

---

*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

<b>1</b>	<b>Network Topologies Under Simulation</b>	<b>4</b>
1.1	Topology for Task A - Wired . . . . .	4
1.2	Topology for Task A - Wireless Low Rate (Static) . . . . .	4
1.3	Topology for Task B - MANET AODV . . . . .	4
<b>2</b>	<b>Parameters Under Variation</b>	<b>5</b>
2.1	Parameters for Task A - Wired . . . . .	5
2.2	Parameters for Task A - Wireless Low Rate (Static) . . . . .	5
2.3	Parameters for Task B - MANET AODV . . . . .	5
<b>3</b>	<b>Overview of the Proposed Algorithm</b>	<b>6</b>
3.1	What is AODV? . . . . .	6
3.1.1	Route Discovery . . . . .	6
3.1.2	Route Maintenance . . . . .	7
3.2	Issues with AODV . . . . .	7
3.3	Proposed CC-AODV Mechanism . . . . .	7
3.3.1	Flowchart . . . . .	8
3.3.2	Algorithm . . . . .	8
3.3.3	Implementation Guideline . . . . .	9
<b>4</b>	<b>Modifications Made in the Simulator</b>	<b>9</b>
4.1	Add Congestion Flag in RREP Packet Header . . . . .	10
4.2	Congestion Flag in RoutingTableEntry . . . . .	12
4.3	Add Congestion Counter and Max Count . . . . .	12
4.4	Drop RREQ Counter Greater than Max Count . . . . .	13
4.5	Counter Increment or Decrement and Flag Operations . . . . .	14
<b>5</b>	<b>Paper Result vs Result of My Task B</b>	<b>15</b>
5.1	Plot Graph . . . . .	15
5.2	My Observation about the Result . . . . .	16
5.3	Minor Improvement Suggestion from my side . . . . .	17
<b>6</b>	<b>Results For Task A - Wired Topology</b>	<b>17</b>
6.1	Plot Graph . . . . .	17
6.2	My Observation about the Result . . . . .	30
6.2.1	Variation of Number of Nodes . . . . .	30
6.2.2	Variation of Number of Flows . . . . .	30
6.2.3	Variation of Number of Packets Per Second . . . . .	30
<b>7</b>	<b>Results For Task A - Wireless Low Rate (Static)</b>	<b>30</b>
7.1	Plot Graph . . . . .	30
7.2	My Observation about the Result . . . . .	47
7.2.1	Variation of Number of Nodes . . . . .	47
7.2.2	Variation of Number of Flows . . . . .	47
7.2.3	Variation of Number of Packets Per Second . . . . .	47

7.2.4	Variation of Tx Range . . . . .	47
<b>8</b>	<b>Conclusion</b>	<b>47</b>

# 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.

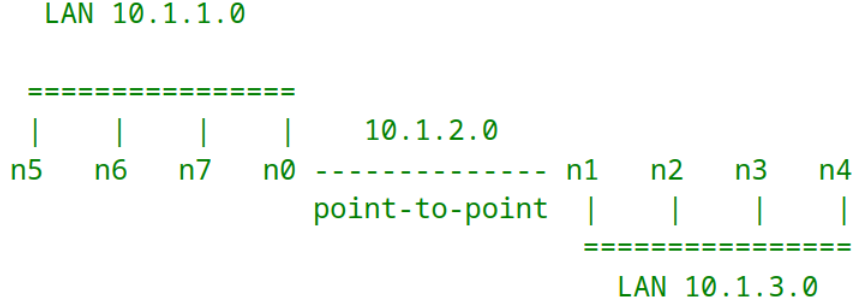


Figure 1: Wired Topology

## 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.

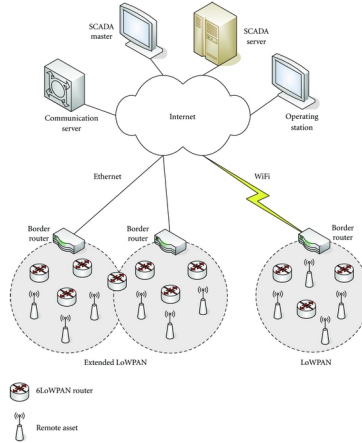


Figure 2: Wireless Low Rate Network Topology

## 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.

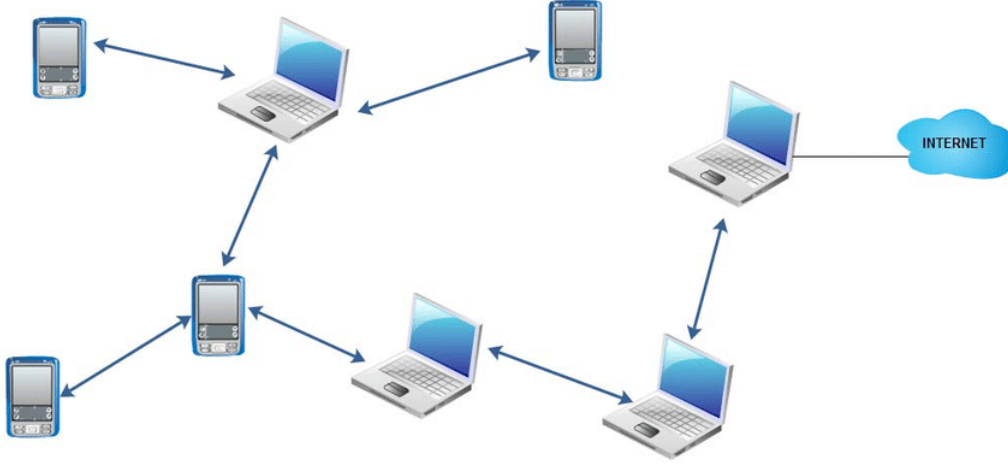


Figure 3: Adhoc Network Topology

## 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

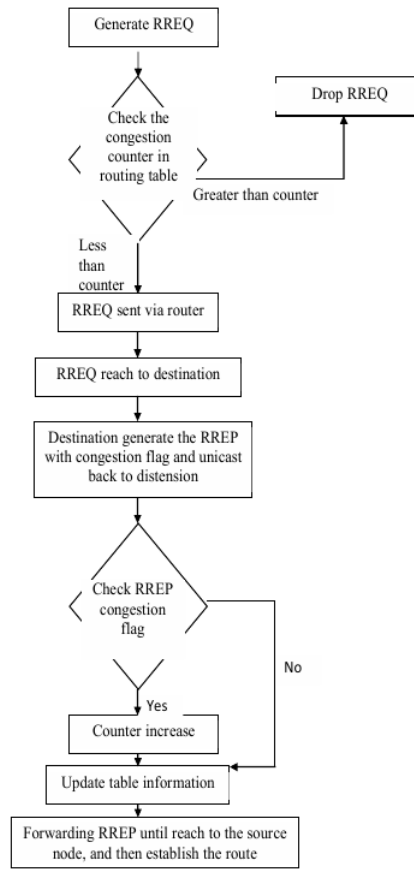


Figure 4: Process of CC-AODV flow chart

### 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.

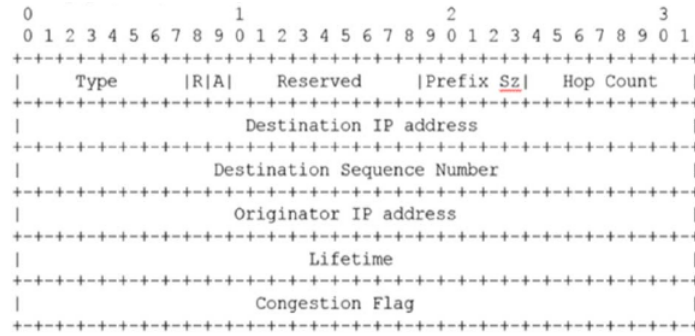


Figure 5: New-RREP Packets

## 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

```

C aody-packet.h > {} ns3 > {} aodv > RrepHeader > m_congestionFlag
473 */
474 bool operator== (RrepHeader const & o) const;
475 private:
476 uint8_t m_flags;          ///< A - acknowledgment
477 required flag
478 uint8_t m_prefixSize;     ///< Prefix Size
479 uint8_t m_hopCount;       ///< Hop Count
480 Ipv4Address m_dst;        ///< Destination IP Address
481 uint32_t m_dstSeqNo;      ///< Destination Sequence Number
482 Ipv4Address m_origin;     ///< Source IP Address
483 uint32_t m_lifetime;      ///< Lifetime (in milliseconds)
484 };
477 */
478 bool operator== (RrepHeader const & o) const;
479 private:
480 uint8_t m_flags;          ///< A - acknowledgment
481 required flag
482 uint8_t m_prefixSize;     ///< Prefix Size
483 uint8_t m_hopCount;       ///< Hop Count
484 Ipv4Address m_dst;        ///< Destination IP Address
485 uint32_t m_dstSeqNo;      ///< Destination Sequence Number
486 Ipv4Address m_origin;     ///< Source IP Address
487 uint32_t m_lifetime;      ///< Lifetime (in milliseconds)
488+ uint32_t m_congestionFlag;
489 };

```

```

C aody-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
442 */
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;
441+
442 // Flags
443 /**
444 * \brief Set the ack required flag
445 * \param f the ack required flag
446 */

```

```

C aody-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,
349 Ipv4Address dst =
350 Ipv4Address (), uint32_t dstSeqNo = 0, Ipv4Address
351 Ipv4Address (), Time lifetime = MilliSeconds (0));
352
353 * \brief Get the type ID.
354 * \return the object TypeId
355 static TypeId GetTypeId ();
356 TypeId GetInstanceTypeId () const;
357 uint32_t GetSerializedSize () 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,
349 Ipv4Address dst =
350 Ipv4Address (), uint32_t dstSeqNo = 0, Ipv4Address
351 Ipv4Address (), Time lifetime = MilliSeconds (0),
352+ int32_t congestionFlag = 0);
353
354 * \brief Get the type ID.
355 * \return the object TypeId
356 static TypeId GetTypeId ();
357 TypeId GetInstanceTypeId () const;
358 uint32_t GetSerializedSize () const;

```

```

C aody-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,
300 Ipv4Address dst,
301+ uint32_t dstSeqNo, Ipv4Address origin, Time
302+ lifetime, int32_t congestionFlag)
303
304 : m_flags (0),
305 m_prefixSize (prefixSize),
306 m_hopCount (hopCount),
307 m_dst (dst),
308 m_dstSeqNo (dstSeqNo),
309 m_origin (origin)
310 {
311 m_lifetime = uint32_t (lifetime.GetMilliSeconds ());
312+ m_congestionFlag = congestionFlag;
313 }
298 -----
299 RrepHeader::RrepHeader (uint8_t prefixSize, uint8_t hopCount,
300 Ipv4Address dst,
301+ uint32_t dstSeqNo, Ipv4Address origin, Time
302+ lifetime, int32_t congestionFlag)
303
304 : m_flags (0),
305 m_prefixSize (prefixSize),
306 m_hopCount (hopCount),
307 m_dst (dst),
308 m_dstSeqNo (dstSeqNo),
309 m_origin (origin)
310 {
311 m_lifetime = uint32_t (lifetime.GetMilliSeconds ());
312+ m_congestionFlag = congestionFlag;
313 }

```

```

aody-packet.cc > {} ns3 > {} aody > RrepHeader(uint8_t, uint8_t, Ipv4Address, uint32_t, Ipv4Address, Time, int32_t)
329
330 uint32_t
331 RrepHeader::GetSerializedSize () const
332 {
333- return 19;
334 }
335
336 void
337 RrepHeader::Serialize (Buffer::Iterator i) const
338 {
339     i.WriteU8 (m_flags);
340     i.WriteU8 (m_prefixSize);
341     i.WriteU8 (m_hopCount);
342     WriteTo (i, m_dst);
343     i.WriteHtonU32 (m_dstSeqNo);
344     WriteTo (i, m_origin);
345     i.WriteHtonU32 (m_lifeTime);
346 }
347
348 uint32_t
349 RrepHeader::Deserialize (Buffer::Iterator start)
350 {
351     Buffer::Iterator i = start;
352
353     m_flags = i.ReadU8 ();
354     m_prefixSize = i.ReadU8 ();
355     m_hopCount = i.ReadU8 ();
356     ReadFrom (i, m_dst);
357     m_dstSeqNo = i.ReadNtohU32 ();
358     ReadFrom (i, m_origin);
359     m_lifeTime = i.ReadNtohU32 ();
360
361     uint32_t + dist = i.GetDistanceFrom (start);
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427-
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

aody-packet.cc > {} ns3 > {} aody > RrepHeader(uint8_t, uint8_t, Ipv4Address, uint32_t, Ipv4Address, Time, int32_t)
373
374- os << " source ipv4 " << m_origin << " lifetime " << m_lifeTime
375- << " acknowledgment required flag " << (*this).GetAckRequired ()
376- ;
377
378 void
379 RrepHeader::SetLifeTime (Time t)
380 {
381     m_lifeTime = t.GetMilliSeconds ();
382 }
383
384 Time
385 RrepHeader::GetLifeTime () const
386 {
387     Time t (MilliSeconds (m_lifeTime));
388     return t;
389 }
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

aody-packet.cc > {} ns3 > {} aody > RrepHeader(uint8_t, uint8_t, Ipv4Address, uint32_t, Ipv4Address, Time, int32_t)
418 {
419     return m_prefixSize;
420 }
421
422 bool
423 RrepHeader::operator== (RrepHeader const & o) const
424 {
425     return (m_flags == o.m_flags && m_prefixSize == o.m_prefixSize
426             && m_hopCount == o.m_hopCount && m_dst == o.m_dst &&
427             m_dstSeqNo == o.m_dstSeqNo
428             && m_origin == o.m_origin && m_lifeTime == o.m_lifeTime);
429 }
430
431 void
432 RrepHeader::SetHello (Ipv4Address origin, uint32_t srcSeqNo, Time
lifetime)
433 {
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

## 4.2 Congestion Flag in RoutingTableEntry

<pre> C aodv-rtable.h &gt; {} ns3 &gt; {} aodv &gt; RoutingTableEntry &gt; m_congestionFlag 371 RouteFlags m_flag; 372 373 /// List of precursors 374 std::vector&lt;Ipv4Address&gt; m_precursorList; 375 /// When I can send another request 376 Time m_routeRequestTimeout; 377 /// Number of route requests 378 uint8_t m_reqCount; 379 /// Indicate if this entry is in "blacklist" 380 bool m_blackListState; 381 /// Time for which the node is put into the blacklist 382 Time m_blackListTimeout; 383 }; </pre>	<pre> 371 RouteFlags m_flag; 372 373 /// List of precursors 374 std::vector&lt;Ipv4Address&gt; m_precursorList; 375 /// When I can send another request 376 Time m_routeRequestTimeout; 377 /// Number of route requests 378 uint8_t m_reqCount; 379 /// Indicate if this entry is in "blacklist" 380 bool m_blackListState; 381 /// Time for which the node is put into the blacklist 382 Time m_blackListTimeout; 383+ 384+ int32_t m_congestionFlag; 385 }; </pre>
<pre> C aodv-rtable.h &gt; {} ns3 &gt; {} aodv &gt; RoutingTableEntry &gt; m_congestionFlag 59 class RoutingTableEntry 60 { 61 public: 62 /** 63  * constructor 64  */ 65  * lparam dev the device 66  * lparam dst the destination IP address 67  * lparam vSeqNo verify sequence number flag 68  * lparam seqNo the sequence number 69  * lparam iface the interface 70  * lparam hops the number of hops 71  * lparam nextHop the IP address of the next hop 72  * lparam lifetime the lifetime of the entry 73  */ 74 RoutingTableEntry (Ptr&lt;NetDevice&gt; dev = 0, Ipv4Address dst = 75 Ipv4Address (), bool vSeqNo = false, uint32_t seqNo = 0, 76 Ipv4InterfaceAddress iface = 77 Ipv4InterfaceAddress (), uint16_t hops = 0, 78 Ipv4Address nextHop = Ipv4Address (), Time 79 lifetime = Simulator::Now ()); 80 81 ~RoutingTableEntry (); </pre>	<pre> 59 class RoutingTableEntry 60 { 61 public: 62 /** 63  * constructor 64  */ 65  * lparam dev the device 66  * lparam dst the destination IP address 67  * lparam vSeqNo verify sequence number flag 68  * lparam seqNo the sequence number 69  * lparam iface the interface 70  * lparam hops the number of hops 71  * lparam nextHop the IP address of the next hop 72  * lparam lifetime the lifetime of the entry 73  */ 74 RoutingTableEntry (Ptr&lt;NetDevice&gt; dev = 0, Ipv4Address dst = 75 Ipv4Address (), bool vSeqNo = false, uint32_t seqNo = 0, 76 Ipv4InterfaceAddress iface = 77 Ipv4InterfaceAddress (), uint16_t hops = 0, 78 Ipv4Address nextHop = Ipv4Address (), Time 79 lifetime = Simulator::Now ()), int32_t 80 congestionFlag = 0); 81 82 ~RoutingTableEntry (); </pre>
<pre> C aodv-rtable.cc &gt; {} ns3 &gt; {} aodv 44 RoutingTableEntry::RoutingTableEntry (Ptr&lt;NetDevice&gt; dev, 45 Ipv4Address dst, bool vSeqNo, uint32_t seqNo, 46 Ipv4InterfaceAddress iface, 47 uint16_t hops, Ipv4Address 48 nextHop, Time lifetime) 49 { 50 m_ackTimer (Timer::CANCEL_ON_DESTROY), 51 m_validSeqNo (vSeqNo), 52 m_seqNo (seqNo), 53 m_hops (hops), 54 m_lifetime (lifetime + Simulator::Now ()), 55 m_iface (iface), 56 m_flag (VALID), 57 m_reqCount (0), 58 m_blackListState (false), 59 m_blackListTimeout (Simulator::Now ()) 60 { 61 m_ipv4Route = Create&lt;Ipv4Route&gt; (); 62 m_ipv4Route-&gt;SetDestination (dst); </pre>	<pre> 44 RoutingTableEntry::RoutingTableEntry (Ptr&lt;NetDevice&gt; dev, 45 Ipv4Address dst, bool vSeqNo, uint32_t seqNo, 46 Ipv4InterfaceAddress iface, 47 uint16_t hops, Ipv4Address 48 nextHop, Time lifetime, 49 int32_t congestionFlag) 50 { 51 m_ackTimer (Timer::CANCEL_ON_DESTROY), 52 m_validSeqNo (vSeqNo), 53 m_seqNo (seqNo), 54 m_hops (hops), 55 m_lifetime (lifetime + Simulator::Now ()), 56 m_iface (iface), 57 m_flag (VALID), 58 m_reqCount (0), 59 m_blackListState (false), 60 m_blackListTimeout (Simulator::Now ()) 61 { 62 m_ipv4Route = Create&lt;Ipv4Route&gt; (); 63 m_ipv4Route-&gt;SetDestination (dst); </pre>

## 4.3 Add Congestion Counter and Max Count

<pre> C aodv-routing-protocol.cc &gt; MAX_CONGESTION_COUNT 46 #include &lt;limits&gt; 47 48 namespace ns3 { 49 50 NS_LOG_COMPONENT_DEFINE ("AodvRoutingProtocol"); 51 52 namespace aodv { 53 NS_OBJECT_ENSURE_REGISTERED (RoutingProtocol); 54 </pre>	<pre> 46 #include &lt;limits&gt; 47 48+ #define MAX_CONGESTION_COUNT 4 49+ 50 namespace ns3 { 51 52 NS_LOG_COMPONENT_DEFINE ("AodvRoutingProtocol"); 53 54 namespace aodv { 55 NS_OBJECT_ENSURE_REGISTERED (RoutingProtocol); 56 </pre>
---	---

```

C aadv-routing-protocol.h > { } ns3 > { } aadv > RoutingProtocol > m_congestionCounter
251 // Handle duplicated RREQ
252 IdCache m_rreqIdCache;
253 // Handle duplicated broadcast/multicast packets
254 DuplicatePacketDetection m_dpd;
255 // Handle neighbors
256 Neighbors m_nb;
257 // Number of RREQs used for RREQ rate control
258 uint16_t m_rreqCount;
259 // Number of RRRs used for RRR rate control
260 uint16_t m_rerrCount;
261
251 // Handle duplicated RREQ
252 IdCache m_rreqIdCache;
253 // Handle duplicated broadcast/multicast packets
254 DuplicatePacketDetection m_dpd;
255 // Handle neighbors
256 Neighbors m_nb;
257 // Number of RREQs used for RREQ rate control
258 uint16_t m_rreqCount;
259 // Number of RRRs used for RRR rate control
260 uint16_t m_rerrCount;
261
262+ uint32_t m_congestionCounter; // You, a minutes ago
263+

```

```

aadv-routing-protocol.cc > { } ns3 > { } aadv > RoutingProtocol()
169 m_rreqIdCache (m_pathDiscoveryTime),
170 m_dpd (m_pathDiscoveryTime),
171 m_nb (m_helloInterval),
172 m_rreqCount (0),
173 m_rerrCount (0),
174 m_htimer (Timer::CANCEL_ON_DESTROY),
175 m_rreqRateLimitTimer (Timer::CANCEL_ON_DESTROY),
176 m_rerrRateLimitTimer (Timer::CANCEL_ON_DESTROY),
177 m_lastBcastTime (Seconds (0))
178 {
179 m_nb.SetCallback (MakeCallback (&
RoutingProtocol::SendRerrWhenBreaksLinkToNextHop, this));
180 }

171 m_rreqIdCache (m_pathDiscoveryTime),
172 m_dpd (m_pathDiscoveryTime),
173 m_nb (m_helloInterval),
174 m_rreqCount (0),
175 m_rerrCount (0),
176+ m_congestionCounter(0); // You, a minutes ago
177 m_htimer (Timer::CANCEL_ON_DESTROY),
178 m_rreqRateLimitTimer (Timer::CANCEL_ON_DESTROY),
179 m_rerrRateLimitTimer (Timer::CANCEL_ON_DESTROY),
180 m_lastBcastTime (Seconds (0))
181 {
182 m_nb.SetCallback (MakeCallback (&
RoutingProtocol::SendRerrWhenBreaksLinkToNextHop, this));
183+ // std::cout<<"Congestion Counter = "<<m_congestionCounter<<" and
+ Max Congestion Count set to "<<MAX_CONGESTION_COUNT<<std::endl;
184 }

```

```

aadv-routing-protocol.cc > { } ns3 > { } aadv > GetTypeId(void)
234 .AddAttribute ("MyRouteTimeout", "Value of lifetime field in
RREP generating by this node = 2 * max(ActiveRouteTimeout,
PathDiscoveryTime)",
235 TimeValue (Seconds (11.2)),
236 MakeTimeAccessor (&
RoutingProtocol::m_myRouteTimeout),
237 MakeTimeChecker ())
238 .AddAttribute ("BlackListTimeout", "Time for which the node is
put into the blacklist = RreqRetries * NetTraversalTime",
239 TimeValue (Seconds (5.6)),
240 MakeTimeAccessor (&
RoutingProtocol::m_blackListTimeout),
241 MakeTimeChecker ())
242+ .AddAttribute ("m_congestionCounter", "m_congestionCounter.",
243+ UIntegerValue (0),
244+ MakeUIntegerAccessor (&
RoutingProtocol::m_congestionCounter),
245+ MakeUIntegerChecker (<uint32_t> ())) // You, a minutes ago
246 .AddAttribute ("BlackListTimeout", "Time for which the node is
put into the blacklist = RreqRetries * NetTraversalTime",
247 TimeValue (Seconds (5.6)),
248 MakeTimeAccessor (&
RoutingProtocol::m_blackListTimeout),
249 MakeTimeChecker ())

```

## 4.4 Drop RREQ Counter Greater than Max Count

```

aadv-routing-protocol.cc > { } ns3 > { } aadv > RecvRequest(Ptr<Packet>, Ipv4Address, Ipv4Address)
1248 * Node checks to determine whether it has received a RREQ with
the same Originator IP Address and RREQ ID.
1249 * If such a RREQ has been received, the node silently discards
the newly received RREQ.
1250 */
1251 if (m_rreqIdCache.IsDuplicate (origin, id))
1252 {
1253 NS_LOG_DEBUG ("Ignoring RREQ due to duplicate");
1254 return;
1255 }
1256
1257 // Increment RREQ hop count
uint8_t hon = rreqHeader.GetHopCount () + 1;

1256 * Node checks to determine whether it has received a RREQ with
the same Originator IP Address and RREQ ID.
1257 * If such a RREQ has been received, the node silently discards
the newly received RREQ.
1258 */
1259 if (m_rreqIdCache.IsDuplicate (origin, id))
1260 {
1261 NS_LOG_DEBUG ("Ignoring RREQ due to duplicate");
1262 return;
1263 }
1264
1265+ if (m_congestionCounter > MAX_CONGESTION_COUNT){
1266+ // std::cout<<"Congestion Counter reached Max Count"<<std::endl;
1267+ NS_LOG_DEBUG ("Ignoring RREQ due to congestion max count
reached");
1268+ return;
1269+ } // You, a minutes ago
1270+
1271 // Increment RREQ hop count
uint8_t hon = rreqHeader.GetHopCount () + 1;

```

## 4.5 Counter Increment or Decrement and Flag Operations

```

aodv-routing-protocol.cc > {} ns3 > {} aodv > SendReply(RreqHeader const &, RoutingTableEntry const &)
1413 void
1414 RoutingProtocol::SendReply (RreqHeader const & rreqHeader,
1415 RoutingTableEntry const & toOrigin)
1416 {
1417     NS_LOG_FUNCTION (this << toOrigin.GetDestination ());
1418     /*
1419      * Destination node MUST increment its own sequence number by one
1420      * if the sequence number in the RREQ packet is equal to that
1421      * incremented value. Otherwise, the destination does not change
1422      * its sequence number before generating the RREP message.
1423      */
1424     if (!rreqHeader.GetUnknownSeqNo () && (rreqHeader.GetDstSeqNo ()
1425 == m_seqNo + 1))
1426     {
1427         m_seqNo++;
1428     }
1429     RrepHeader rrepHeader ( /*prefixSize=*/ 0, /*hops=*/ 0, /*dst=*/
1430 rreqHeader.GetDst (),
1431 /*dstSeqNo=*/ m_seqNo, /
1432 /*origin=*/ toOrigin.
1433 GetDestination (), /
1434 /*lifeTime=*/
1435 m_myRouteTimeout);
1436 }
1437
1438 void
1439 RoutingProtocol::SendReply (RreqHeader const & rreqHeader,
1440 RoutingTableEntry const & toOrigin)
1441 {
1442     NS_LOG_FUNCTION (this << toOrigin.GetDestination ());
1443     /*
1444      * Destination node MUST increment its own sequence number by one
1445      * if the sequence number in the RREQ packet is equal to that
1446      * incremented value. Otherwise, the destination does not change
1447      * its sequence number before generating the RREP message.
1448      */
1449     if (!rreqHeader.GetUnknownSeqNo () && (rreqHeader.GetDstSeqNo ()
1450 == m_seqNo + 1))
1451     {
1452         m_seqNo++;
1453     }
1454     RrepHeader rrepHeader ( /*prefixSize=*/ 0, /*hops=*/ 0, /*dst=*/
1455 rreqHeader.GetDst (),
1456 /*dstSeqNo=*/ m_seqNo, /
1457 /*origin=*/ toOrigin.
1458 GetDestination (), /
1459 /*lifeTime=*/
1460 m_myRouteTimeout, 1);
1461 // std::cout<<"Set Congestion Flag to 1"<<std::endl;
1462 }
1463
1464 aodv-routing-protocol.cc > {} ns3 > {} aodv > RecvReply(Ptr<Packet>, Ipv4Address, Ipv4Address)
1465 ProcessHello (rreqHeader, receiver);
1466 return;
1467 }
1468
1469 if(rreqHeader.GetCongestionFlag()==1){
1470     m_congestionCounter++;
1471     // std::cout<<"Congestion Counter ++"<<std::endl;
1472 }
1473
1474 /*
1475  * If the route table entry to the destination is created or
1476  * updated, then the following actions occur:
1477  * - the route is marked as active,
1478  * - the destination sequence number is marked as valid,
1479  * - the next hop in the route entry is assigned to be the node
1480  * from which the RREP is received,
1481  * which is indicated by the source IP address field in the IP
1482  * header,
1483  * - the hop count is set to the value of the hop count from
1484  * RREP message + 1
1485  * - the expiry time is set to the current time plus the value
1486  * of the lifetime in the RREP message,
1487  * - and the destination sequence number is the Destination
1488  * Sequence Number in the RREP message.
1489  */
1490 Ptr<NetDevice> dev = m_ipv4->GetNetDevice
1491 (m_ipv4->GetInterfaceForAddress (receiver));
1492 RoutingTableEntry newEntry (/*device=*/ dev, /*dst=*/ dst, /
1493 /*validSeqNo=*/ true, /*seqno=*/ rreqHeader.GetDstSeqNo (),
1494 /*iface=*/
1495 m_ipv4->GetAddress
1496 (m_ipv4->GetInterfaceForAd
1497 dress (receiver), 0), /
1498 /*hop=*/ hop,
1499 /*nextHop=*/ sender, /
1500 /*lifeTime=*/ rreqHeader.
1501 GetLifeTime ());
1502
1503 m_routingTable.LookupRoute (toOrigin.GetNextHop (),
1504 toNextHopToOrigin);
1505 toNextHopToOrigin.InsertPrecursor (toDst.GetNextHop ());
1506 m_routingTable.Update (toNextHopToOrigin);
1507 }
1508 SocketIpTtlTag tag;
1509 p->RemovePacketTag (tag);
1510 if (tag.GetTtl () < 2)
1511 {
1512     NS_LOG_DEBUG ("TTL exceeded. Drop RREP destination " << dst
1513 << " origin " << rreqHeader.GetOrigin ());
1514 }
1515 return;
1516 }
1517
1518 m_routingTable.LookupRoute (toOrigin.GetNextHop (),
1519 toNextHopToOrigin);
1520 toNextHopToOrigin.InsertPrecursor (toDst.GetNextHop ());
1521 m_routingTable.Update (toNextHopToOrigin);
1522 }
1523 SocketIpTtlTag tag;
1524 p->RemovePacketTag (tag);
1525 if (tag.GetTtl () < 2)
1526 {
1527     NS_LOG_DEBUG ("TTL exceeded. Drop RREP destination " << dst
1528 << " origin " << rreqHeader.GetOrigin ());
1529 }
1530 if(m_congestionCounter > 0){
1531     m_congestionCounter--;
1532     // std::cout<<"Congestion Counter -- (TTL exceeded)
1533     "<<std::endl;
1534 }
1535 // You, 5 minutes ago · Uncommitted changes
1536 return;
1537 }
1538
1539

```



```

aody-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 }
1757
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+   }
1785 }
1786

```

## 5 Paper Result vs Result of My Task B

### 5.1 Plot Graph

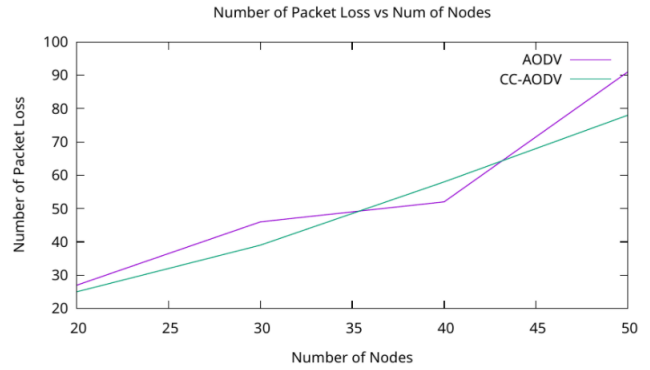
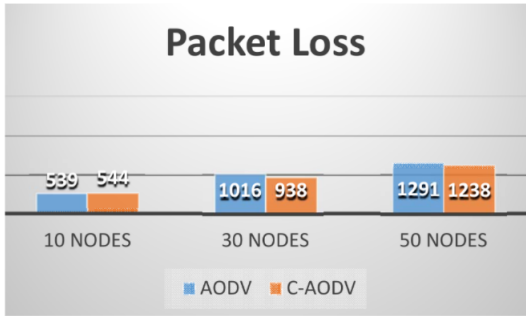


Figure 6: Left: Paper Result, Right: My Result

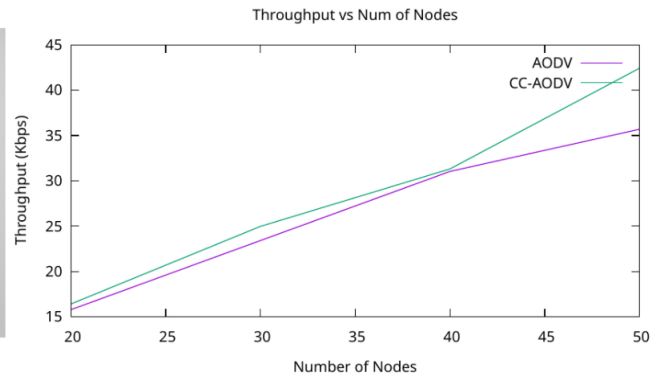
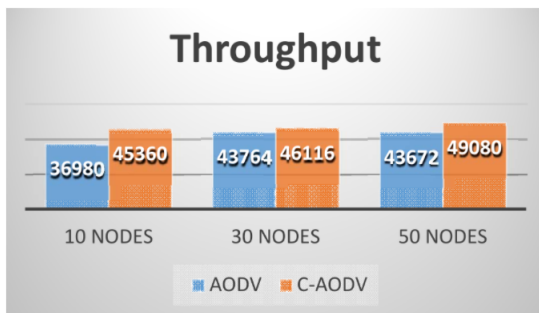


Figure 7: Left: Paper Result, Right: My Result

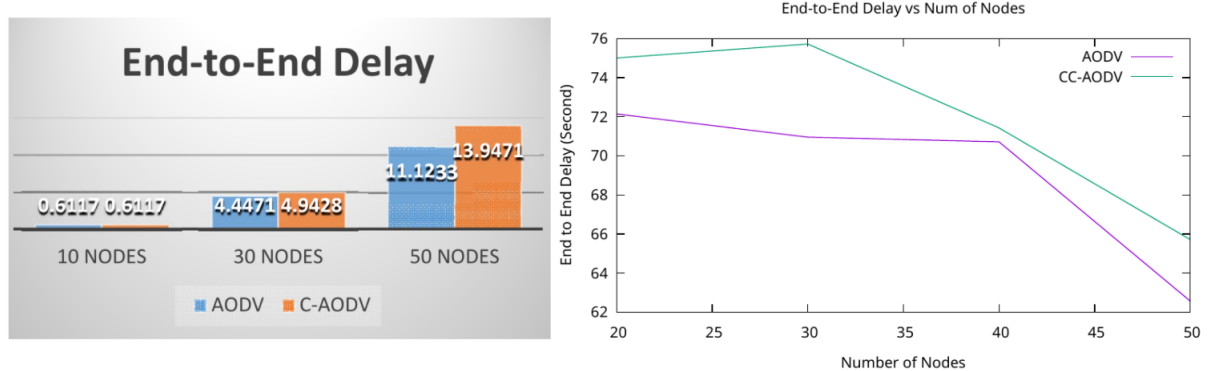


Figure 8: Left: Paper Result, Right: My Result

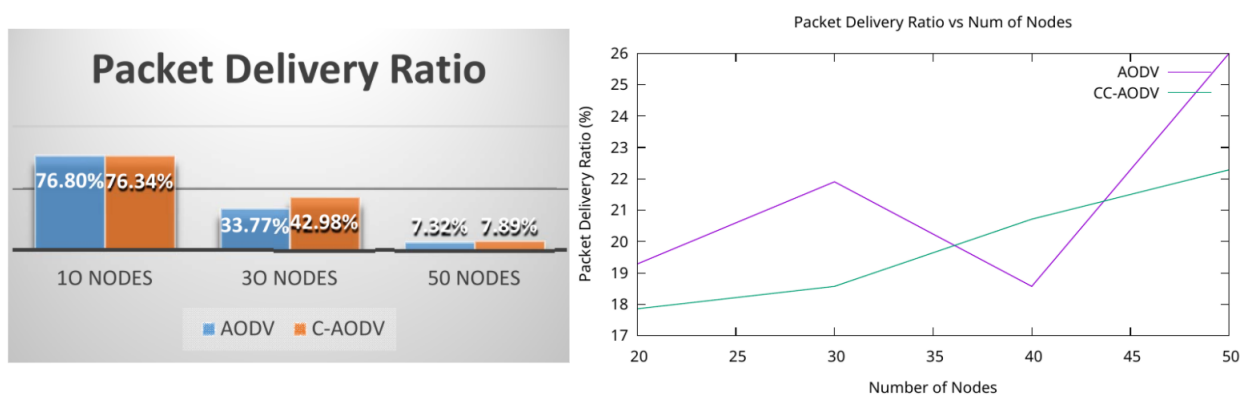


Figure 9: Left: Paper Result, Right: My Result

## 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**

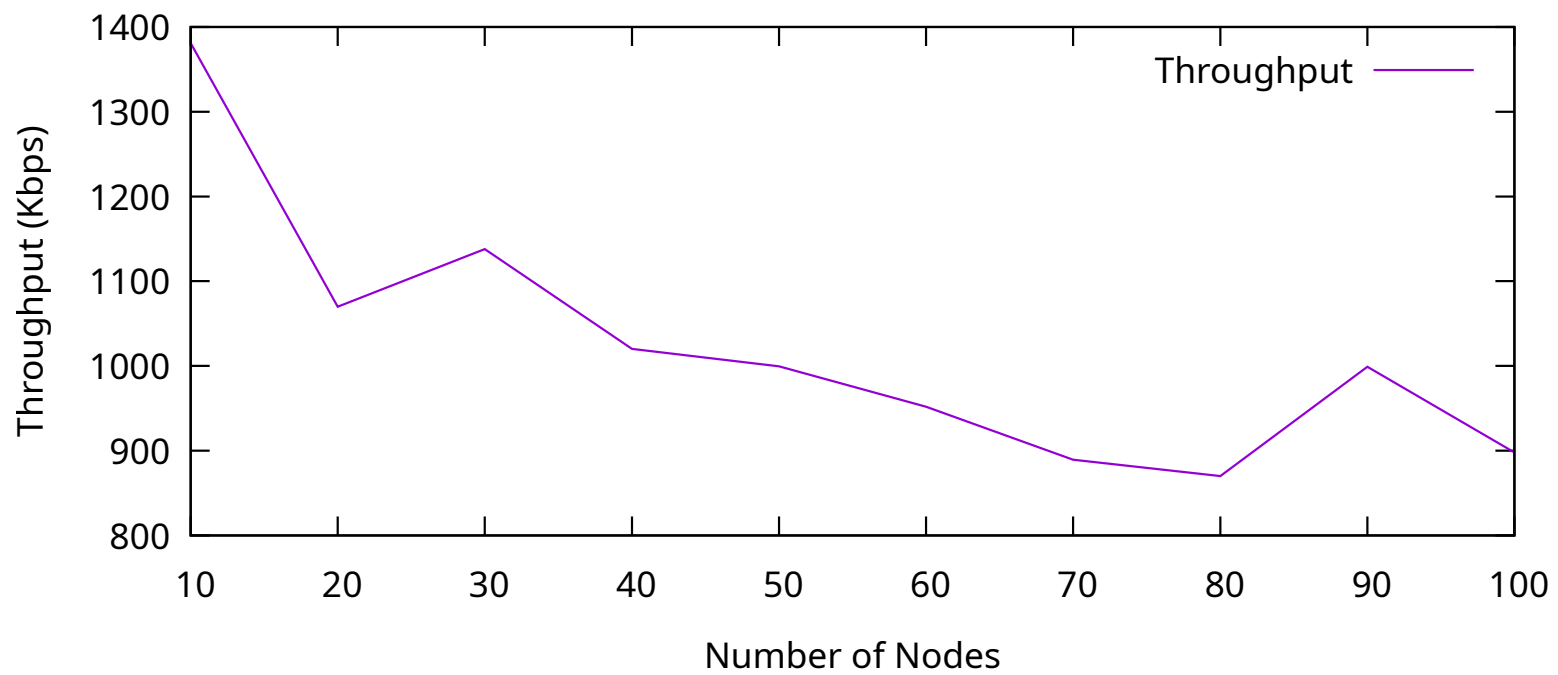
### **6.1 Plot Graph**

### TaskA Wired (Throughput vs Num of Nodes)

Number of Flows = 100

Number of Packets Per Second = 100

Packet Size = 4096 byte

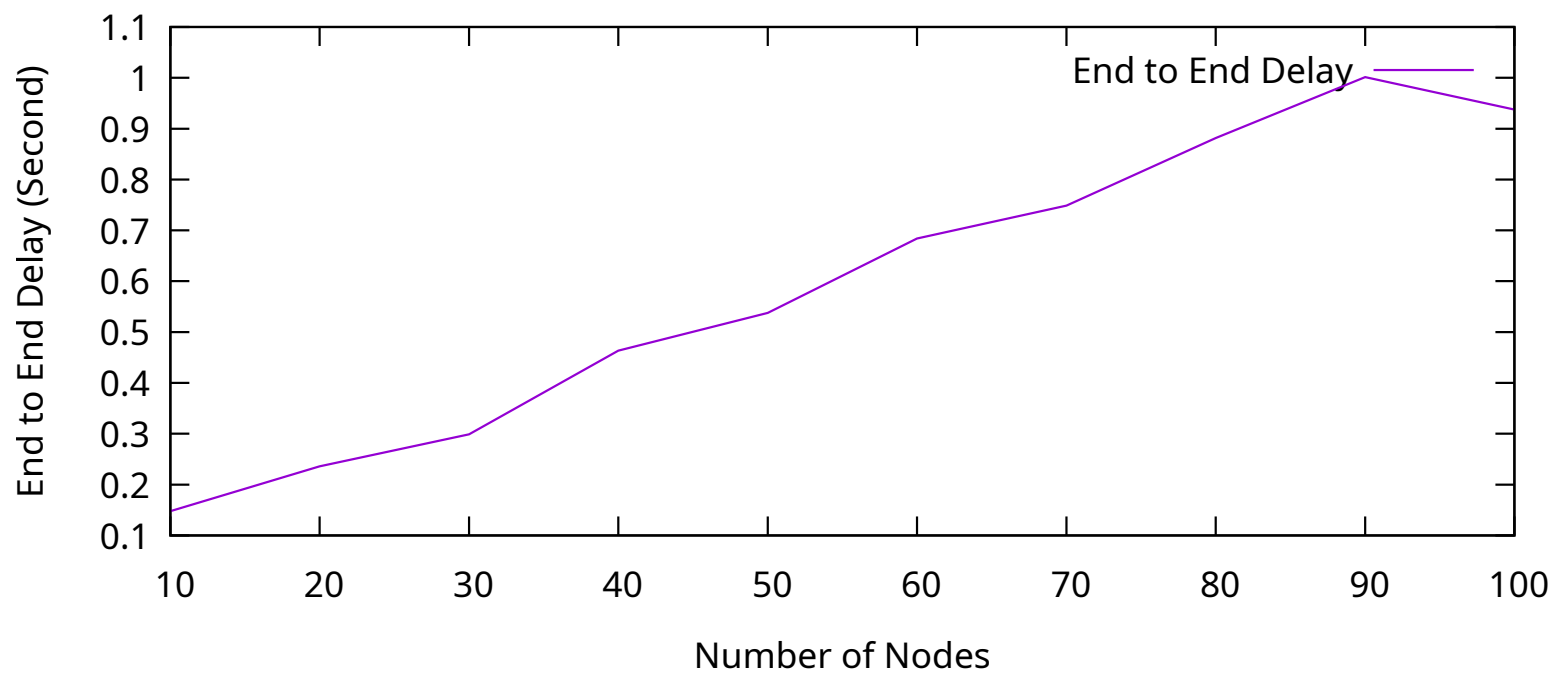


### TaskA Wired (End-to-End Delay vs Num of Nodes)

Number of Flows = 100

Number of Packets Per Second = 100

Packet Size = 4096 byte

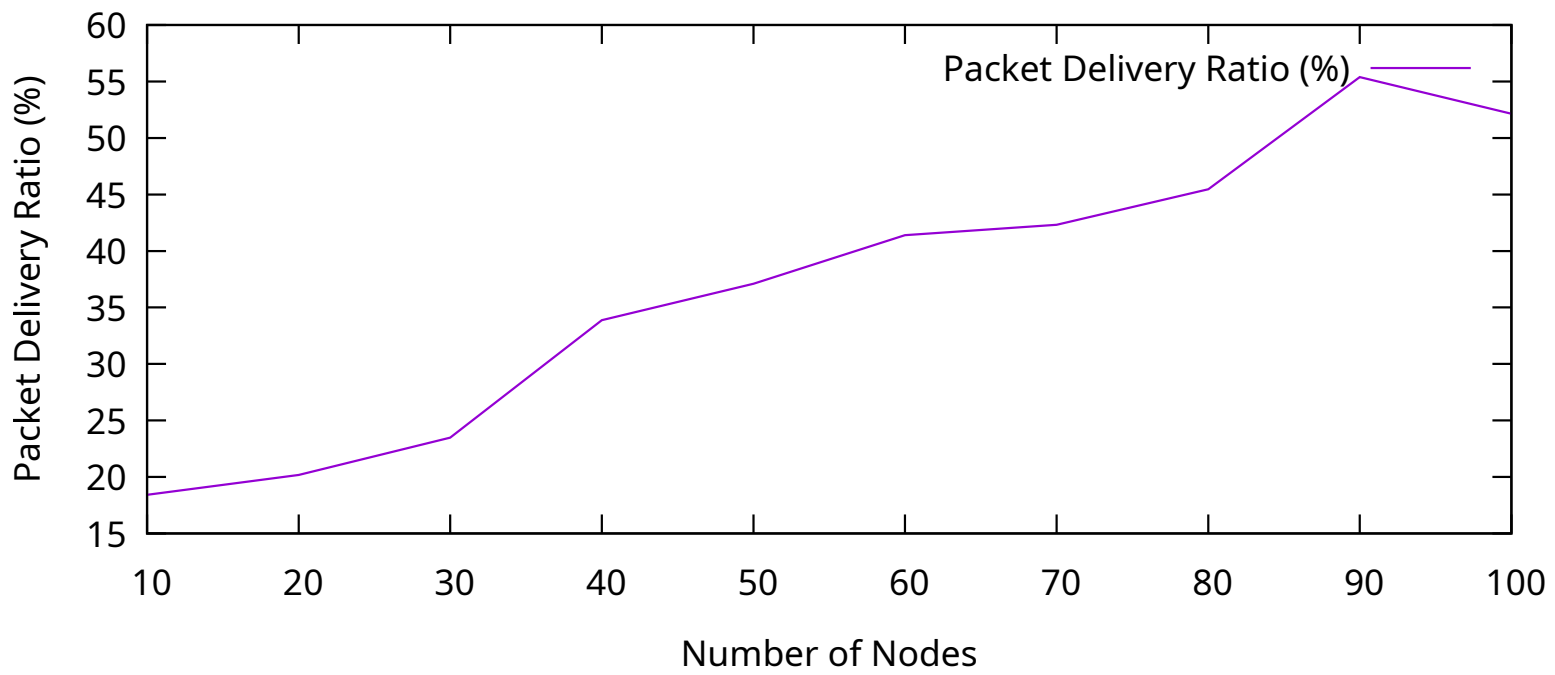


# TaskA Wired (Packet Delivery Ratio vs Num of Nodes)

Number of Flows = 100

Number of Packets Per Second = 100

Packet Size = 4096 byte

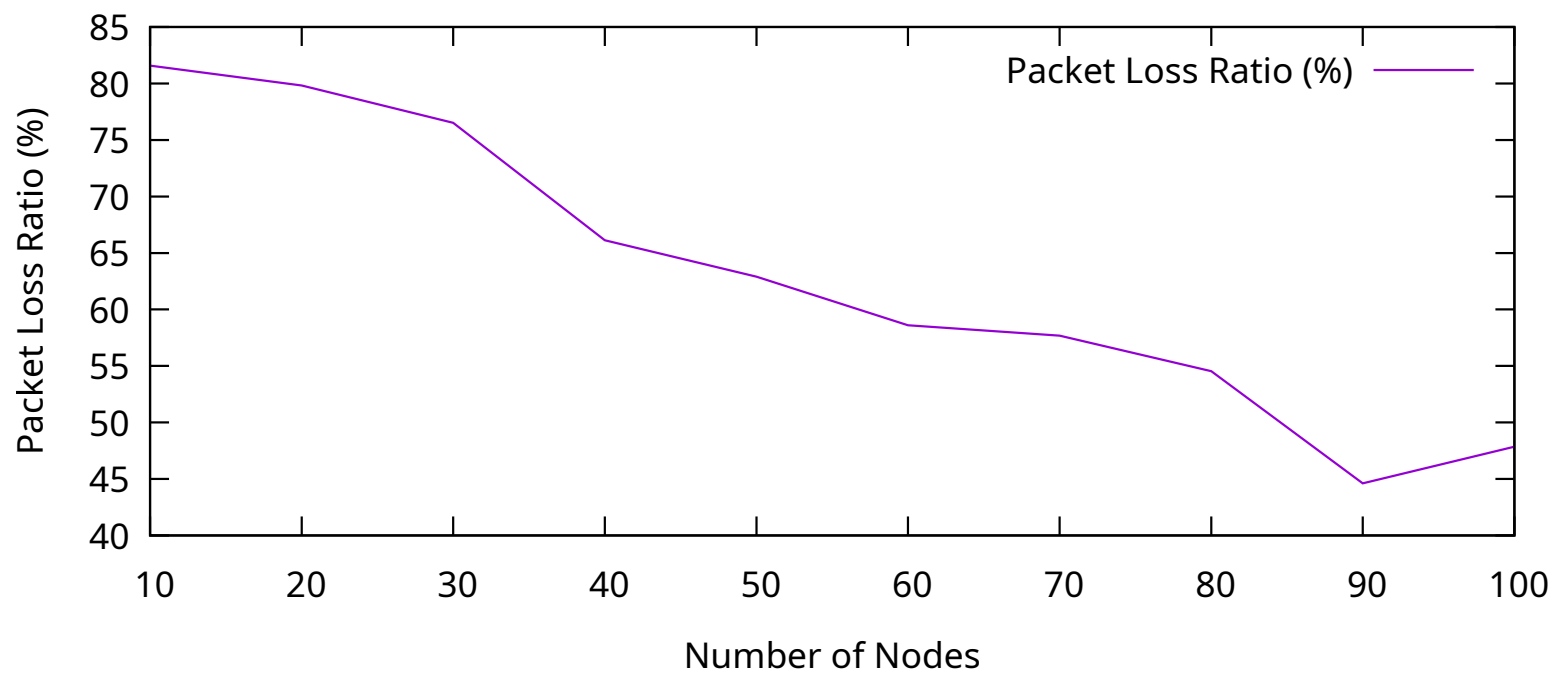


### TaskA Wired (Packet Loss Ratio vs Num of Nodes)

Number of Flows = 100

Number of Packets Per Second = 100

Packet Size = 4096 byte

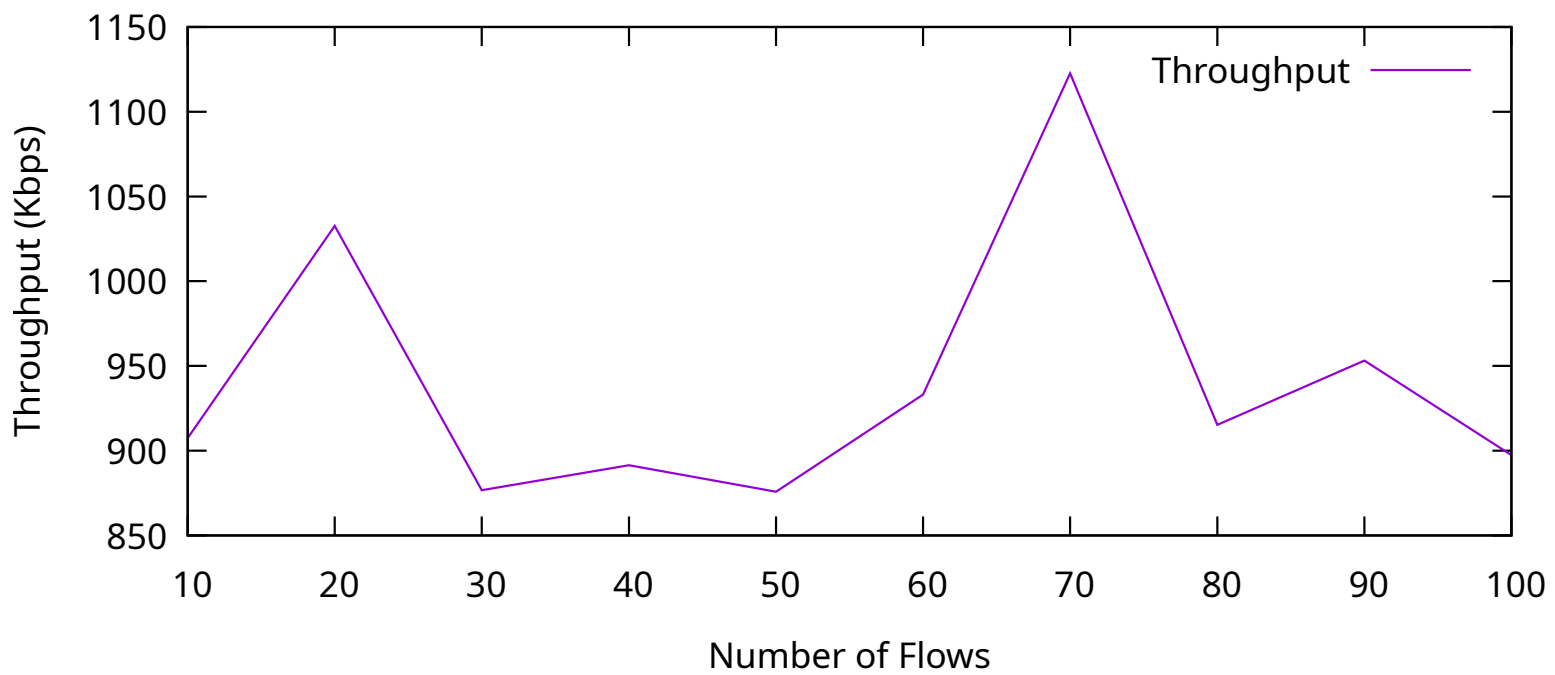


### TaskA Wired (Throughput vs Num of Flows)

Number of Nodes = 50

Number of Packets Per Second = 100

Packet Size = 4096 byte

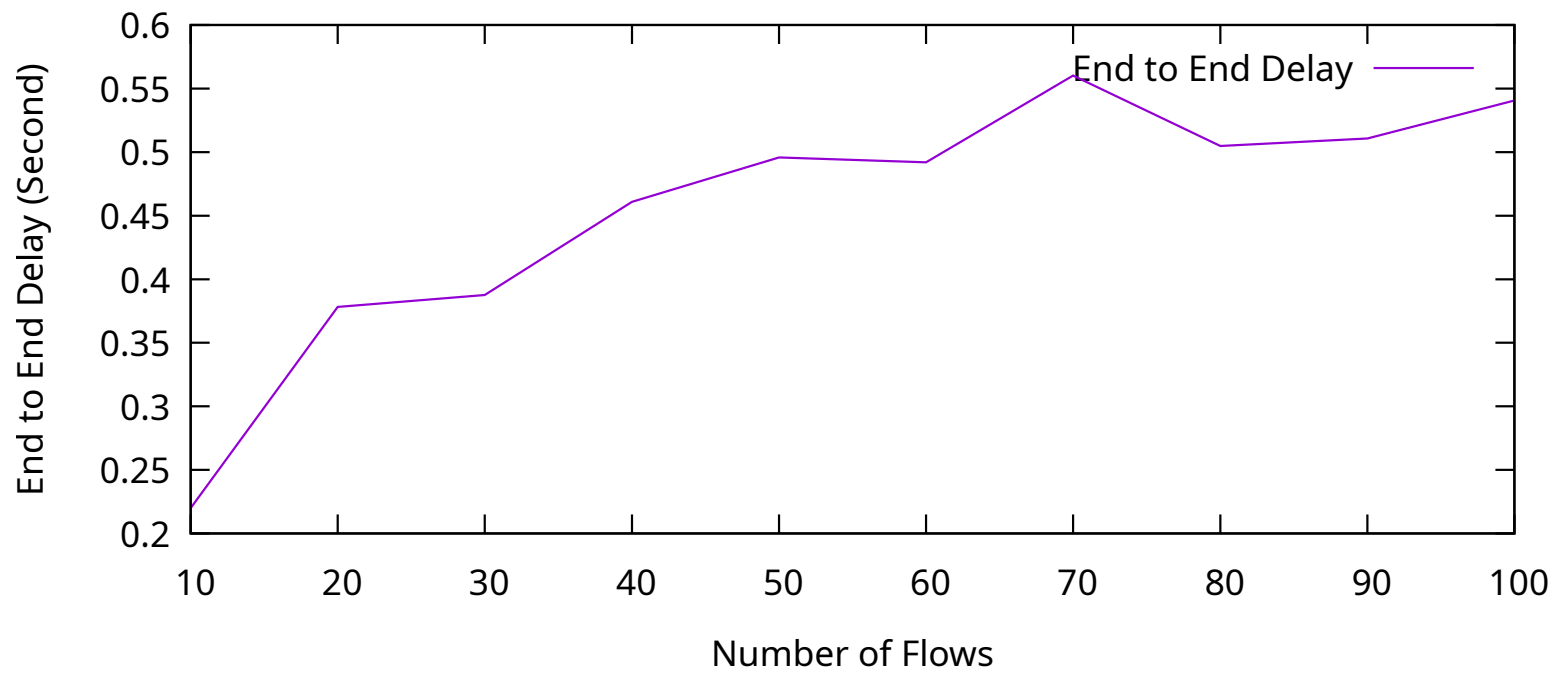


### TaskA Wired (End-to-End Delay vs Num of Flows)

Number of Nodes = 50

Number of Packets Per Second = 100

Packet Size = 4096 byte

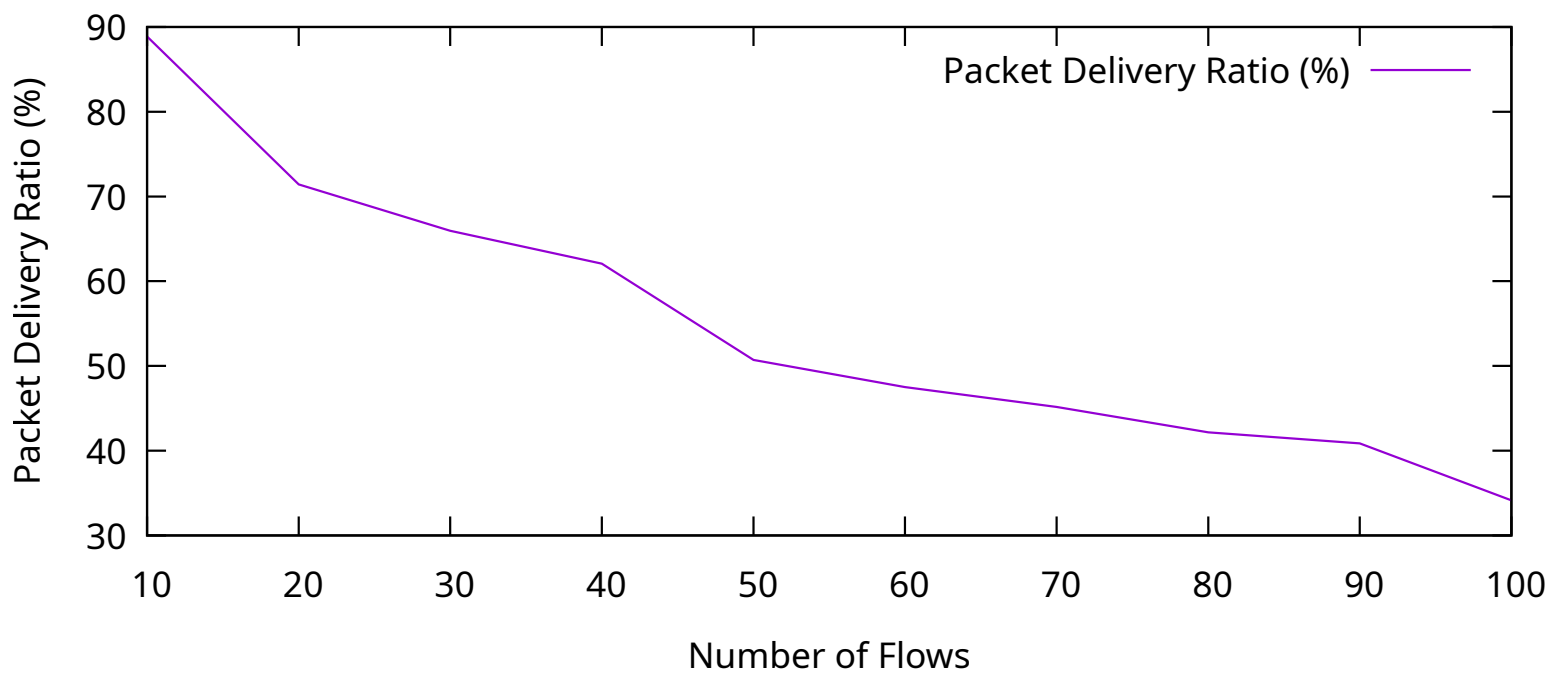


# TaskA Wired (Packet Delivery Ratio vs Num of Flows)

Number of Nodes = 50

Number of Packets Per Second = 100

Packet Size = 4096 byte



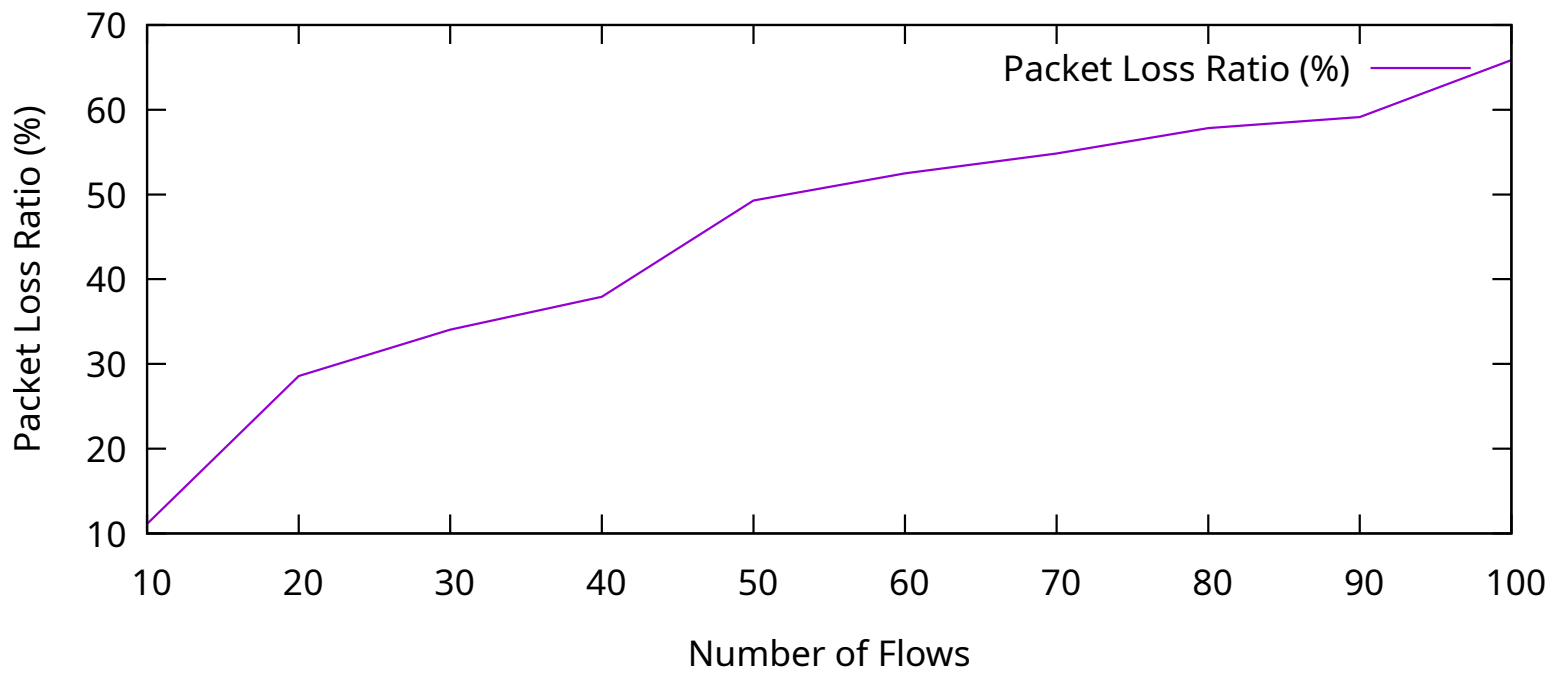


TaskA Wired (Packet Loss Ratio vs Num of Flows)

Number of Nodes = 50

Number of Packets Per Second = 100

Packet Size = 4096 byte

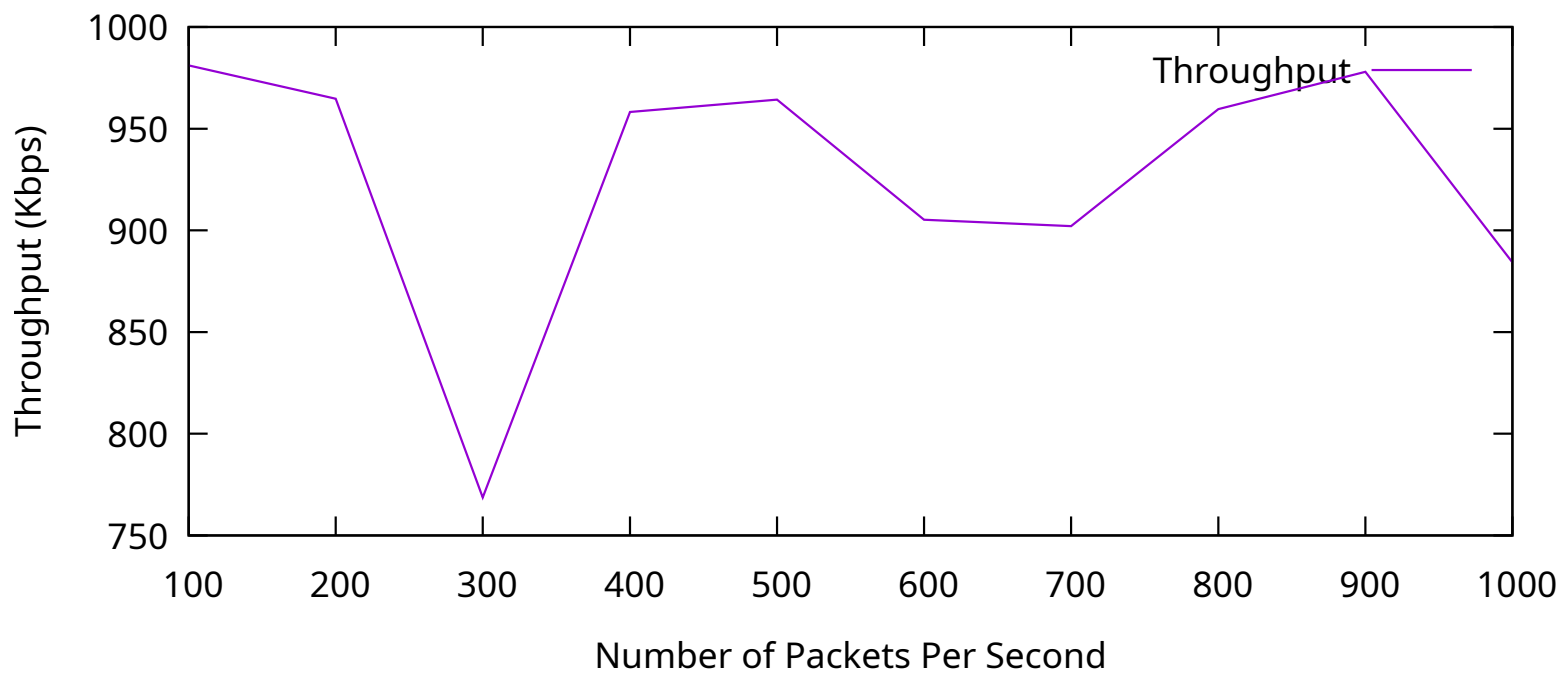


### TaskA Wired (Throughput vs Num of Packets Per Second)

Number of Nodes = 50

Number of Flows = 50

Packet Size = 4096 byte

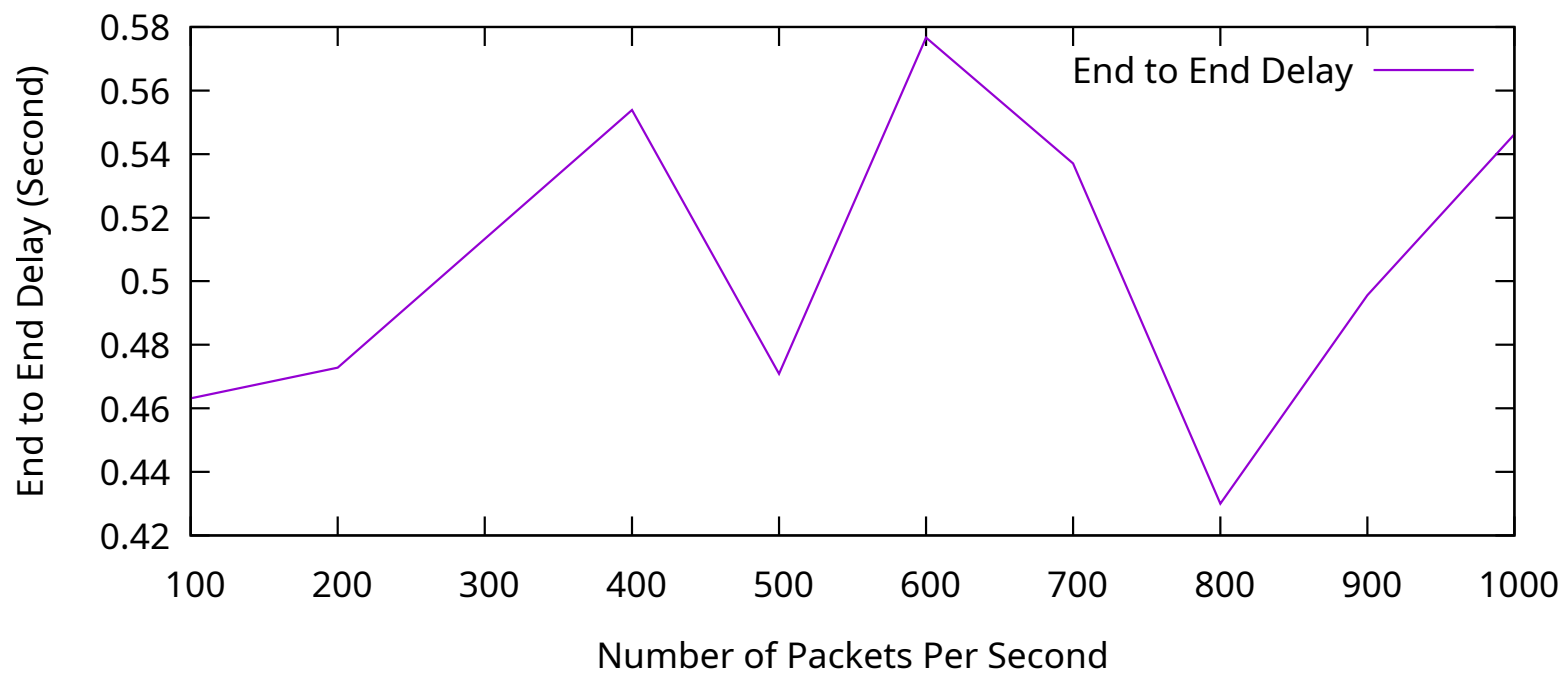


### TaskA Wired (End-to-End Delay vs Num of Packets Per Second)

Number of Nodes = 50

Number of Flows = 50

Packet Size = 4096 byte

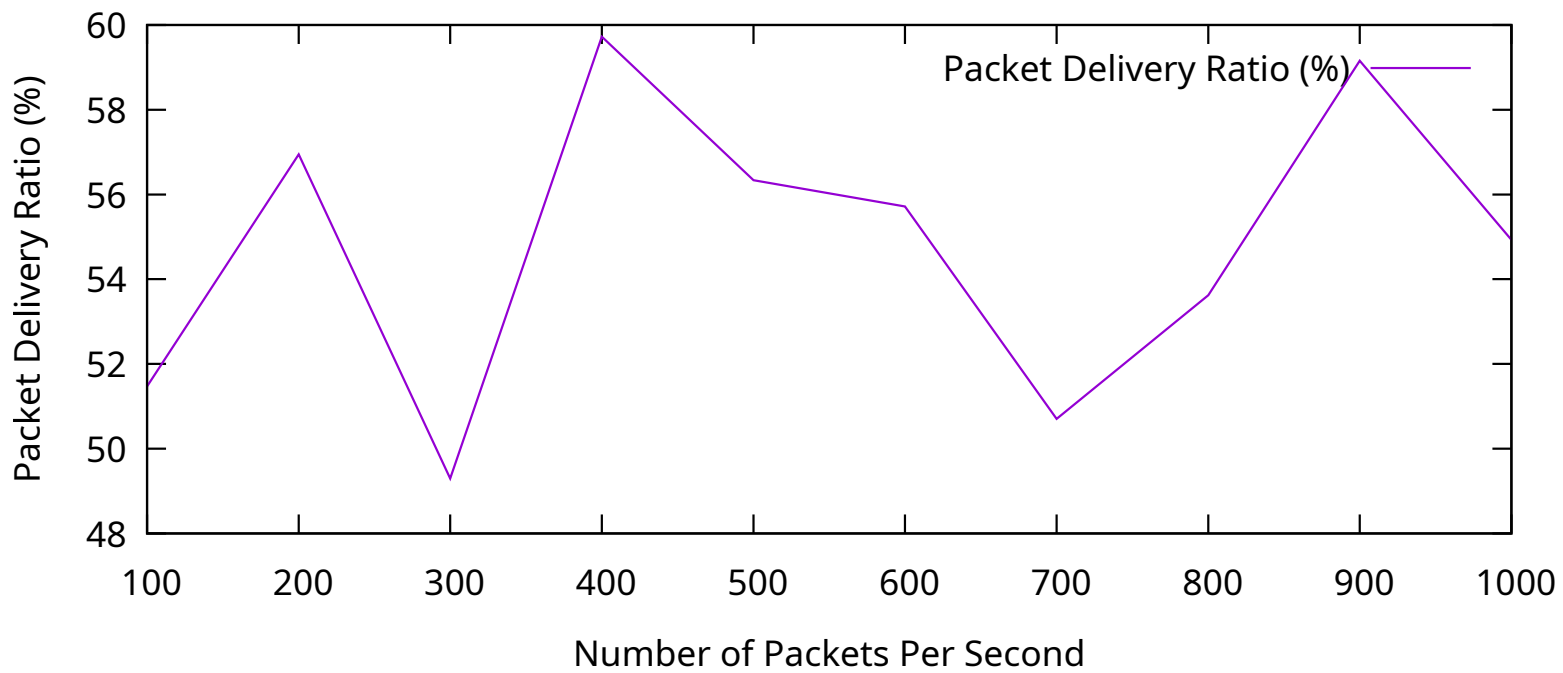


TaskA Wired (Packet Delivery Ratio vs Num of Packets Per Second)

Number of Nodes = 50

Number of Flows = 50

Packet Size = 4096 byte

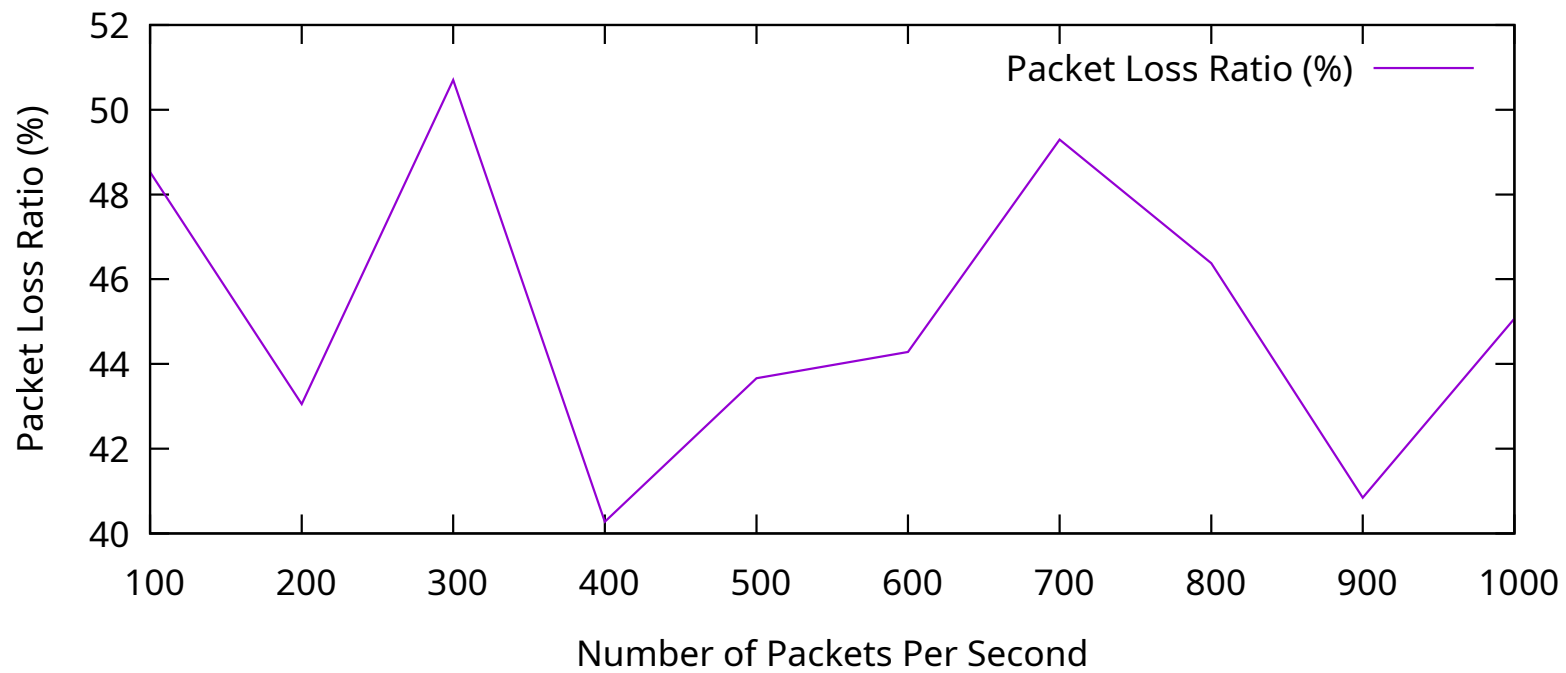


### TaskA Wired (Packet Loss Ratio vs Num of Packets Per Second)

Number of Nodes = 50

Number of Flows = 50

Packet Size = 4096 byte



## **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**

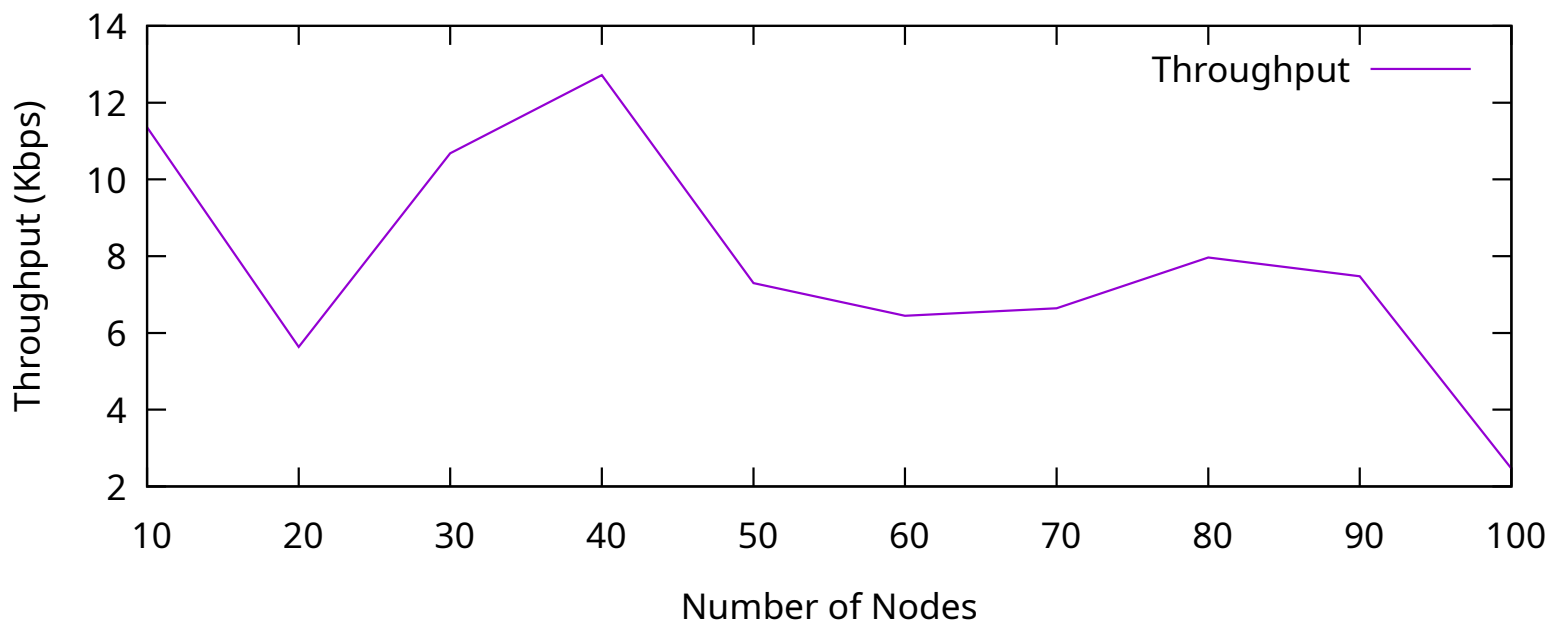
# TaskA Wireless LowRate (Throughput vs Num of Nodes)

Number of Flows = 100

Number of Packets Per Second = 100

Packet Size = 4096 byte

TX Range = 5



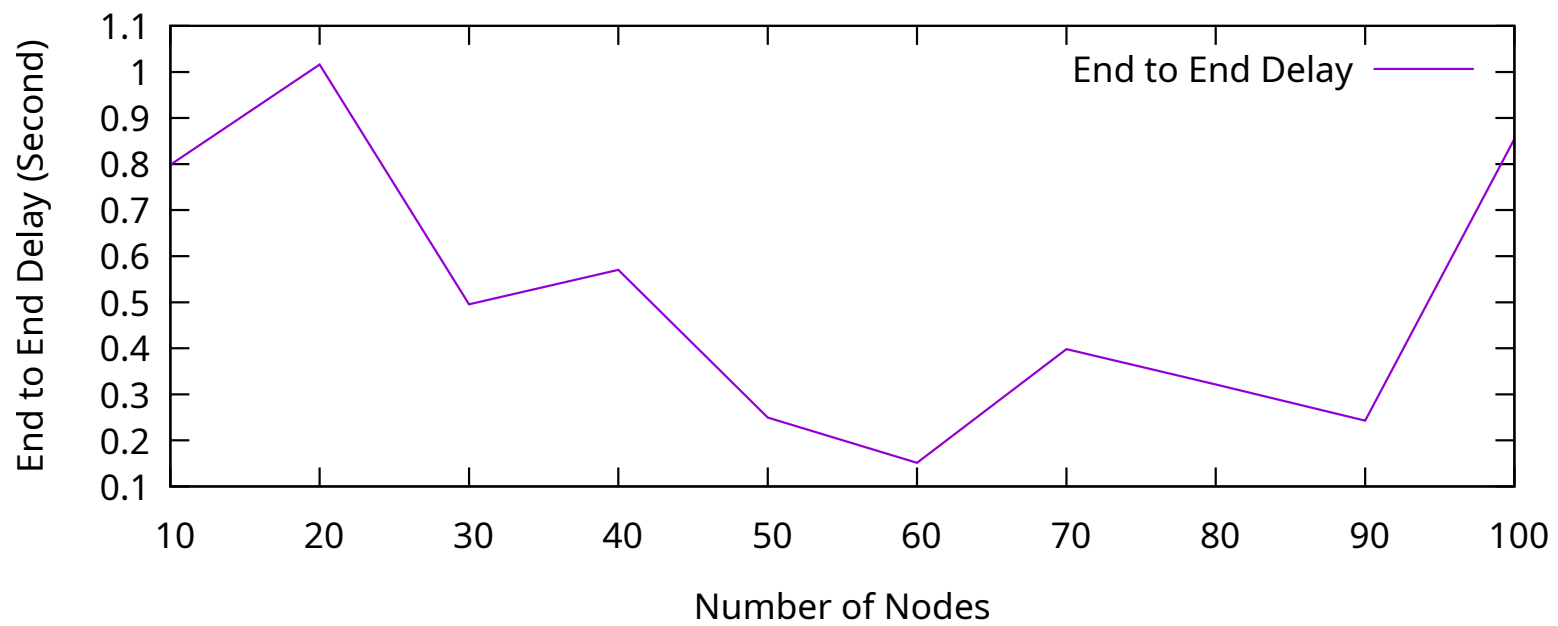
# TaskA Wireless LowRate (End-to-End Delay vs Num of Nodes)

Number of Flows = 100

Number of Packets Per Second = 100

Packet Size = 4096 byte

TX Range = 5





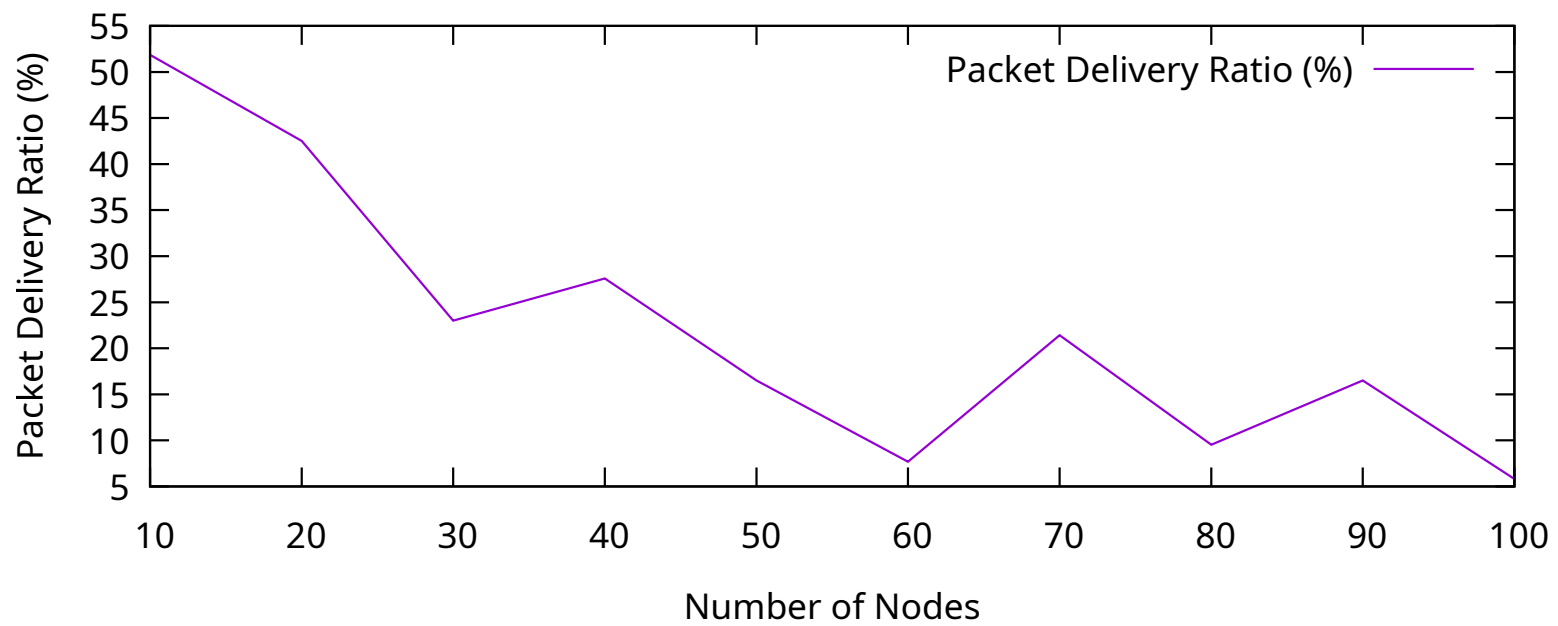
TaskA Wireless LowRate (Packet Delivery Ratio vs Num of Nodes)

Number of Flows = 100

Number of Packets Per Second = 100

Packet Size = 4096 byte

TX Range = 5



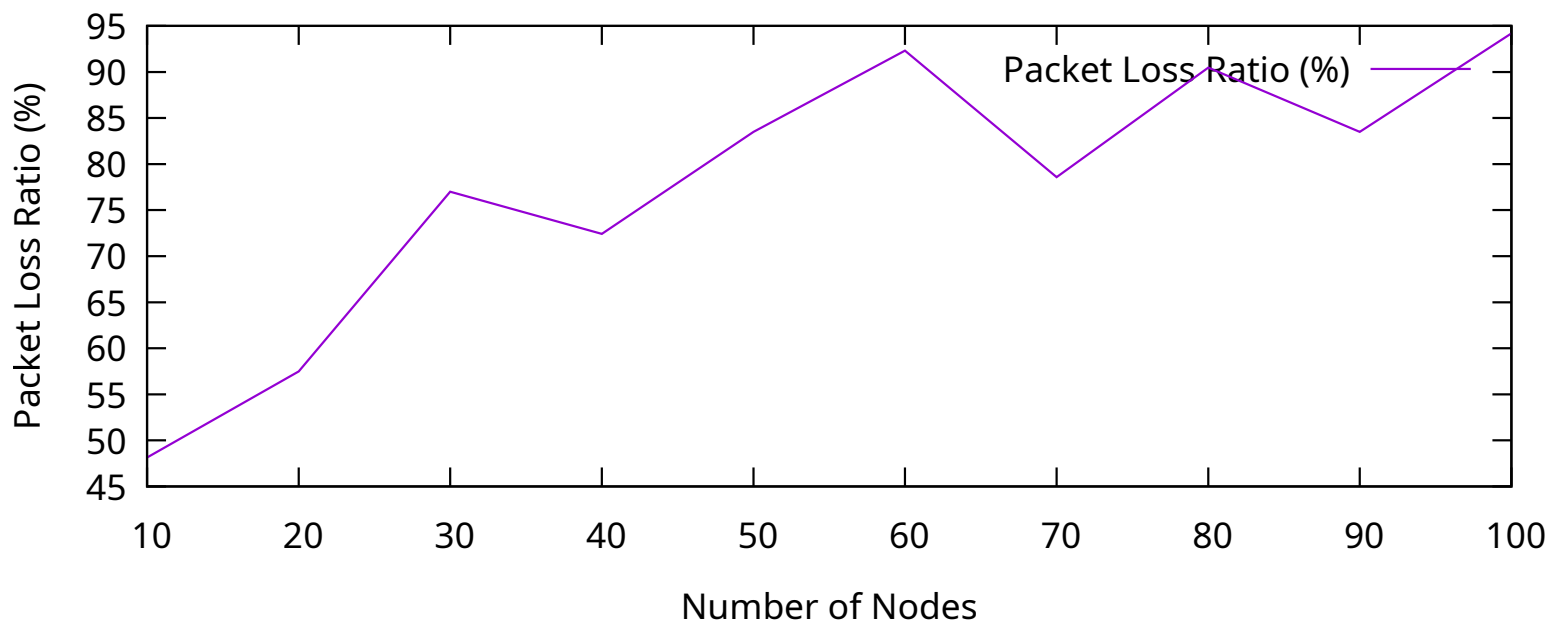
# TaskA Wireless LowRate (Packet Loss Ratio vs Num of Nodes)

Number of Flows = 100

Number of Packets Per Second = 100

Packet Size = 4096 byte

TX Range = 5



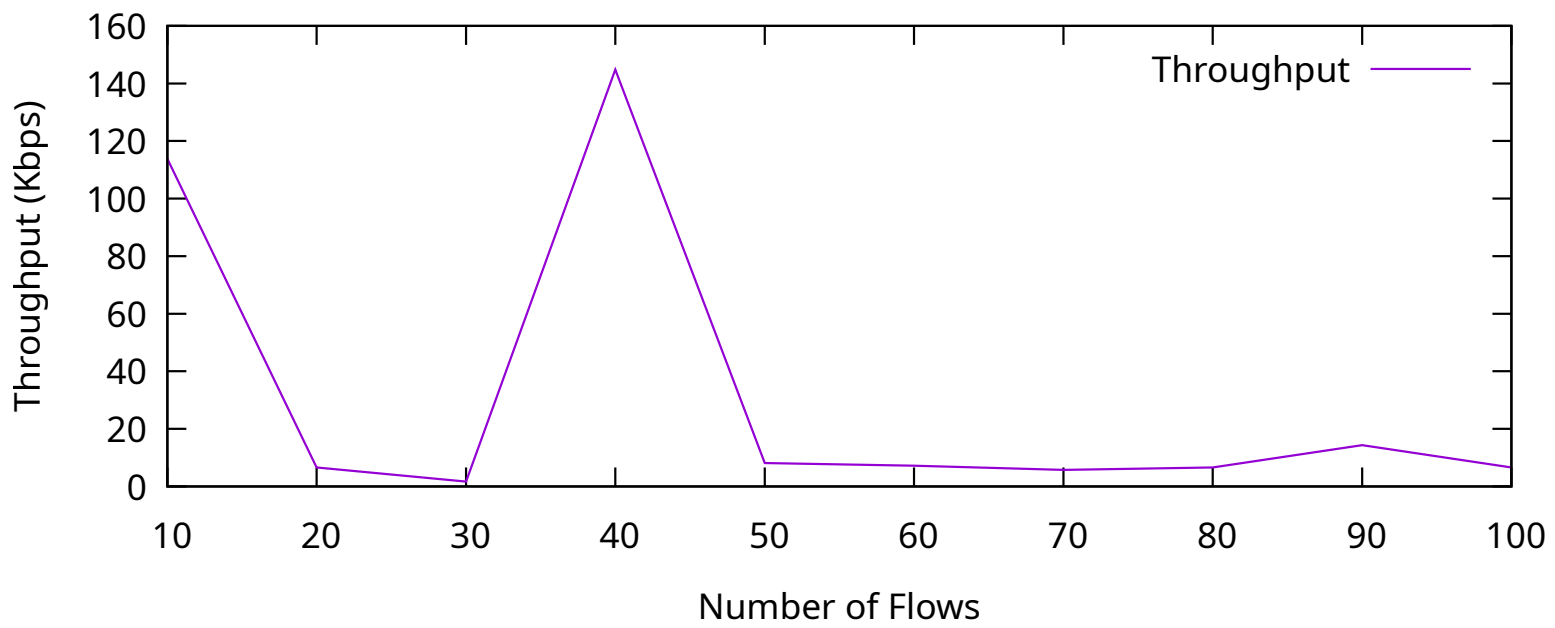
TaskA Wireless LowRate (Throughput vs Num of Flows)

Number of Nodes = 50

Number of Packets Per Second = 100

Packet Size = 4096 byte

TX Range = 5



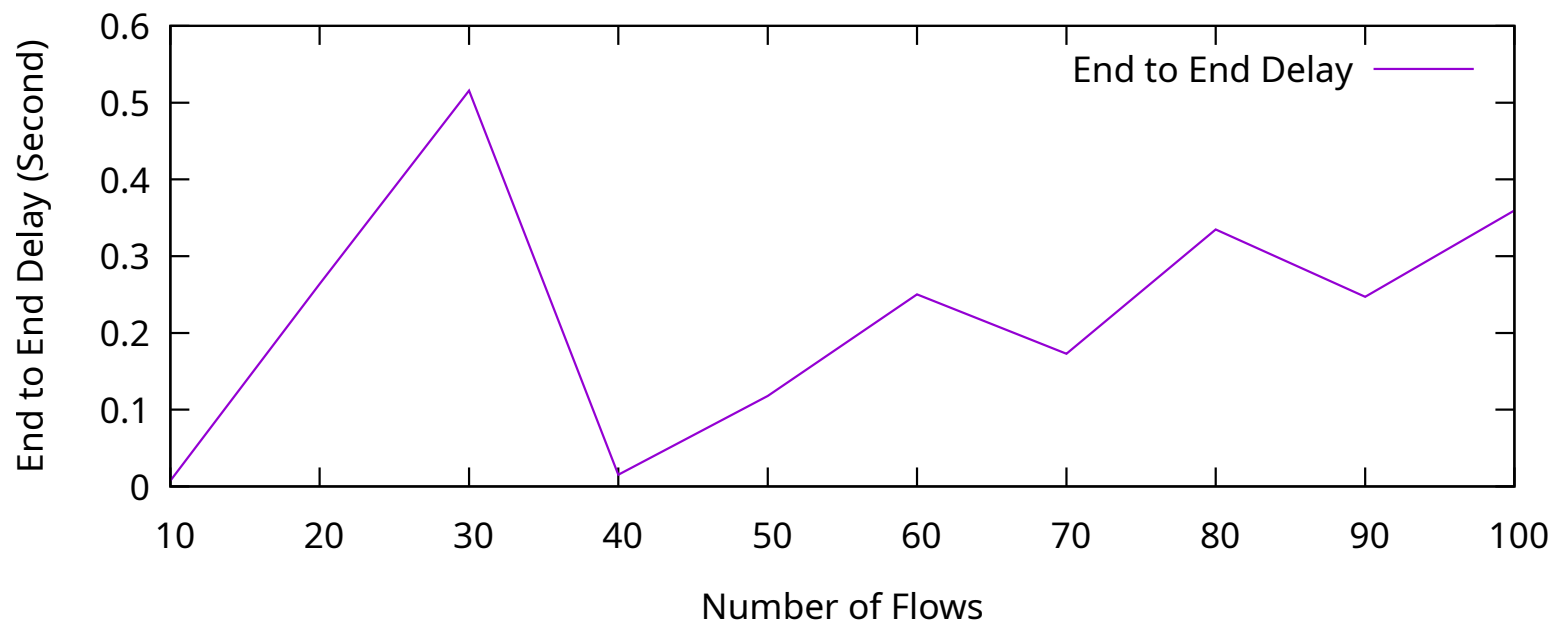
TaskA Wireless LowRate (End-to-End Delay vs Num of Flows)

Number of Nodes = 50

Number of Packets Per Second = 100

Packet Size = 4096 byte

TX Range = 5



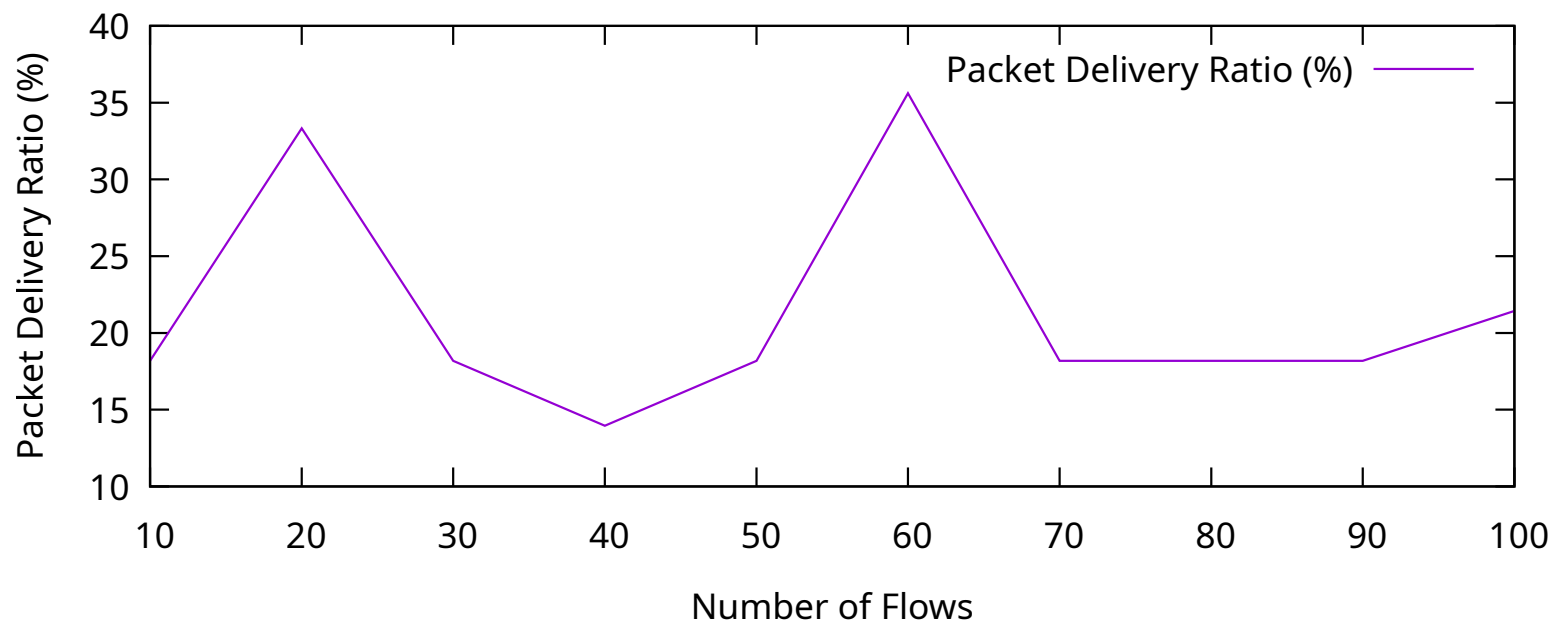
# TaskA Wireless LowRate (Packet Delivery Ratio vs Num of Flows)

Number of Nodes = 50

Number of Packets Per Second = 100

Packet Size = 4096 byte

TX Range = 5



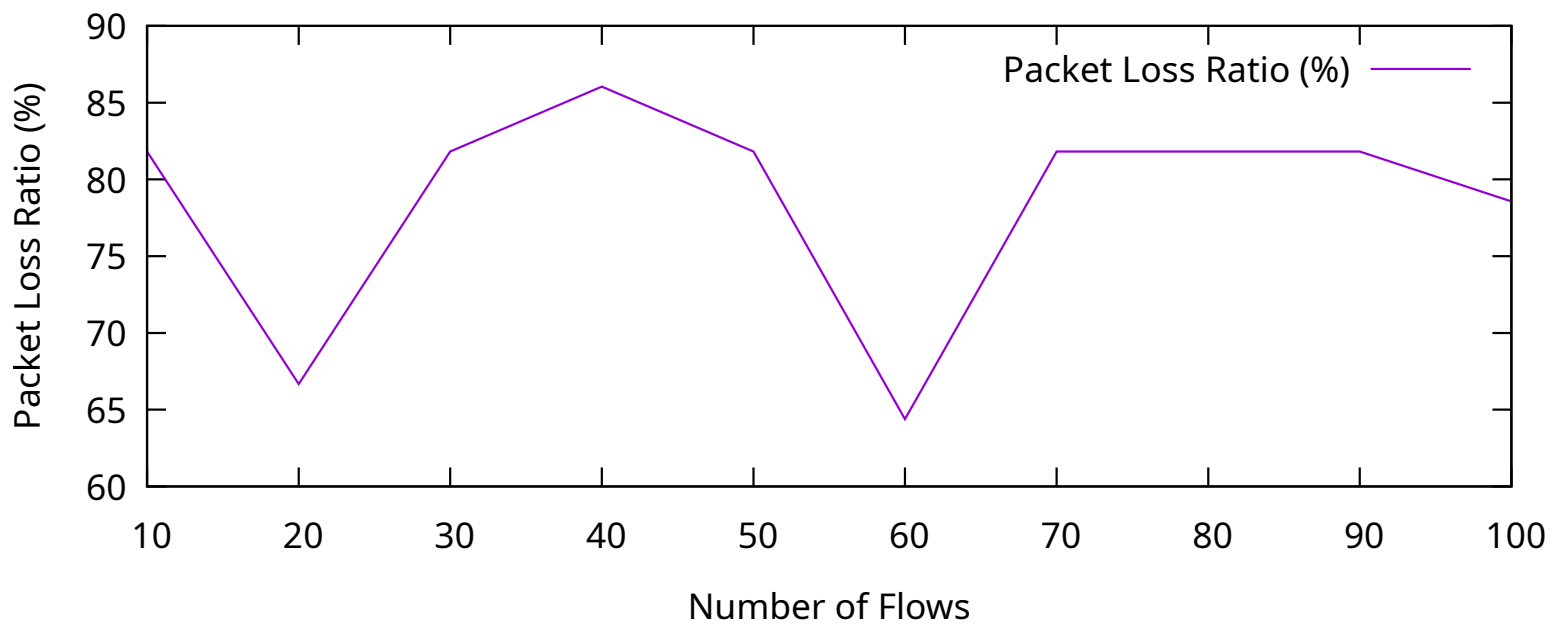
TaskA Wireless LowRate (Packet Loss Ratio vs Num of Flows)

Number of Nodes = 50

Number of Packets Per Second = 100

Packet Size = 4096 byte

TX Range = 5



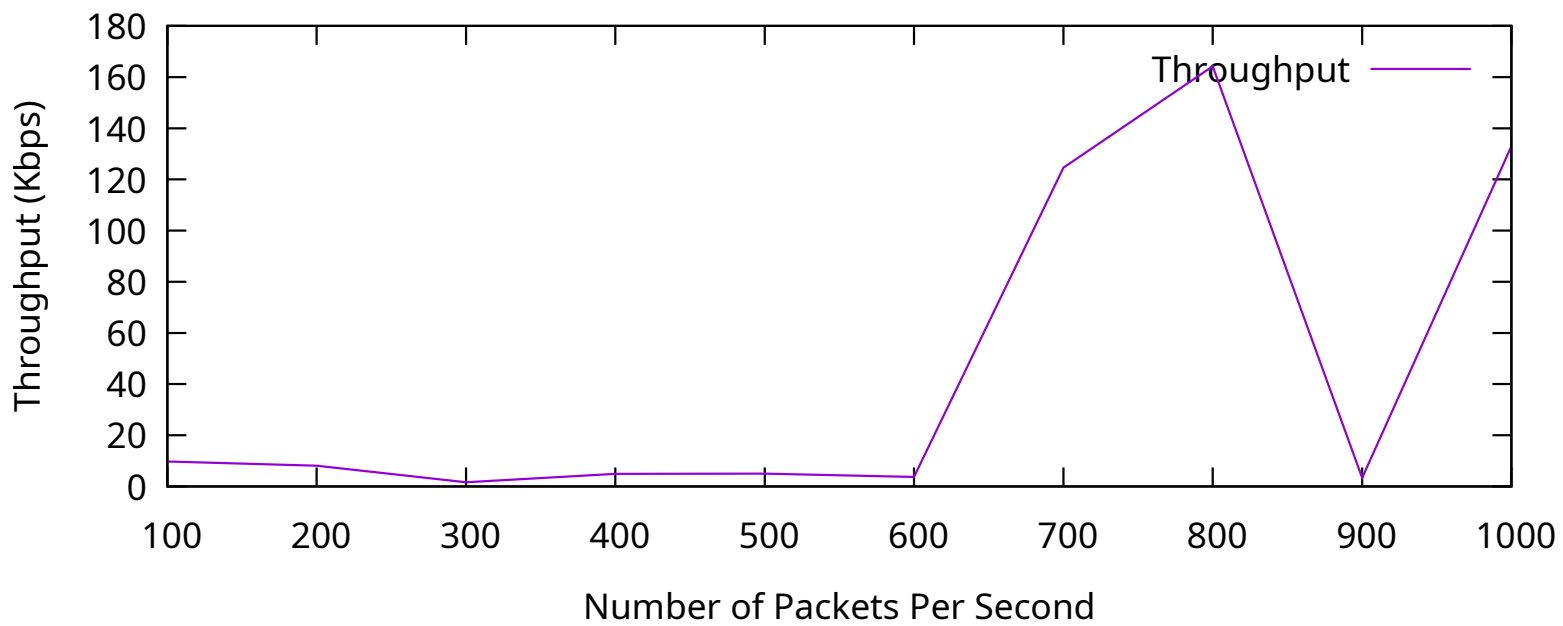
# TaskA Wireless LowRate (Throughput vs Num of Packets Per Second)

Number of Nodes = 50

Number of Flows = 50

Packet Size = 4096 byte

TX Range = 5



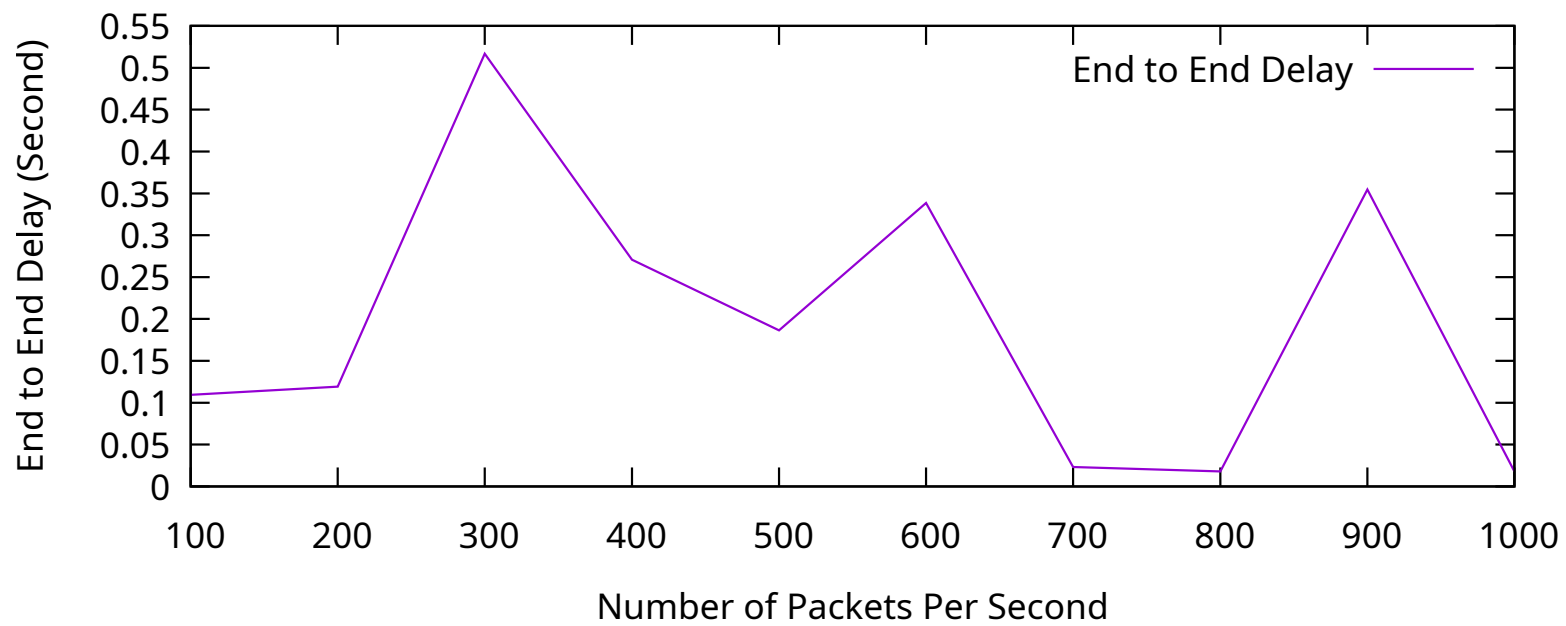
# TaskA Wireless LowRate (End-to-End Delay vs Num of Packets Per Second)

Number of Nodes = 50

Number of Flows = 50

Packet Size = 4096 byte

TX Range = 5





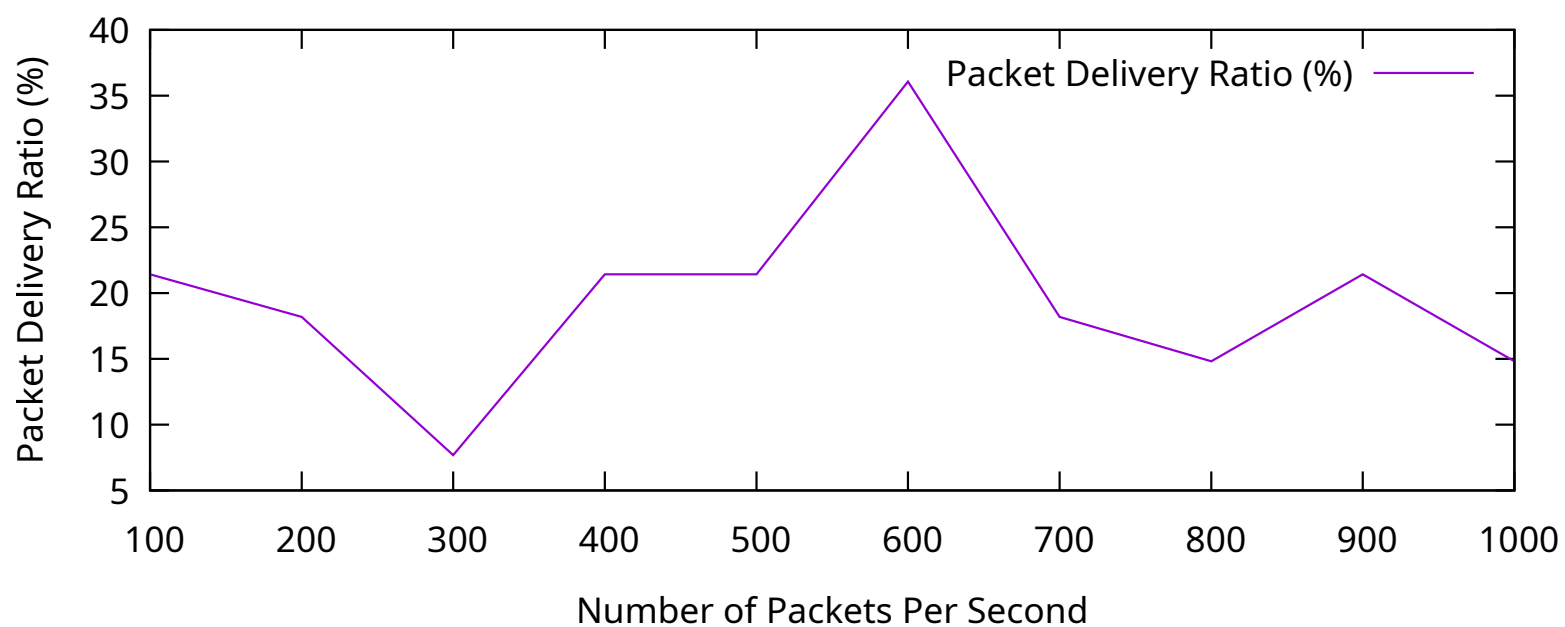
### TaskA Wireless LowRate (Packet Delivery Ratio vs Num of Packets Per Second)

Number of Nodes = 50

Number of Flows = 50

Packet Size = 4096 byte

TX Range = 5



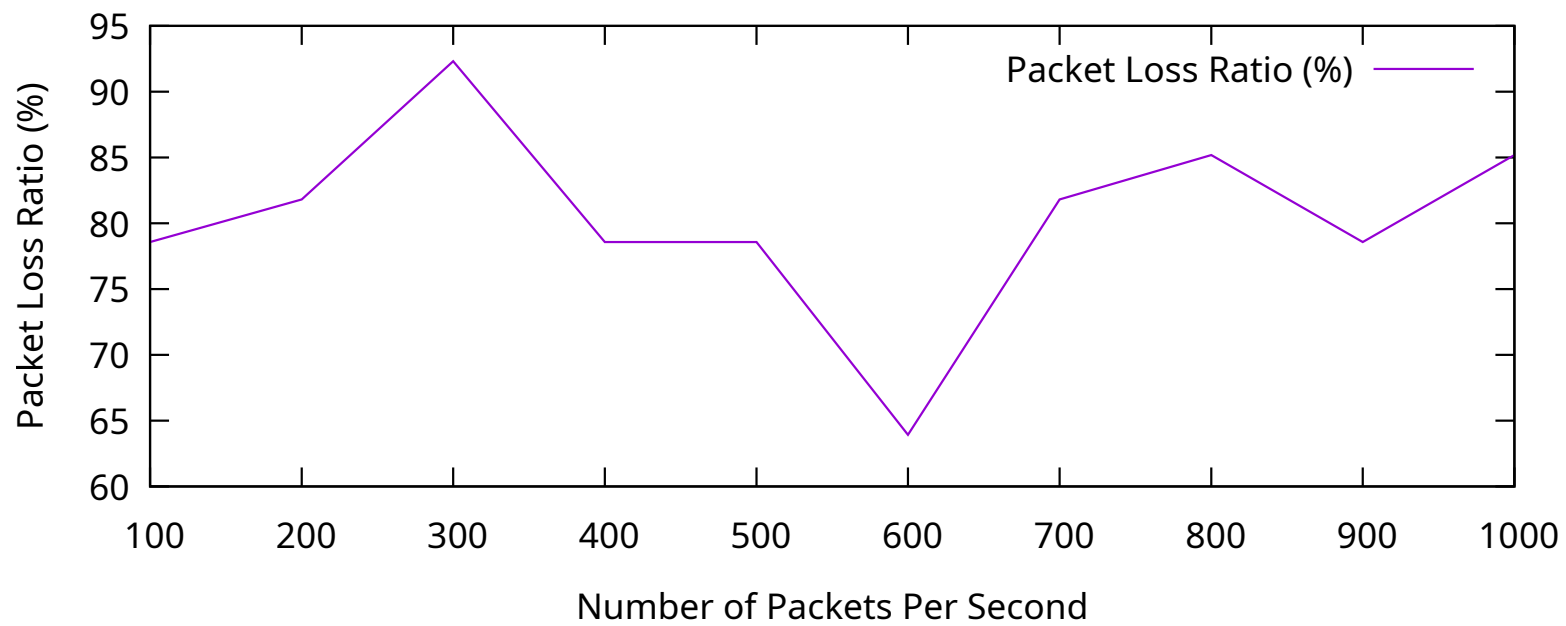
# TaskA Wireless LowRate (Packet Loss Ratio vs Num of Packets Per Second)

Number of Nodes = 50

Number of Flows = 50

Packet Size = 4096 byte

TX Range = 5



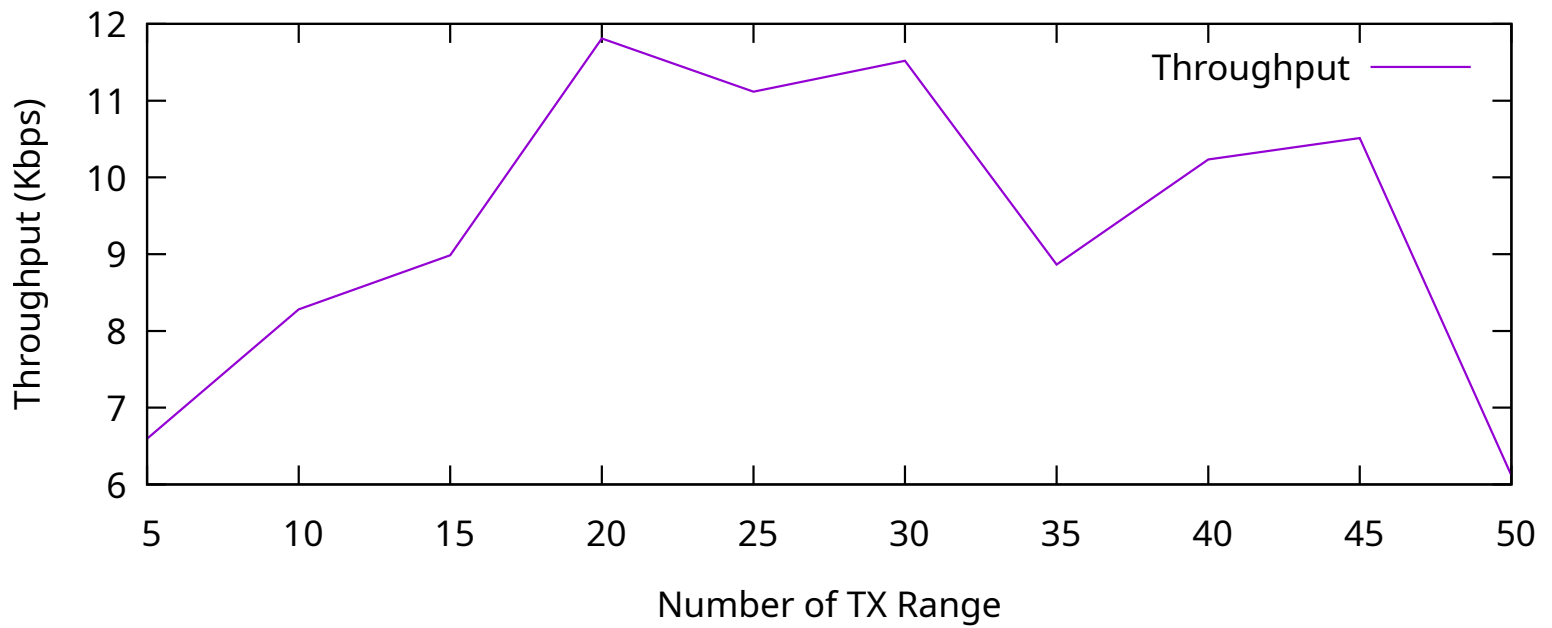
### TaskA Wireless LowRate (Throughput vs TX Range)

Number of Nodes = 50

Number of Flows = 50

Number of Packets Per Second = 100

Packet Size = 4096 byte



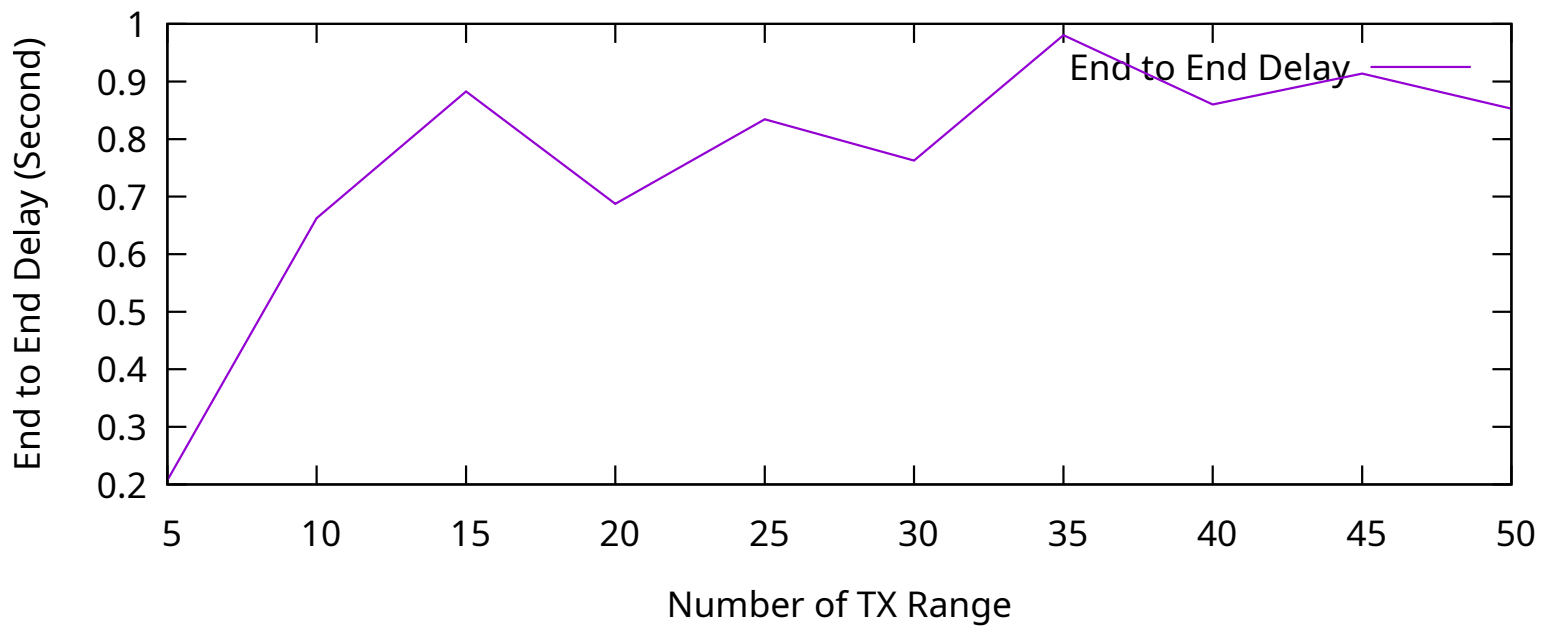
### TaskA Wireless LowRate (End-to-End Delay vs TX Range)

Number of Nodes = 50

Number of Flows = 50

Number of Packets Per Second = 100

Packet Size = 4096 byte



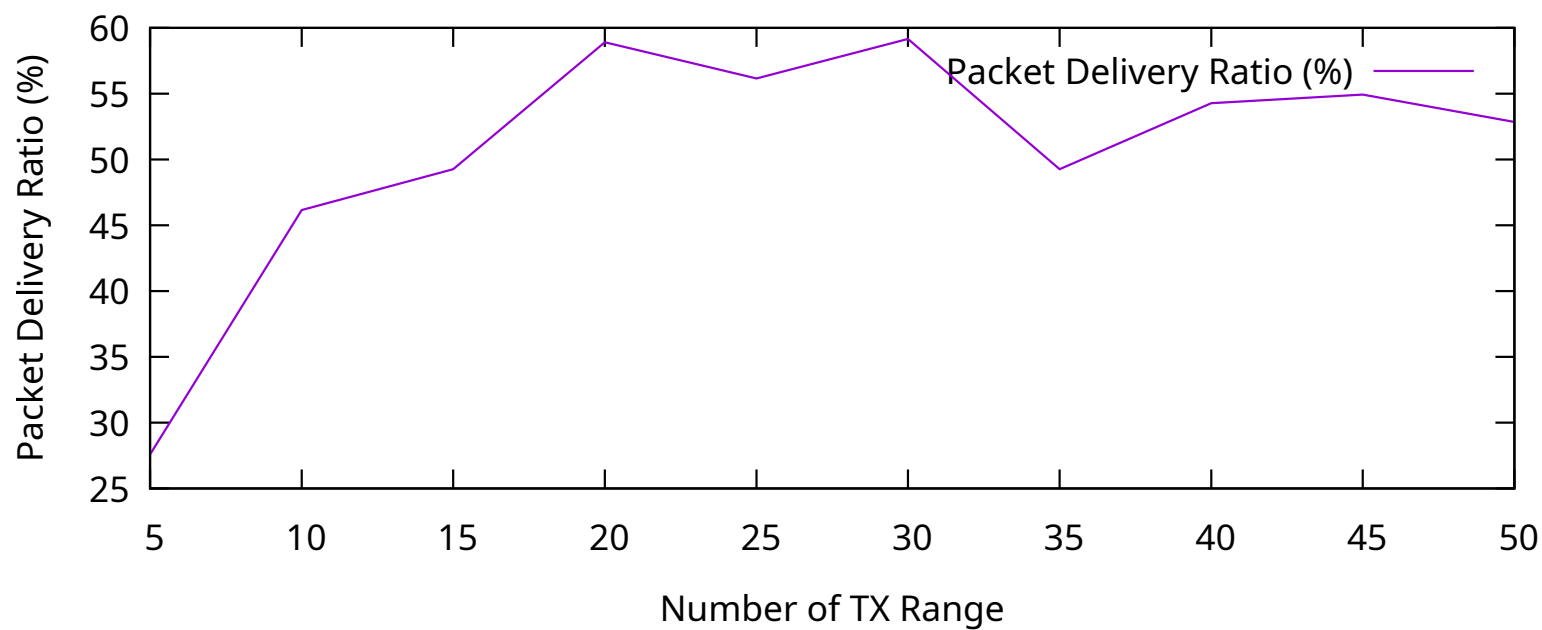
### TaskA Wireless LowRate (Packet Delivery Ratio vs TX Range)

Number of Nodes = 50

Number of Flows = 50

Number of Packets Per Second = 100

Packet Size = 4096 byte



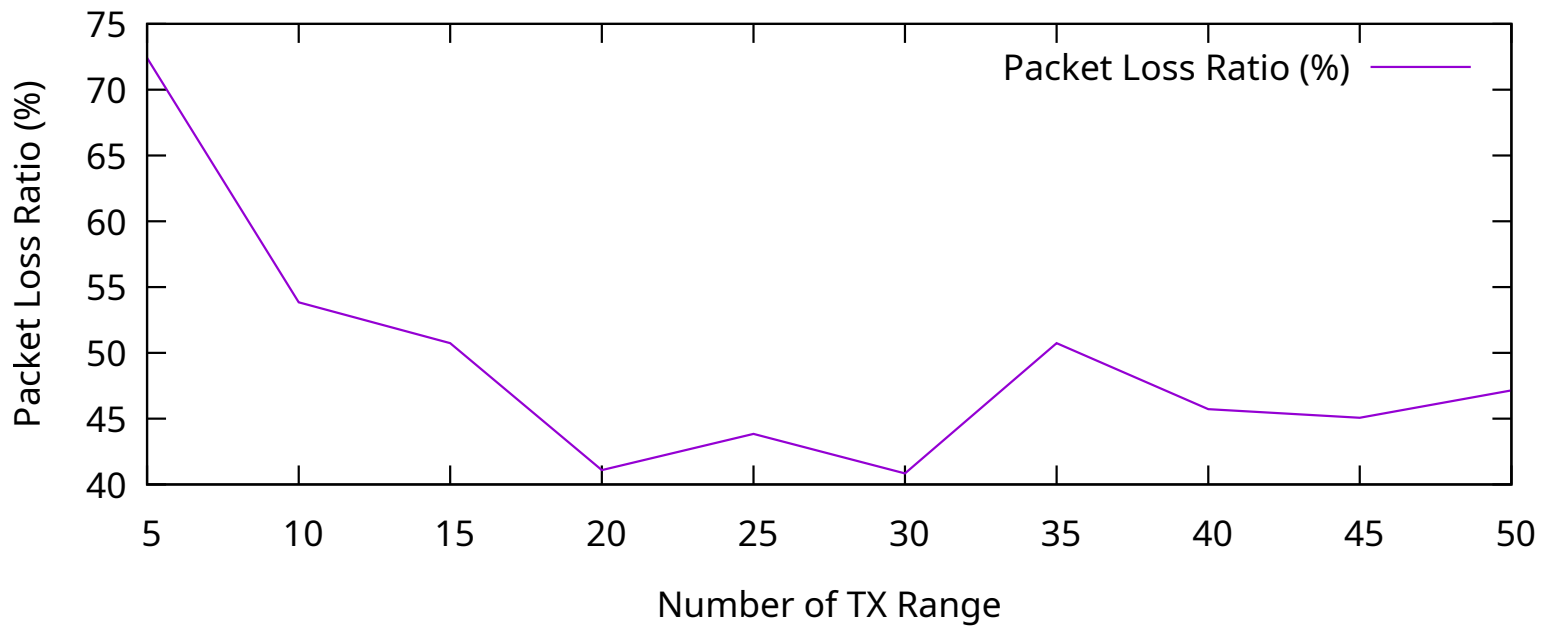
### TaskA Wireless LowRate (Packet Loss Ratio vs TX Range)

Number of Nodes = 50

Number of Flows = 50

Number of Packets Per Second = 100

Packet Size = 4096 byte



## 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>