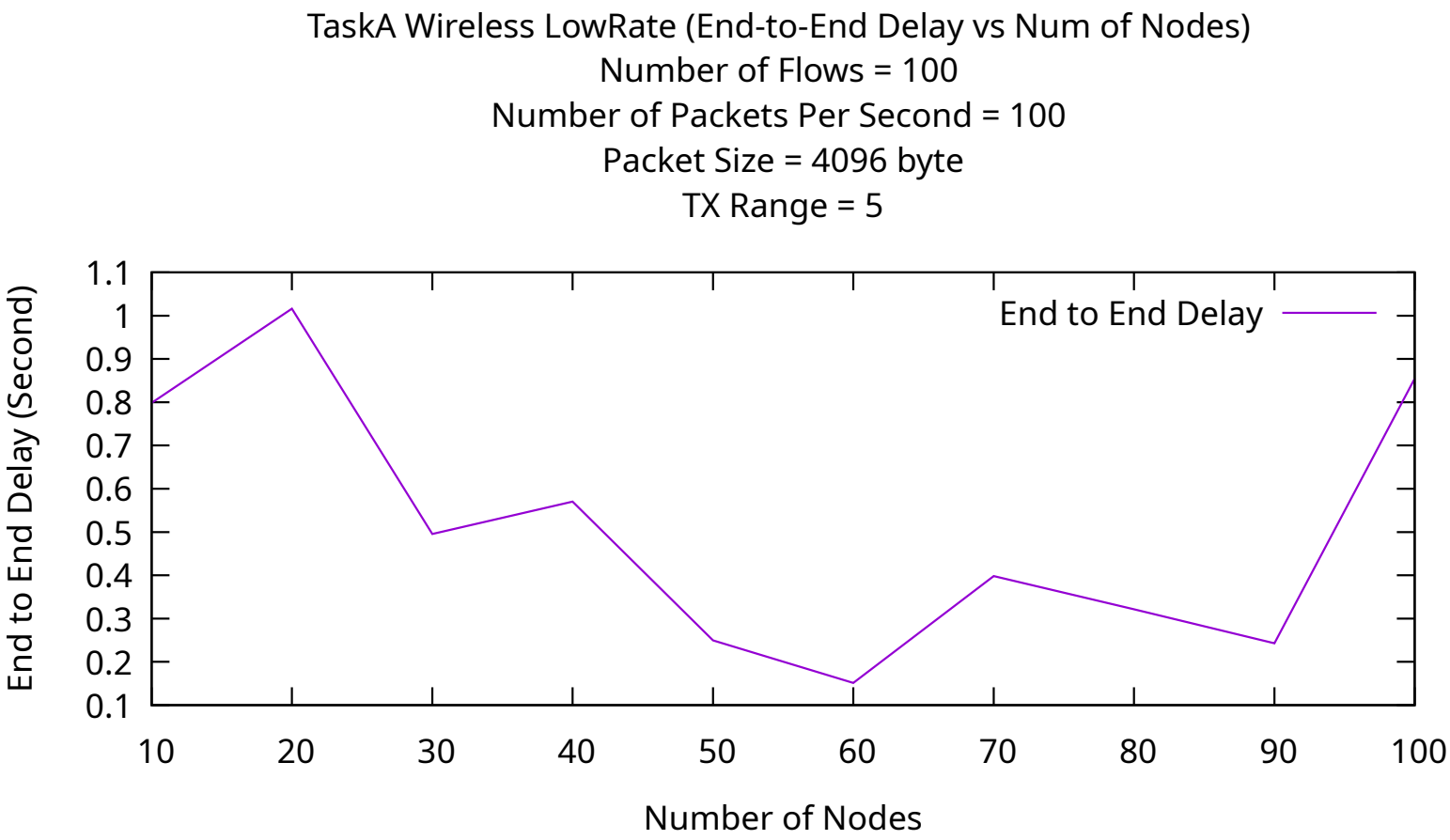
6.2.3 Variation of Number of Packets Per Second

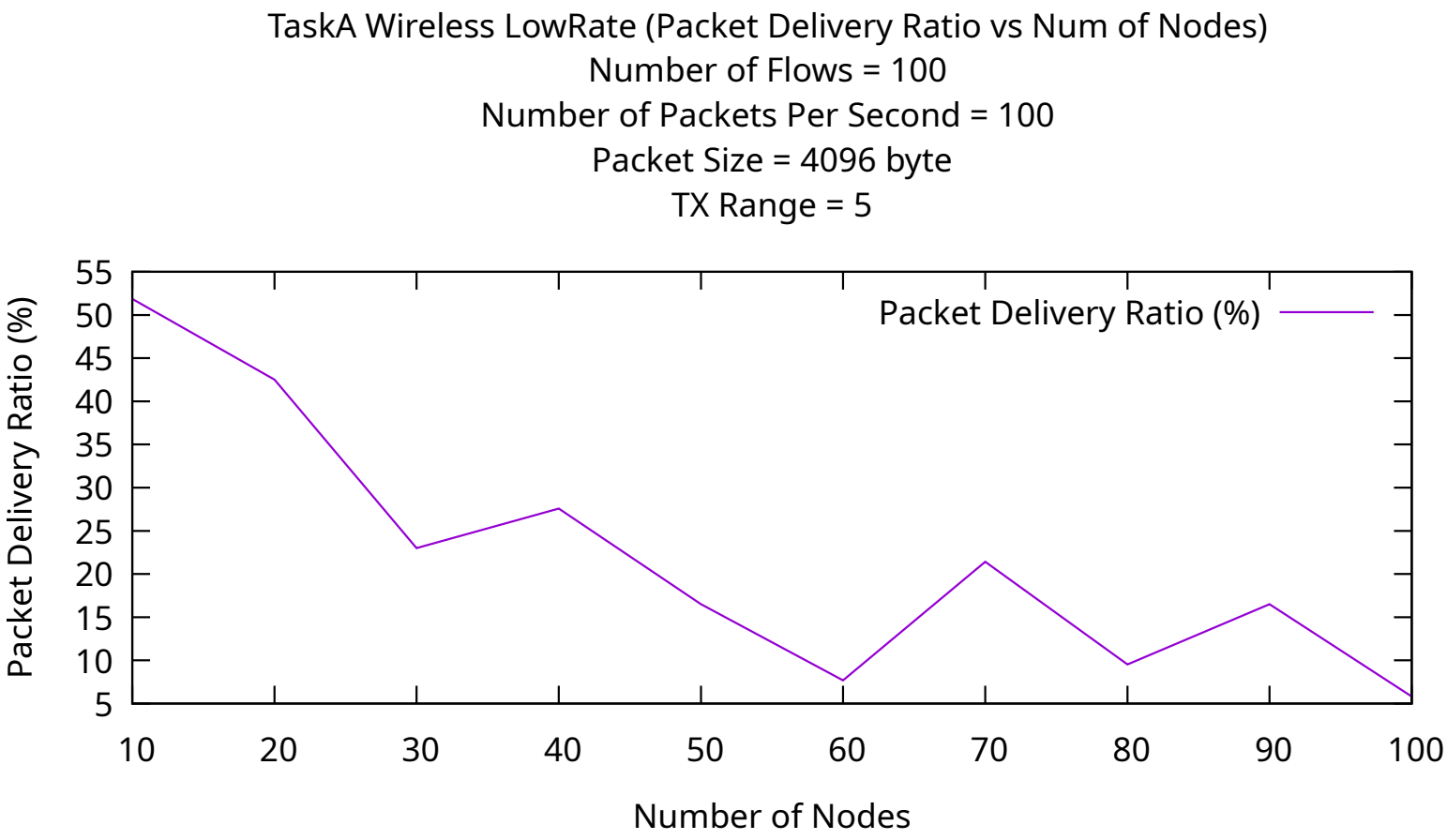
- The increasing in throughput for larger packets is faster than the increasing for smaller ones. It should increase. But flowmonitor didn't generate good plot this time.
- Increasing packets per second should decrease the end to end delay.
- Increasing number of packets per second don't results in good graphs because there are many more parameters which are dependent with packets per second. So packet delivery ratio, packet drop ratio didn't well simulated here.

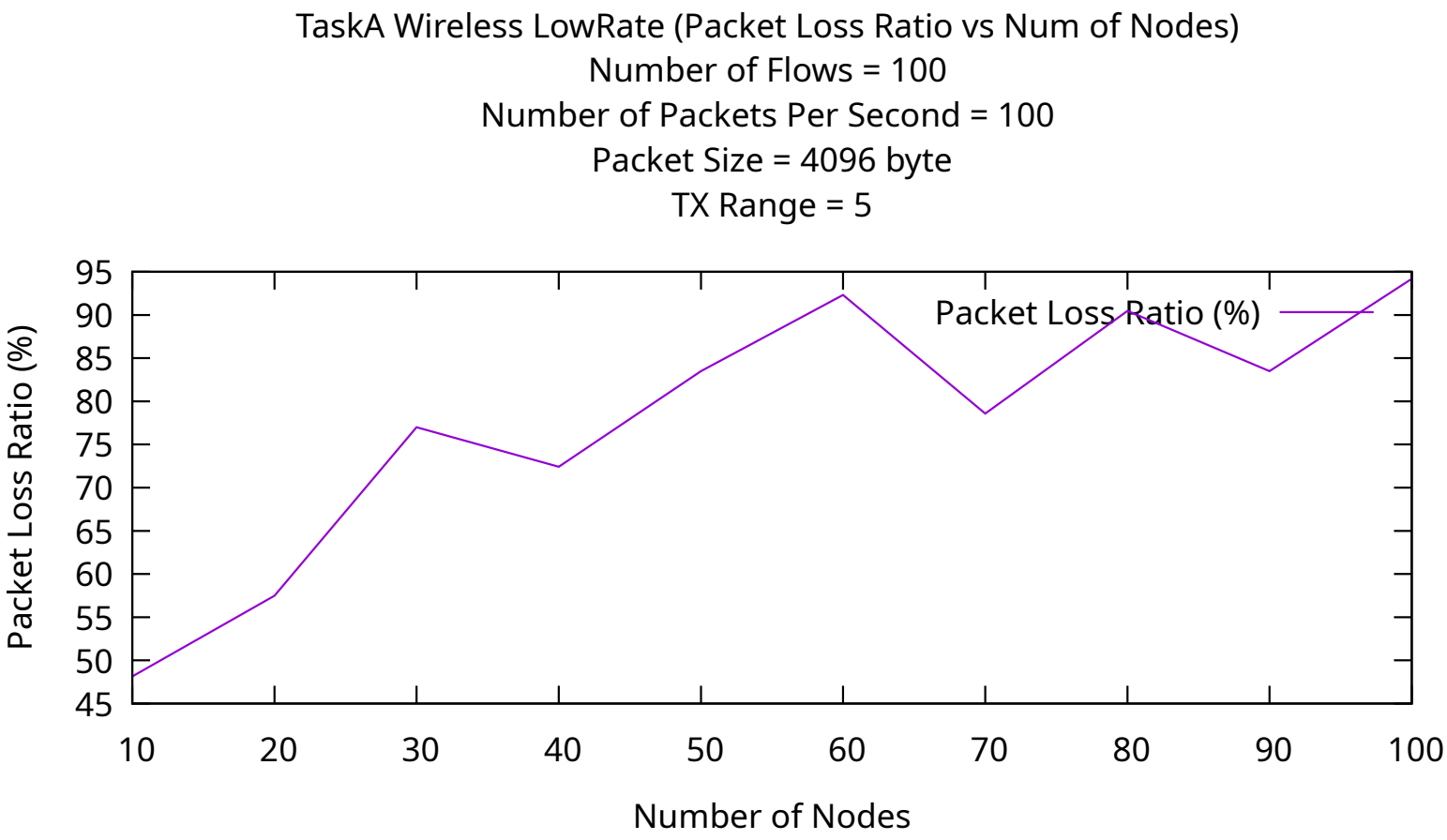
7 Results For Task A - Wireless Low Rate (Static)

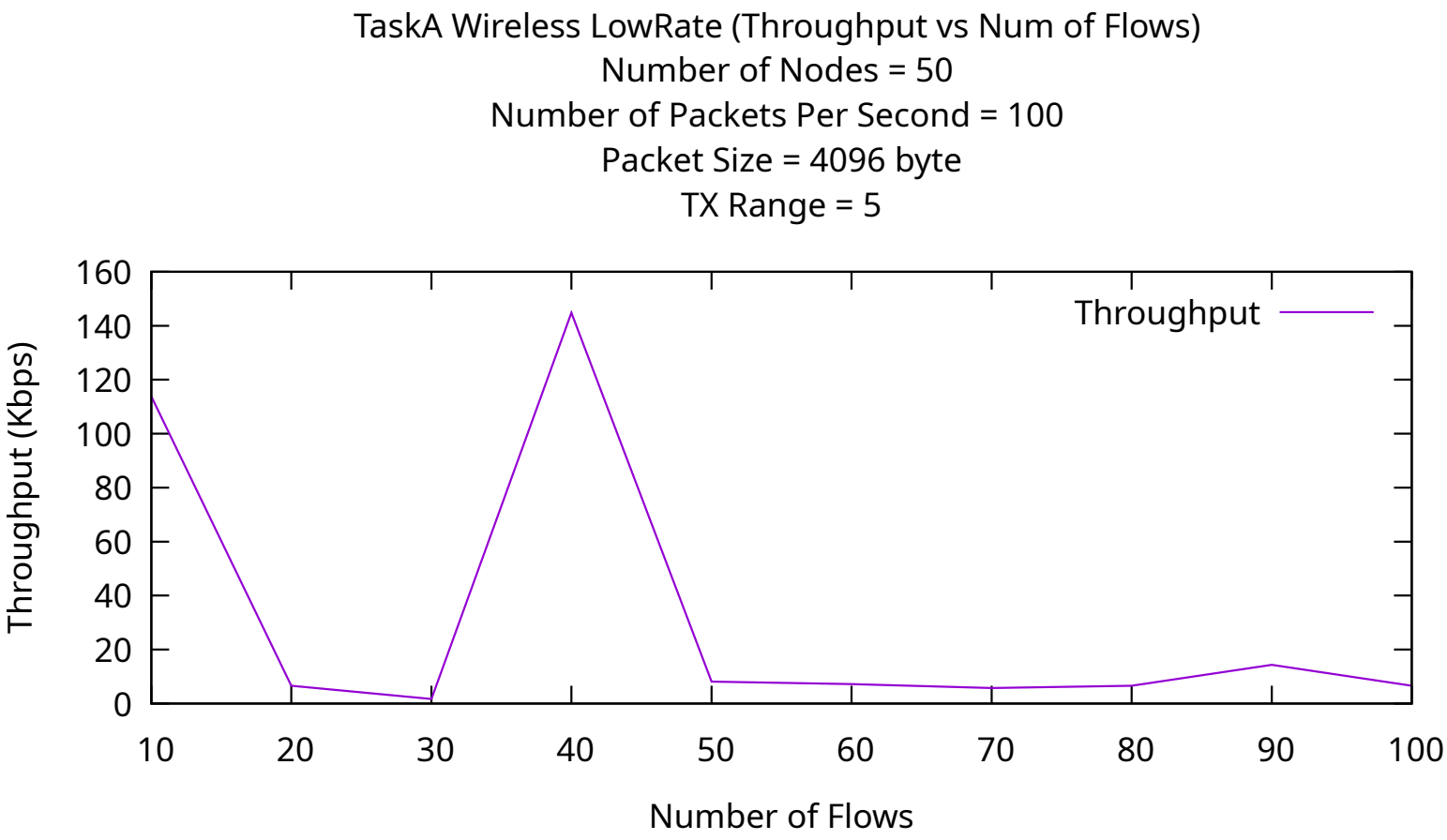
7.1 Plot Graph

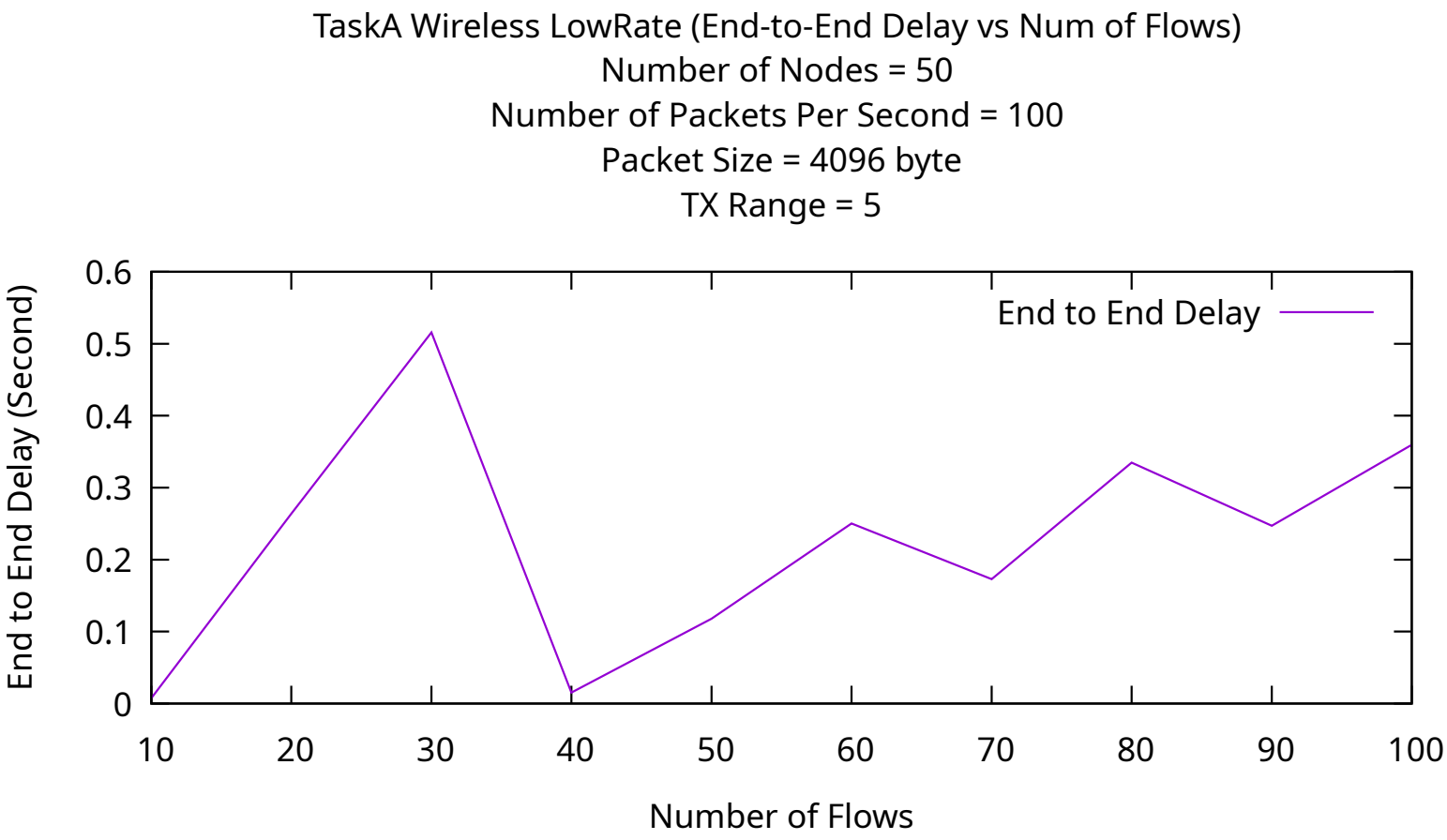

TaskA Wireless LowRate (Throughput vs Num of Nodes)
Number of Flows = 100
Number of Packets Per Second = 100
Packet Size = 4096 byte
TX Range = 5
Throughput
Throughput (Kbps)
Number of Nodes
14
12
10
8
6
4
2
10
20
30
40
50
60
70
80
90
100

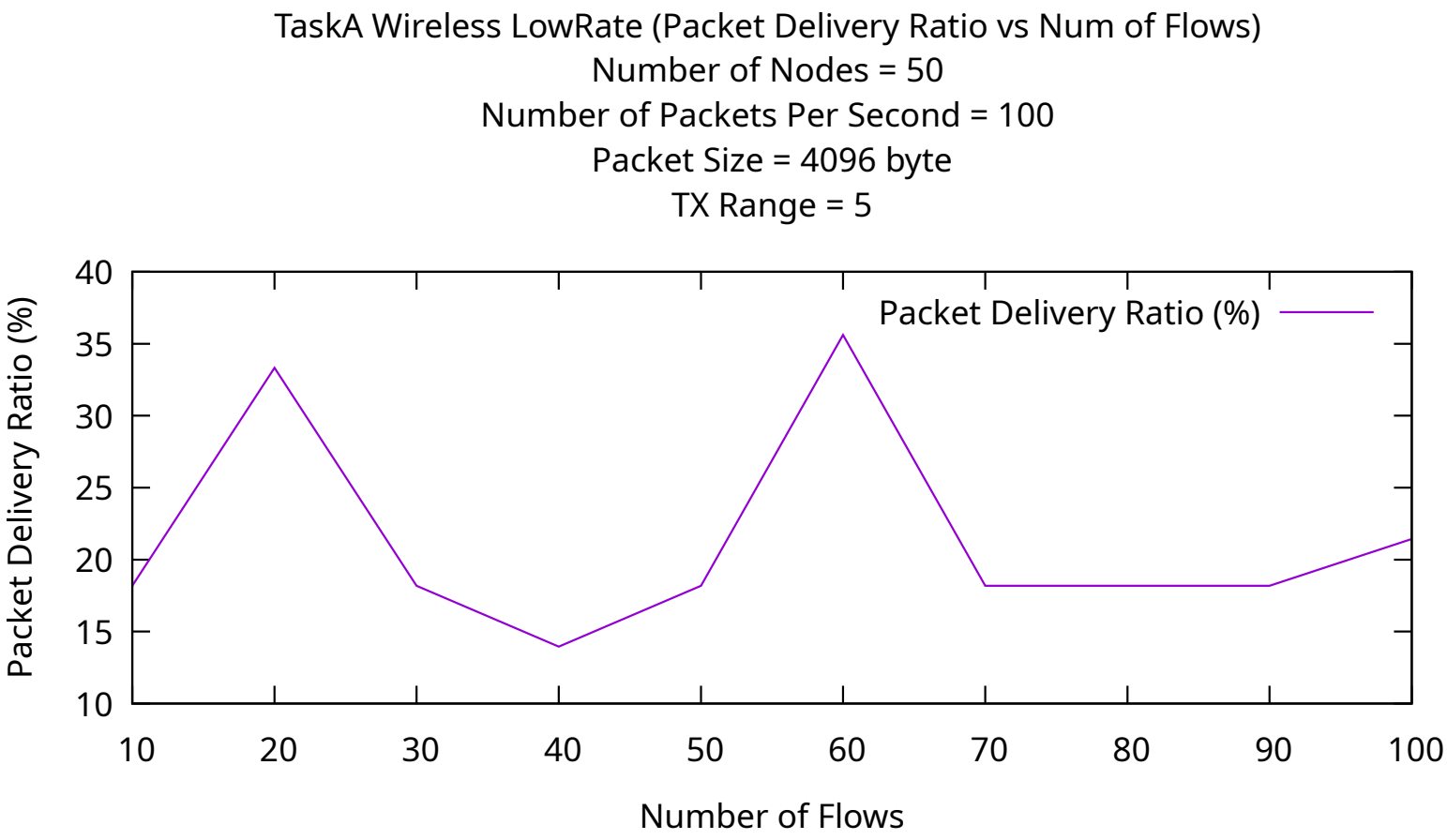

TaskA Wireless LowRate (End-to-End Delay vs Num of Nodes)
Number of Flows = 100
Number of Packets Per Second = 100
Packet Size = 4096 byte
TX Range = 5
End to End Delay (Second)
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
10
20
30
40
50
60
70
80
90
100
Number of Nodes
End to End Delay

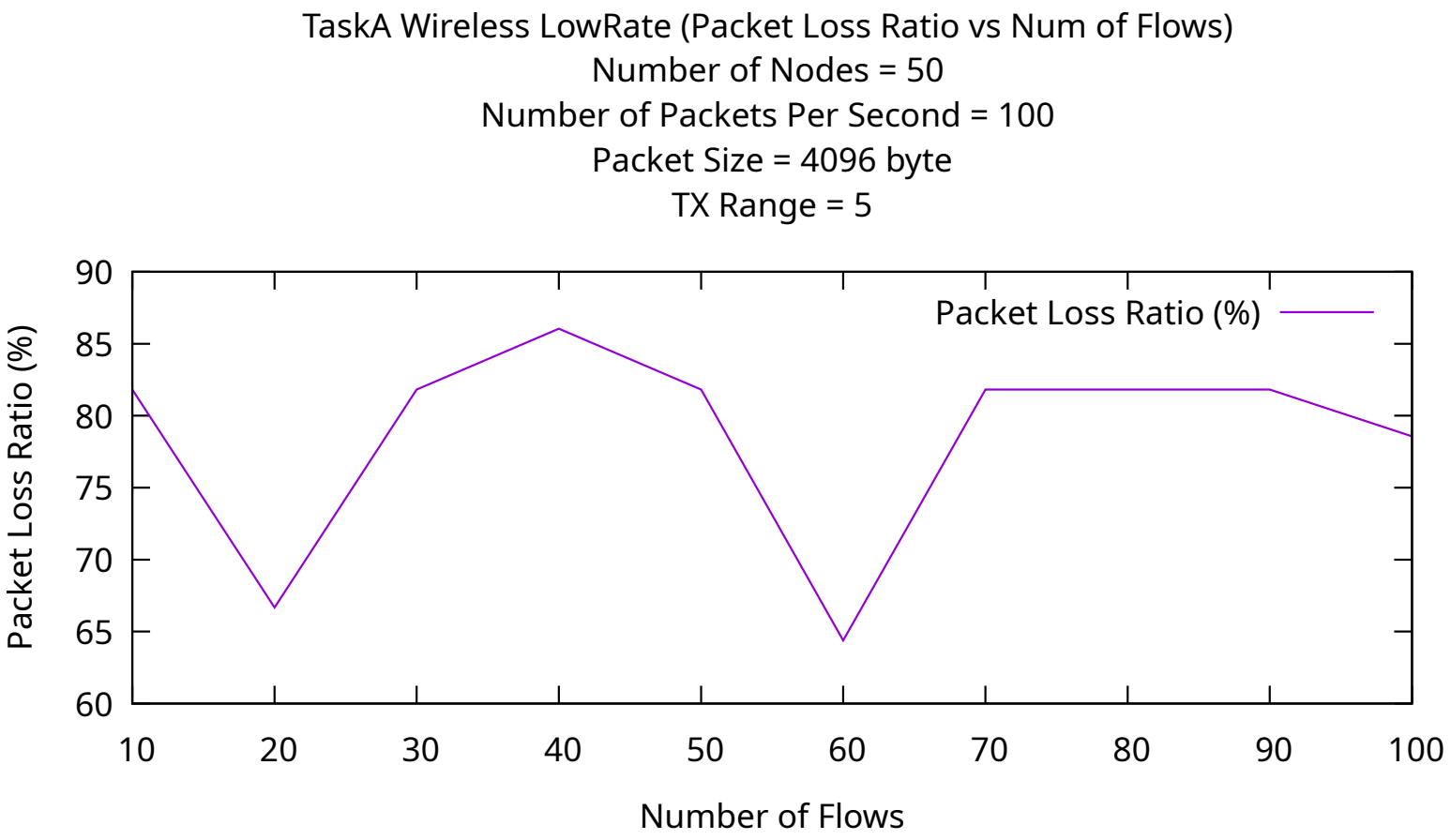

TaskA Wireless LowRate (Packet Delivery Ratio vs Num of Nodes)
Number of Flows = 100
Number of Packets Per Second = 100
Packet Size = 4096 byte
TX Range = 5
Packet Delivery Ratio (%)
55
50
45
40
35
30
25
20
15
10
5
10
20
30
40
50
60
70
80
90
100
Number of Nodes

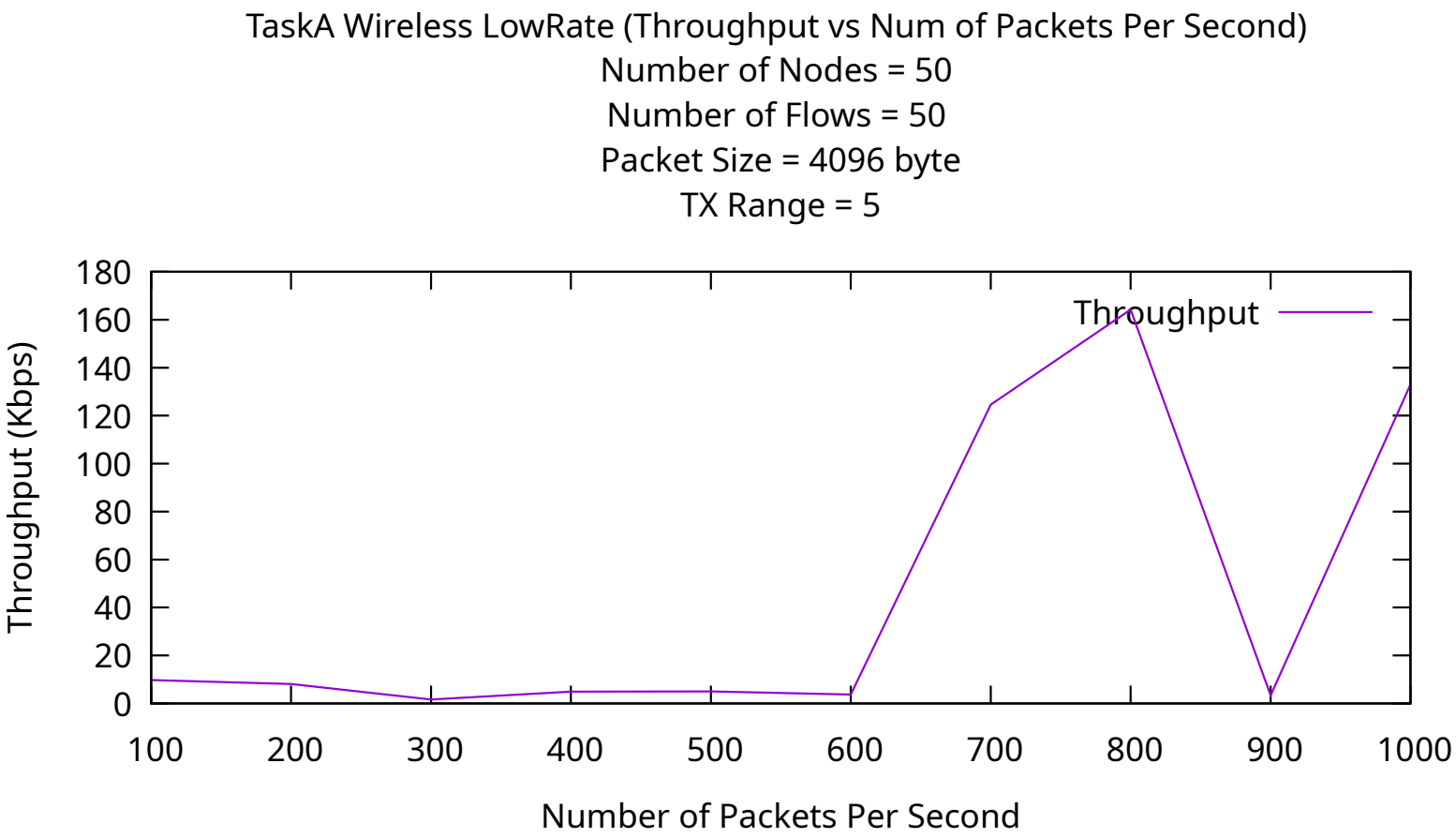

TaskA Wireless LowRate (Packet Loss Ratio vs Num of Nodes)
Number of Flows = 100
Number of Packets Per Second = 100
Packet Size = 4096 byte
TX Range = 5
Packet Loss Ratio (%)
95
90
85
80
75
70
65
60
55
50
45
10
20
30
40
50
60
70
80
90
100
Number of Nodes

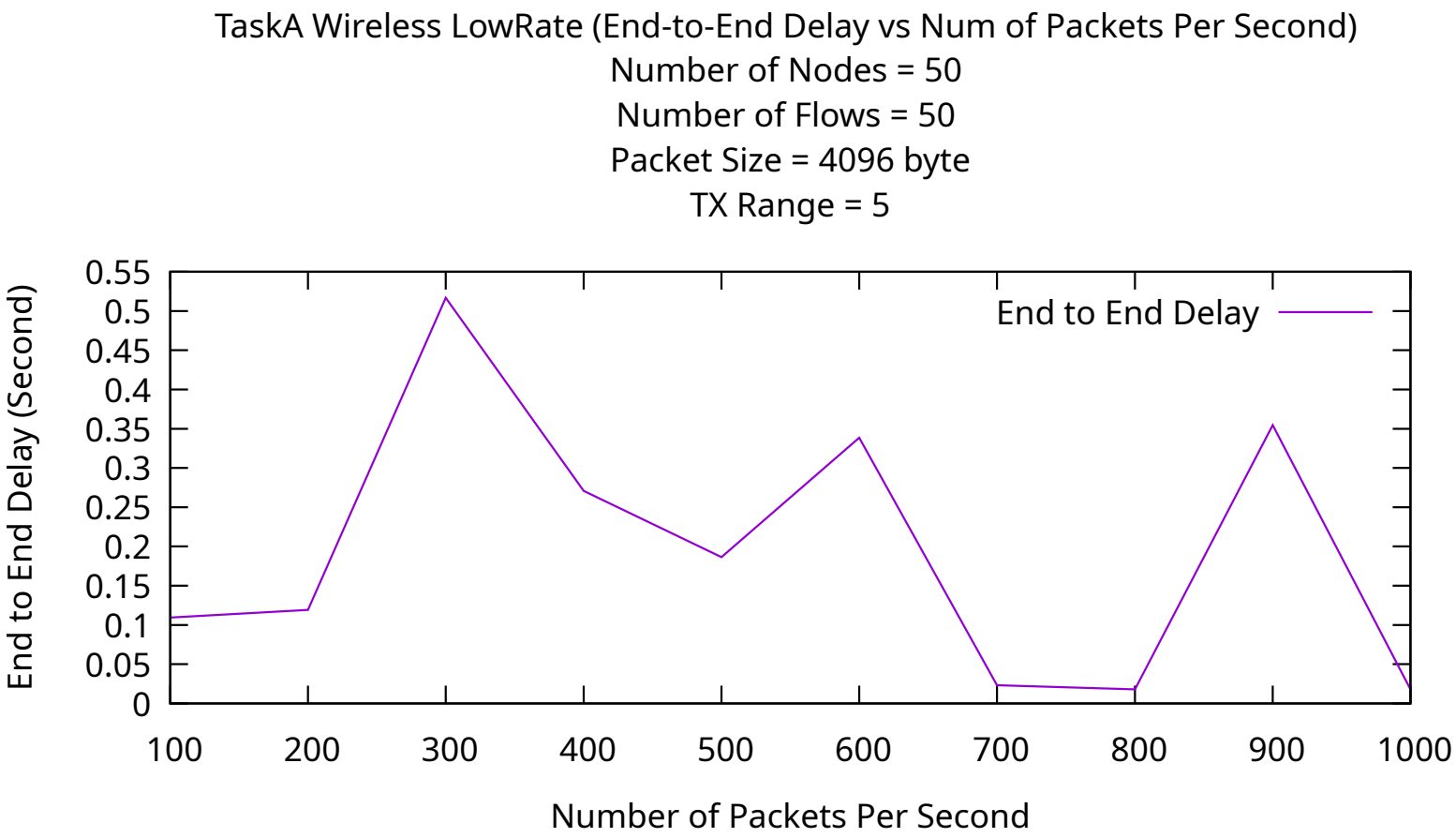

TaskA Wireless LowRate (Throughput vs Num of Flows)
Number of Nodes = 50
Number of Packets Per Second = 100
Packet Size = 4096 byte
TX Range = 5
Throughput (Kbps)
Number of Flows
Throughput
0
20
40
60
80
100
120
140
160
10
20
30
40
50
60
70
80
90
100

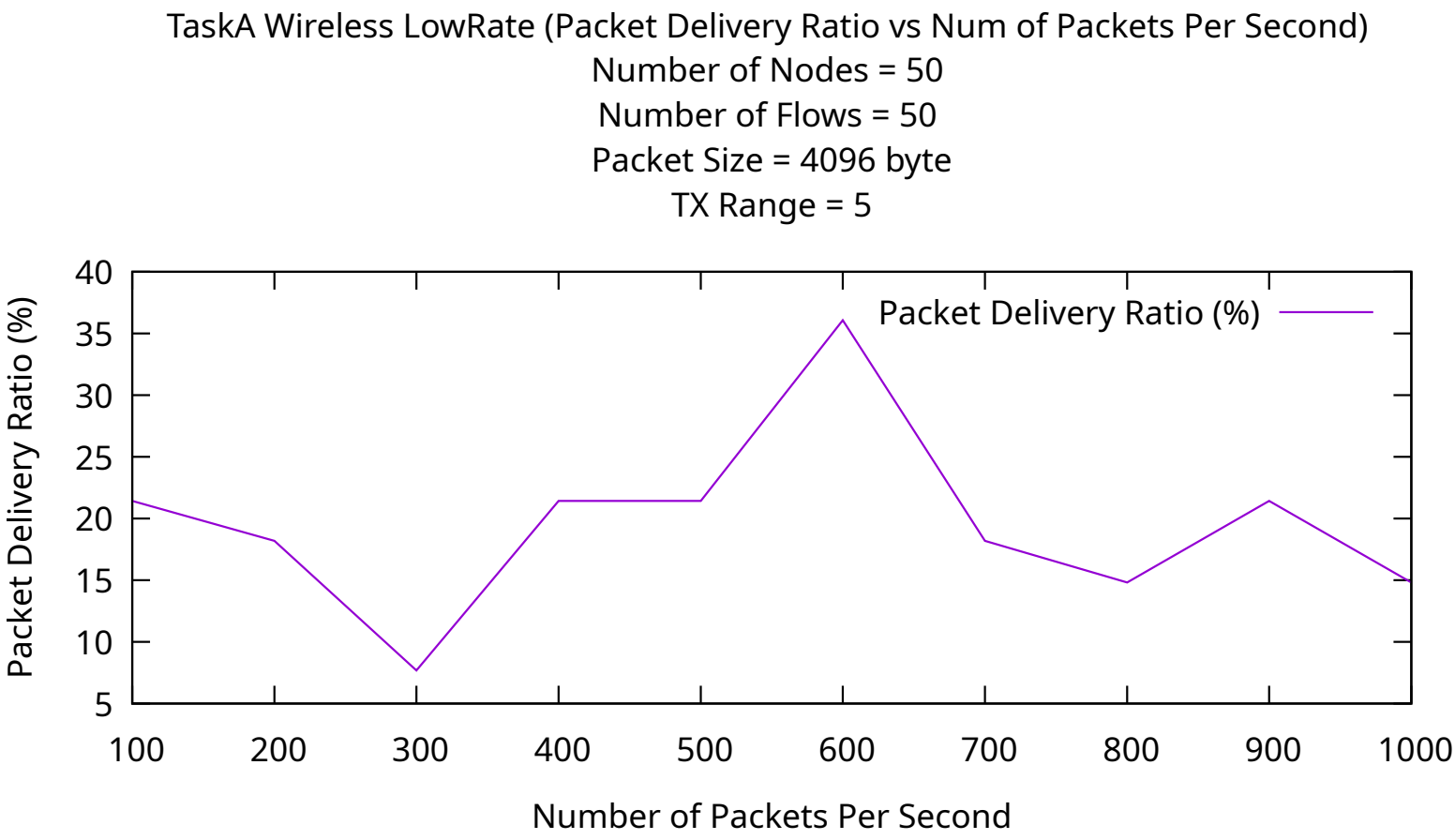

TaskA Wireless LowRate (End-to-End Delay vs Num of Flows)
Number of Nodes = 50
Number of Packets Per Second = 100
Packet Size = 4096 byte
TX Range = 5
End to End Delay (Second)
0.6
0.5
0.4
0.3
0.2
0.1
0
10
20
30
40
50
60
70
80
90
100
Number of Flows
End to End Delay

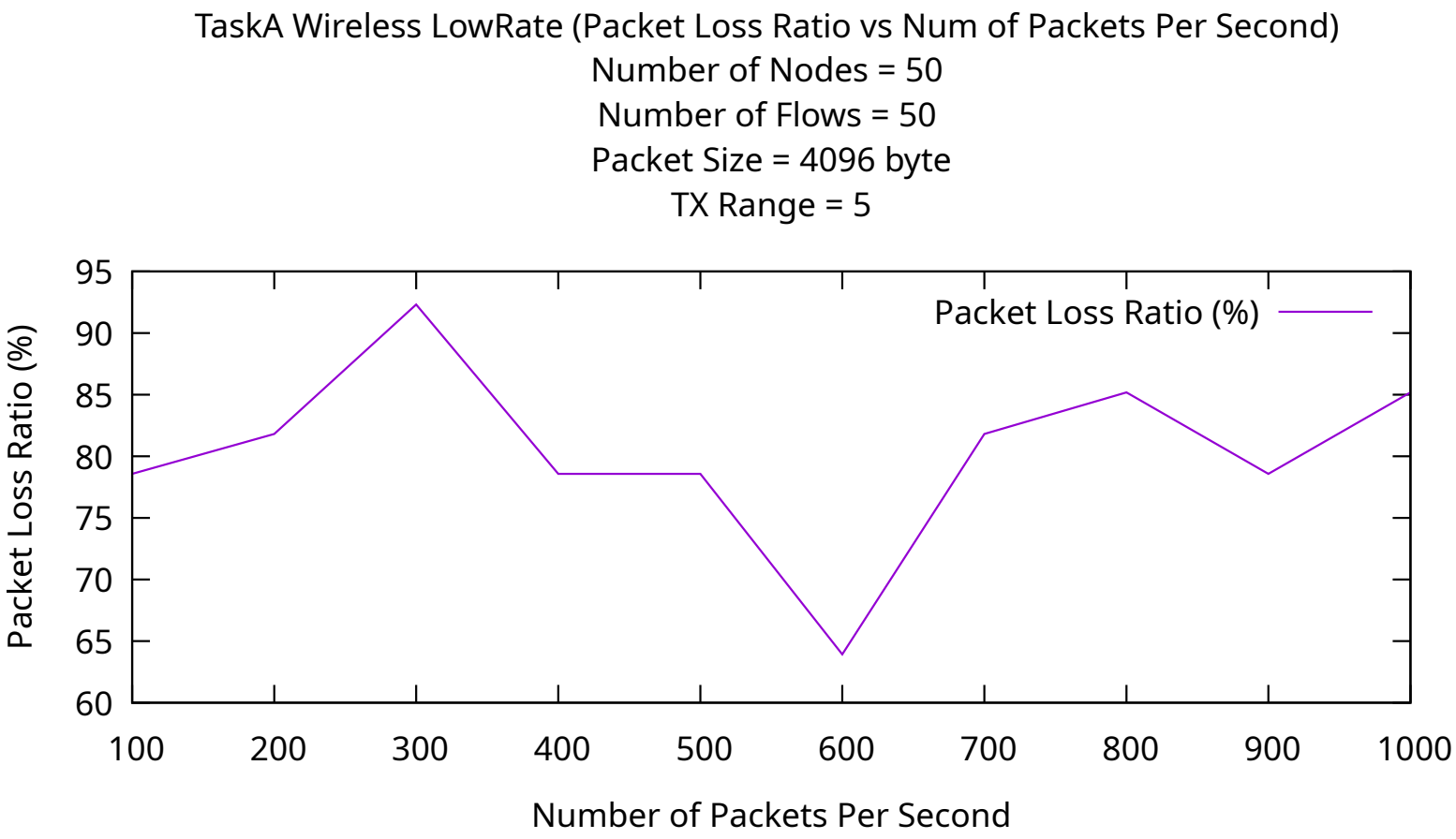

TaskA Wireless LowRate (Packet Delivery Ratio vs Num of Flows)
Number of Nodes = 50
Number of Packets Per Second = 100
Packet Size = 4096 byte
TX Range = 5
Packet Delivery Ratio (%)
40
35
30
25
20
15
10
10
20
30
40
50
60
70
80
90
100
Number of Flows

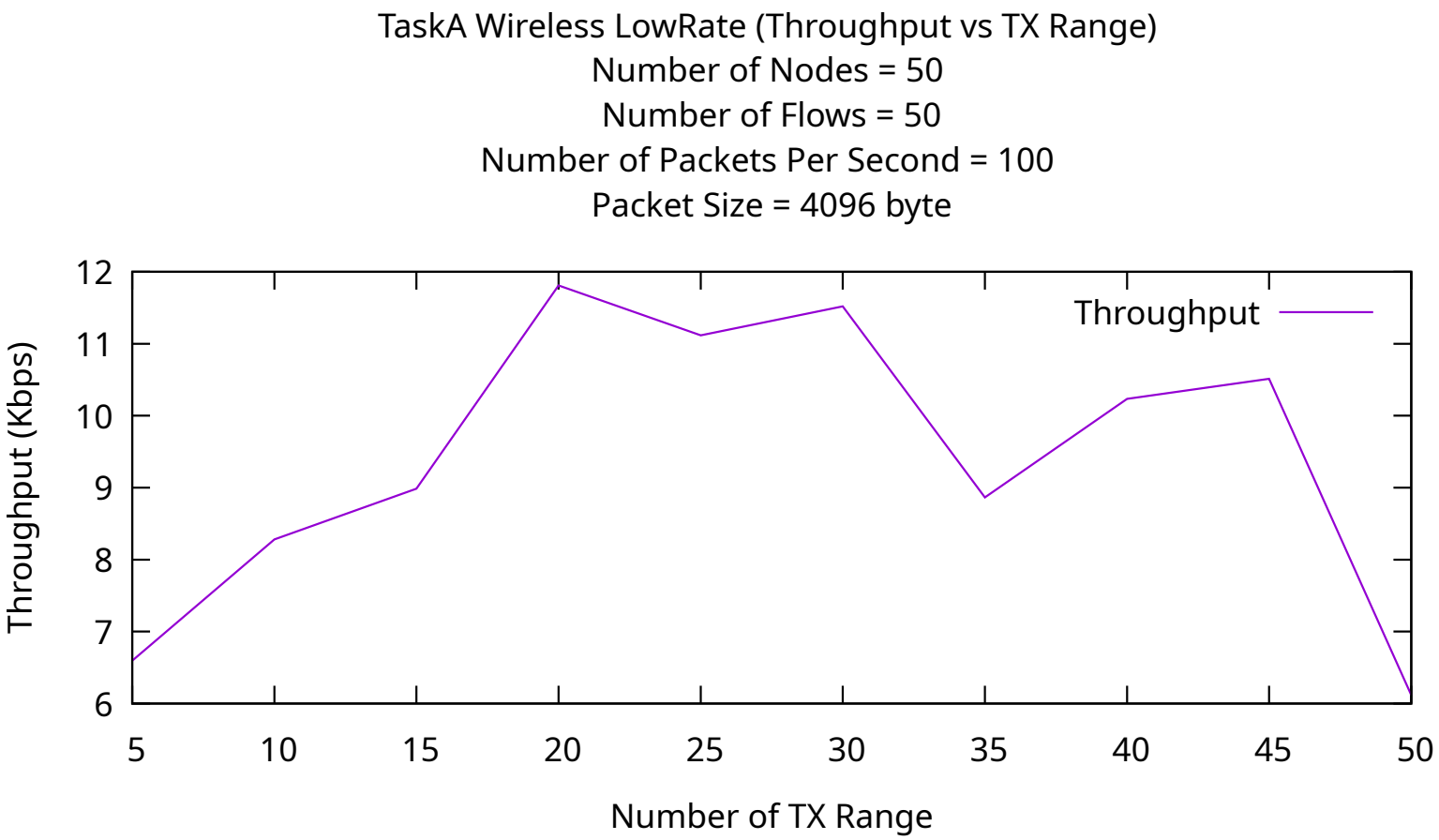

TaskA Wireless LowRate (Packet Loss Ratio vs Num of Flows)
Number of Nodes = 50
Number of Packets Per Second = 100
Packet Size = 4096 byte
TX Range = 5
Packet Loss Ratio (%)
90
85
80
75
70
65
60
10
20
30
40
50
60
70
80
90
100
Number of Flows

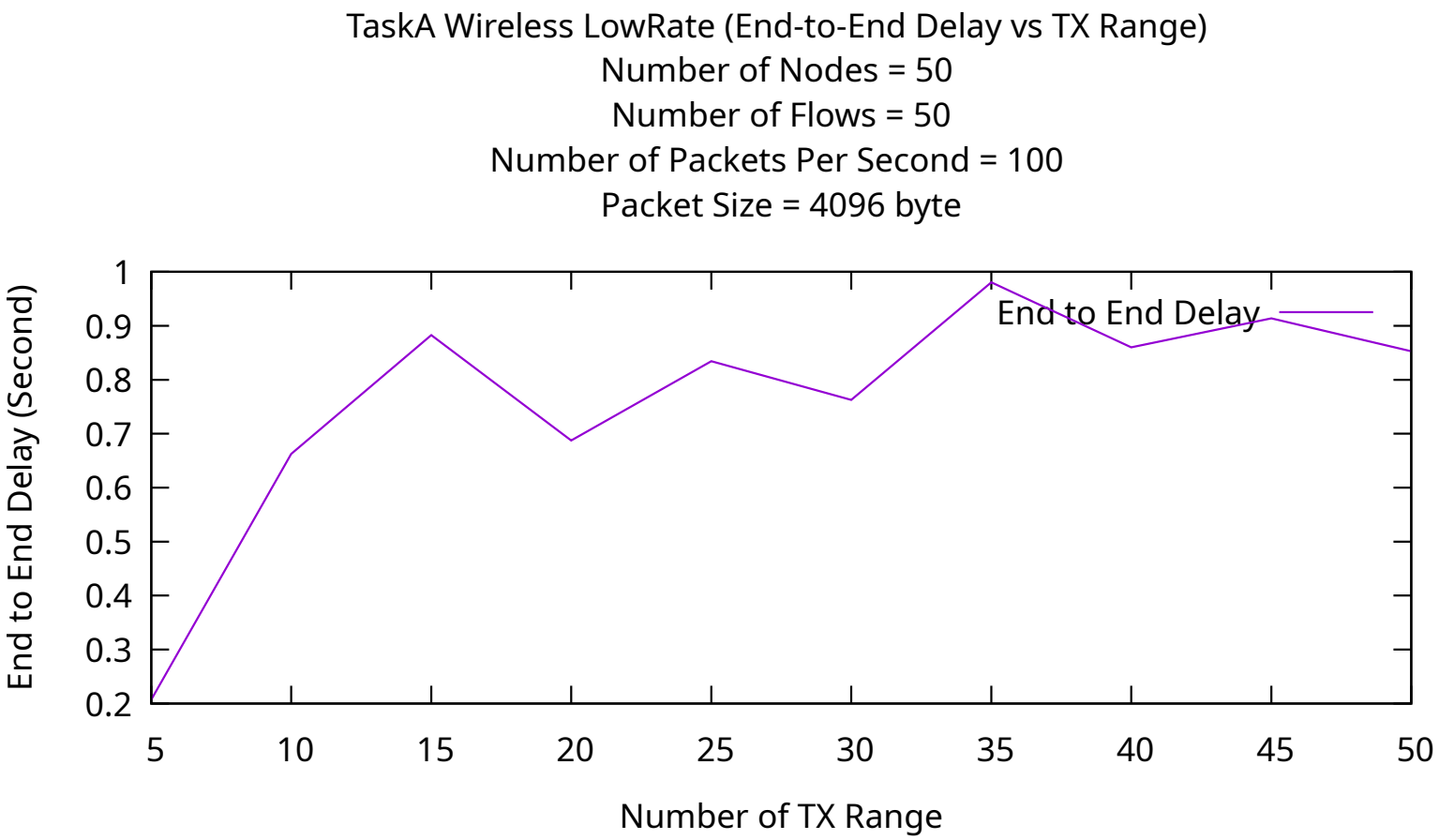

TaskA Wireless LowRate (Throughput vs Num of Packets Per Second)
Number of Nodes = 50
Number of Flows = 50
Packet Size = 4096 byte
TX Range = 5
Throughput (Kbps)
Number of Packets Per Second
Throughput
180
160
140
120
100
80
60
40
20
0
100
200
300
400
500
600
700
800
900
1000

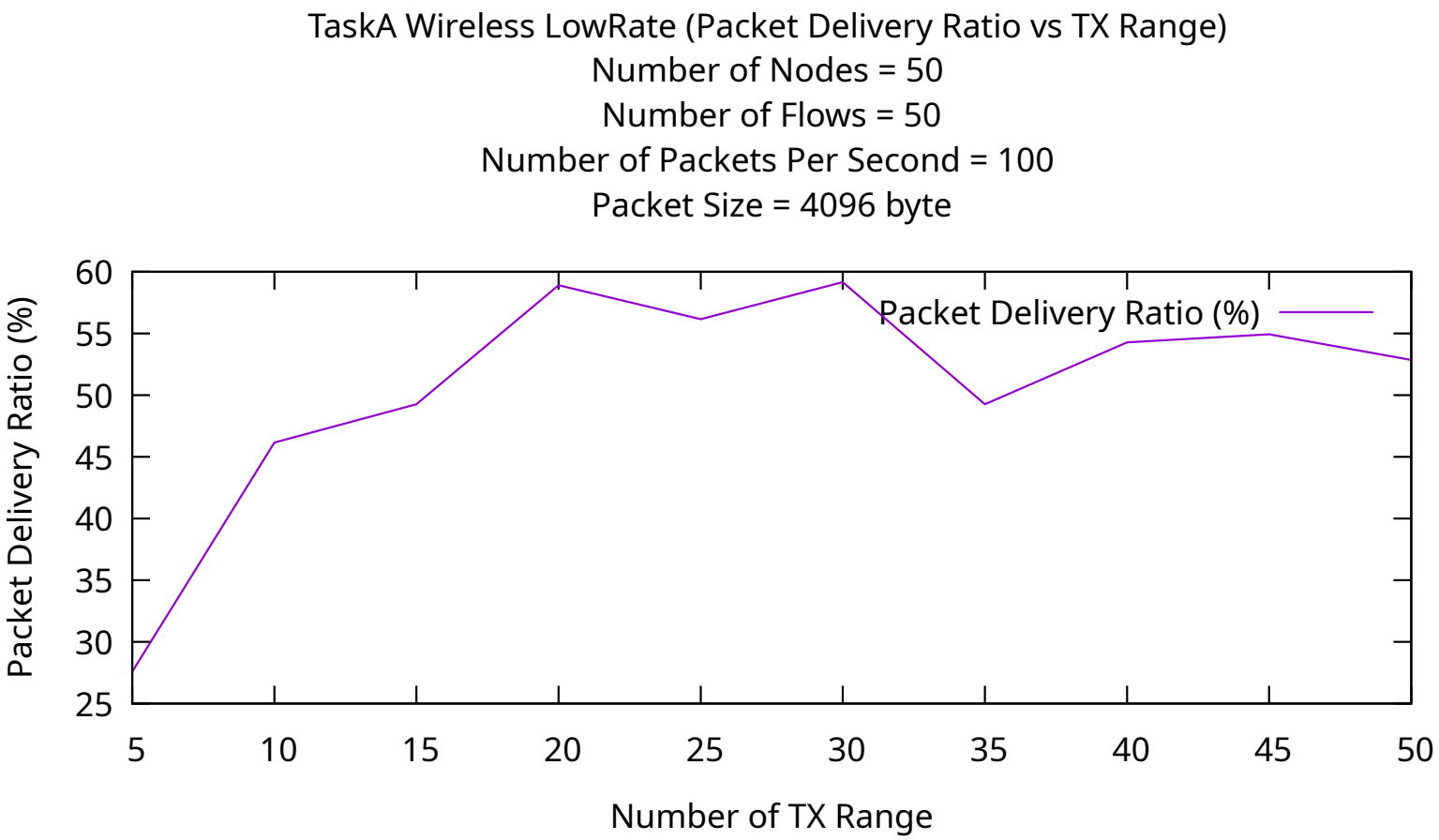

TaskA Wireless LowRate (End-to-End Delay vs Num of Packets Per Second)
Number of Nodes = 50
Number of Flows = 50
Packet Size = 4096 byte
TX Range = 5
End to End Delay (Second)
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
100
200
300
400
500
600
700
800
900
1000
Number of Packets Per Second
End to End Delay

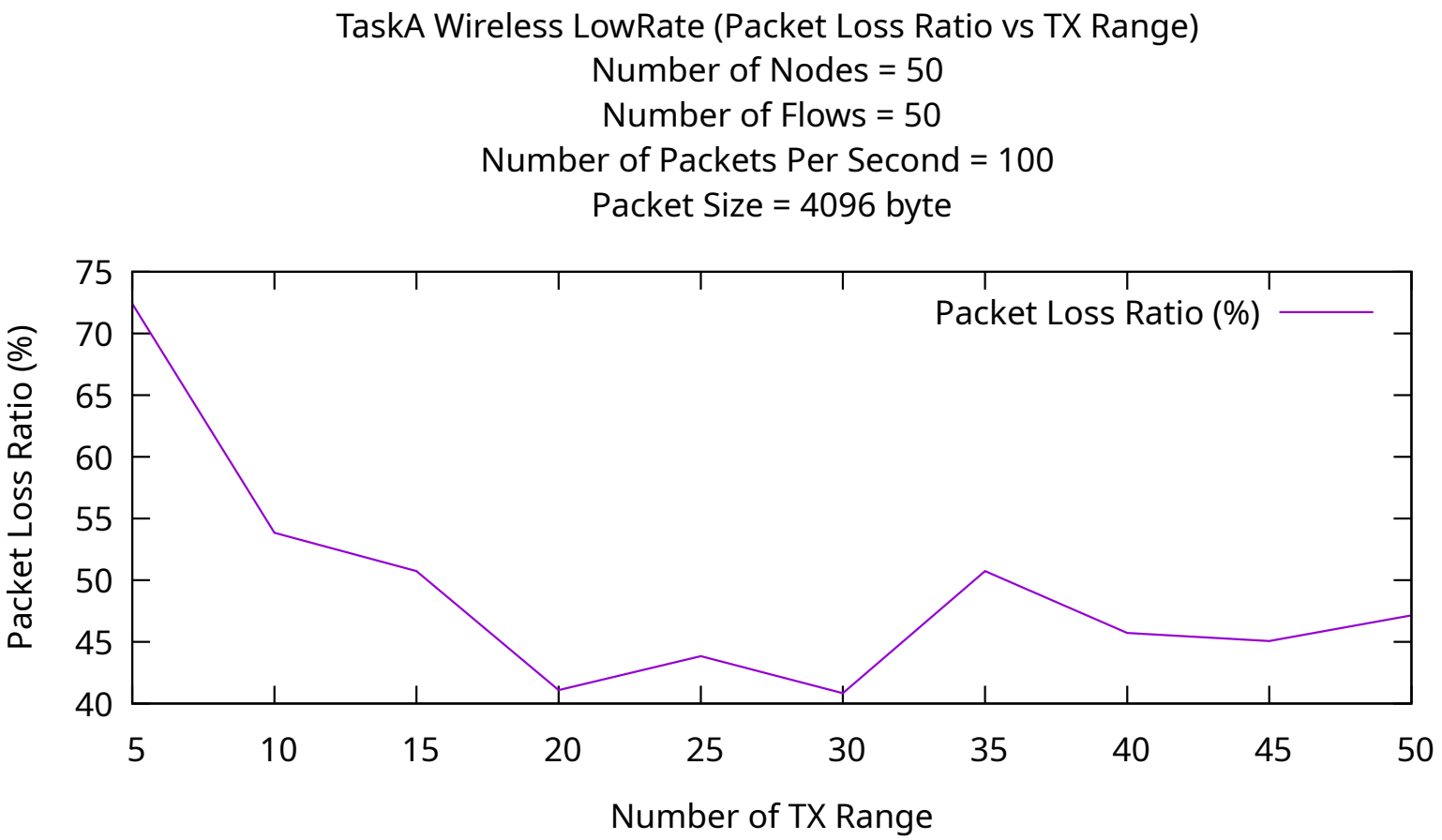

TaskA Wireless LowRate (Packet Delivery Ratio vs Num of Packets Per Second)
Number of Nodes = 50
Number of Flows = 50
Packet Size = 4096 byte
TX Range = 5
Packet Delivery Ratio (%)
40
35
30
25
20
15
10
5
100
200
300
400
500
600
700
800
900
1000
Number of Packets Per Second


TaskA Wireless LowRate (Packet Loss Ratio vs Num of Packets Per Second)
Number of Nodes = 50
Number of Flows = 50
Packet Size = 4096 byte
TX Range = 5
Packet Loss Ratio (%)
95
90
85
80
75
70
65
60
100
200
300
400
500
600
700
800
900
1000
Number of Packets Per Second


TaskA Wireless LowRate (Throughput vs TX Range)
Number of Nodes = 50
Number of Flows = 50
Number of Packets Per Second = 100
Packet Size = 4096 byte
Throughput
Throughput (Kbps)
Number of TX Range
12
11
10
9
8
7
6
5
10
15
20
25
30
35
40
45
50


TaskA Wireless LowRate (End-to-End Delay vs TX Range)
Number of Nodes = 50
Number of Flows = 50
Number of Packets Per Second = 100
Packet Size = 4096 byte
End to End Delay (Second)
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
5
10
15
20
25
30
35
40
45
50
Number of TX Range
End to End Delay


TaskA Wireless LowRate (Packet Delivery Ratio vs TX Range)
Number of Nodes = 50
Number of Flows = 50
Number of Packets Per Second = 100
Packet Size = 4096 byte
Packet Delivery Ratio (%)
60
55
50
45
40
35
30
25
5
10
15
20
25
30
35
40
45
50
Number of TX Range


TaskA Wireless LowRate (Packet Loss Ratio vs TX Range)
Number of Nodes = 50
Number of Flows = 50
Number of Packets Per Second = 100
Packet Size = 4096 byte
Packet Loss Ratio (%)
75
70
65
60
55
50
45
40
5
10
15
20
25
30
35
40
45
50
Number of TX Range
Packet Loss Ratio (%)

7.2 My Observation about the Result

7.2.1 Variation of Number of Nodes

- Increasing the number of nodes causes an increase in the probability that the packet may be dropped while it competes to access the wireless channel at each node to reach its destination.
- End to end delay should be less with the increasing number of nodes. But this is here low rate, so may not behave well due to low rate. Delay may be increased.
- If it were wired, packet delivery ratio would be high with the number of nodes. But here in low rate, packet may not be delivered due to low rate. Packet loss ratio is vice versa.

7.2.2 Variation of Number of Flows

- Increasing number of flows make many packets to drop, so throughput goes down.
- End to end delay increases with the number of flows.
- Packet delivery ratio should increase with the number of flows. But due to low rate, it may misbehave.

7.2.3 Variation of Number of Packets Per Second

- Throughput increases with the number of packets per second.
- End to end delay decreases with the number of packets per second.
- Packet delivery ratio increases with the number of packets per second. Loss ratio is vice versa.

7.2.4 Variation of Tx Range

- Increasing TX range makes a packet to travel more distance. So throughput increases, packet delivery ratio increases, packet drop ratio decreases.
- End to end delay increases as the packet can travel more distance.

8 Conclusion

This project introduced us with NS-3. It is one of the leading network simulating tool. It was difficult to work with for the first time, but with the co-operations of peers and supervisors, NS3 experience has gone well. Hope to work on computer networks in future.
Source codes of this project is available in my Github Repository. `https://github.com/kawshikbuet17/Congestion-Control-AODV`

References

[1] CC-ADOV: An effective multiple paths congestion control AODV
Yefa Mai, Fernando Molina Rodriguez, Nan Wang, https://ieeexplore.ieee.org/document/8301758