ENSC 427: COMMUNICATION NETWORKS
Spring 2019

Simulating MWSN with Varying Parameters

https://loshau.wixsite.com/team-10

Younghoon Jee; (301177482); yjee@sfu.ca

Sam Swerhone; (301162949); sswerhon@sfu.ca

Liam O'Shaughnessy; (301283210); loshaugh@sfu.ca

Group 10

Abstract

Mobile ad-hoc networks (MANET) allow mobile devices to communicate with each other, affording common requirements like collision avoidance and distributed data acquisition. The key features of these non-trivial network designs are decentralization and forward routing. Mobile Wireless Sensor Networks (MWSN), are a subset of MANETs and was the focus of this project. A scenario involving an industrial warehouse with mobile robots collecting sensor data was implemented in ns-3. Four different routing protocols: AODV, OLSR, DSR, and DSDV were compared. When varying parameters such as the number of nodes, transmission power, and physical area it was discovered that DSDV performed the worst, while AODV and OLSR routing protocols were best suited for this scenario.

Table of Contents

1.0 Introduction

An ad hoc network is a decentralized type of network that allows peer-to-peer connections between end devices. Generally, these types of networks are wireless in nature and don't rely on infrastructure such as routers and gateways to establish connections between nodes. Wireless ad hoc networks are primarily categorized into three types: Mobile Ad hoc Networks (MANETs), Mobile Wireless Sensor Networks (MWSNs) and Wireless Mesh Networks. MANETs can be characterized by multi-hop wireless connectivity and dynamic network topologies [1]. There are many applications for these types of networks. These could include military applications such as using sensors to monitor a confined area for enemy intrusion. Other examples could include industrial environments where MANETs could be used to monitor the health of machinery as well as autonomous mobile systems such as drone clustering and self-driving cars. Figure 1 illustrates an example scenario for a military type of application:

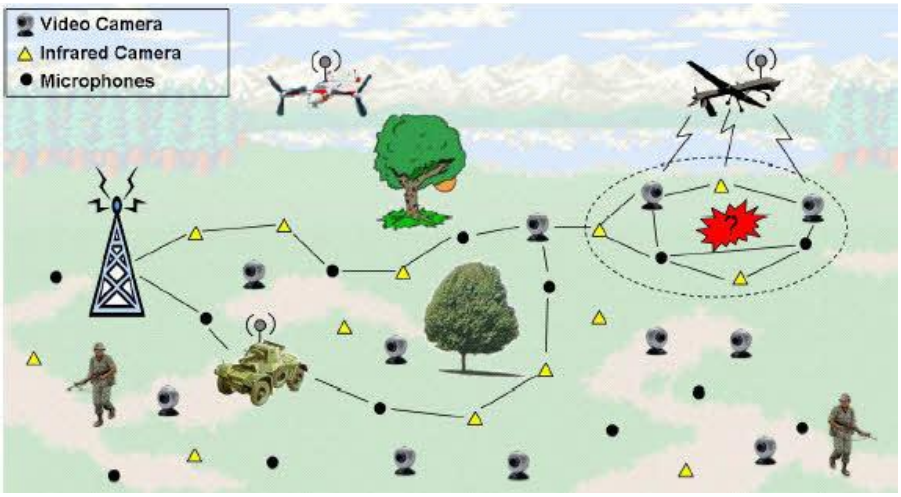

Figure 1 - Military Application Example

MANETs contain nodes that both send and receive packets while operating with limited power and processing capabilities. For this reason, routing protocols need to be performance oriented while also being efficient. These protocols will be discussed in a later section of this report.

MWSNs are considered a subset of MANETs in that they are also operating under tight power requirements. They usually consist of a microcontroller and various sensors for detecting light, pressure, and more [2]. One of the main differences between MWSNs and MANETs are that there is a sink node known as a "base station". This base-station collects data from all the other sensor nodes in the network and is responsible for processing the data. A typical topology of a MWSN can be represented in the form of an undirected graph that is not fully connected and can be seen below in Figure 2.

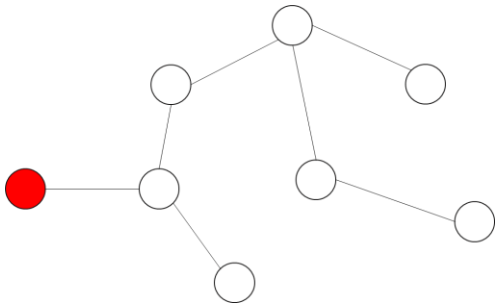

Figure 2 - Example MWSN Topology

The red node seen in Figure 2 represents a base-station that collects data from the white sensor nodes. Our project will focus on MWSNs and how they perform in different scenarios under varying parameters.

1.1 Related Work

In V. Jayalakshmi and T. Razak's study on issues and challenges in mobile ad hoc networks, they outline the various routing protocols and several challenges and issues of ad hoc networking [1]. Proactive routing protocols attempt to maintain consistent and up-to-date routing information from each node to every other node in the network. Example protocols of this include "Destination-Sequence Distance Vector (DSDV) and "Optimized Link State Routing" (OLSR). Reactive routing protocols establish routes only when needed and do so by flooding a network with a route request packet (RREQ). Some reactive routing protocols include "Ad hoc On-Demand Distance Vector (AODV) and "Dynamic Source Routing" (DSR). Hybrid routing protocols attempt to combine the best features of proactive and reactive algorithms by dividing network into zones. Some well known hybrid routing protocols include "Zone-Based Routing Protocol" (ZRP) and "Sharp Hybrid Adaptive Routing Protocol" (SHARP). Some issues of mobile ad hoc networks (MANET) include multicasting, multiple routes, distributed operation, physical security, and unidirectional, and limited power. This paper established a basis for the protocols we will be looking at in our MWSN simulation.

In "Performance Analysis of Routing Protocols for Wireless Ad-Hoc Networks", L. Trajković and S. Lally explore different routing protocols used in an ad-hoc network for Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) traffic [3]. Ad-hoc networks are decentralized networks where hosts can establish connections to each other, without base stations (BS) or an access point (AP). The paper focuses on MANETs for simulation environments as they are more flexible. Due to the self-configuring nature of the network and growth of mobile hosts such as smartphones and laptops, MANETs have been a popular research topic for the past decade. The paper describes three different ad-hoc routing protocols used for OPNET simulations: Ad-Hoc On-Demand Distance Vector (AODV), Dynamic Source Routing (DSR) and Optimized Link State Routing (OLSR). Moreover, the authors extensively describe performances of parameters such as "route discovery time, end-to-end delay, download response time, and routing traffic overhead" for each routing protocol simulation.

In A. Winfield's conference paper "Distributed Sensing and Data Collection Via Broken Ad Hoc Wireless Connected Networks of Mobile Robots", he discusses a scenario in which several mobile robots are required to disperse in a bounded region to take sensor readings and communicate back to a single collection point [4]. A characteristic of ad hoc networks is that it is virtually impossible for each node to

know the entire network topology, which means traditional wide area network routing protocols can't be used. In this paper, the proposed high-level algorithm is as follows:

- At a certain time interval, every robot samples new sensory data
- Every robot then broadcasts its data to each of its neighbours
- In turn each neighbour broadcasts any received data and its own data to its neighbours where it's then stored in a buffer
- Each time a robot detects a new robot has come within its wireless range, then all data within its buffer is broadcast to the new neighbour and to its neighbours, and vice versa
- Each robot's data is marked with that robot's ID (IP address) and when robot 0 (edge of network connected to base station) has received data from every robot, then data collection for that time period is complete

We found the overall scenario to be inspirational for our own, however, we did not use the proposed algorithm in this paper.

In "Mobile Wireless Sensor Networks" written by Dr. V Ramasamy elaborates on phenomena that energy efficient MWSNs design applications can bring about [2]. Dr. Ramasamy starts off with the hurdles that current MWSNs face such as hardware limitations and harsh environments. To overcome and remedy these issues, he suggests that a hybrid network topology is the ideal model for a large-scale mobile sensor network. He pinpoints the reason of why mobility models need to be considered for real world applications. Every living organism is not stationary. As they are moving, observation and analysis of mobility is so crucial in real world. Ramasamy concludes his paper with real-life design challenges such as hardware architecture, algorithms and routing protocols to account for natural characteristics of mobility nodes: randomness.

2.0 Environment

The chosen operating system for this project is Ubuntu and the network simulation tool used is ns-3. We used ns-3 to simulate Mobile Wireless Sensor Networks (MWSNs) as they support various network protocols such as AODV, DSVD, DSR, and OLSR. We chose ns-3 for our main simulation tool as it is free (for education version) and provides more flexibility than some other tools available. We also utilized the Python scripting language as well as a numerical library called "numpy".

3.0 Routing Protocols for Simulation

A routing protocol is a set of rules (standards) that determine paths from source to destination for packet transmission. Ad hoc routing protocols are mainly categorized into three groups: table driven, on-demand driven, and hybrid protocols. Table driven, also known as proactive, protocols periodically update routing tables for each node during packet transmission [3]. The benefit of proactive protocols is that the protocols allow every node to maintain up to date routing tables and topology information. However, one drawback is that unnecessary data is used to maintain routing tables for idle nodes. On-demand driven, also known as reactive, protocols determine a packet route on demand by flooding the network. In contrast to proactive protocols, nodes using reactive protocol only initiate a route discovery process when required. Establishing path connection on demand can result in low overhead, but high latency. Hybrid protocols are the combinations of both table-driven and on-demand protocols. Figure 2 shows ad-hoc routing protocols categorized into the three groups. We used Ad-hoc On-Demand Distance Vector (AODV), Destination Sequence Distance Vector (DSDV), Dynamic Source Routing (DSR) and Optimized Link State Routing (OLSR) for our simulation.

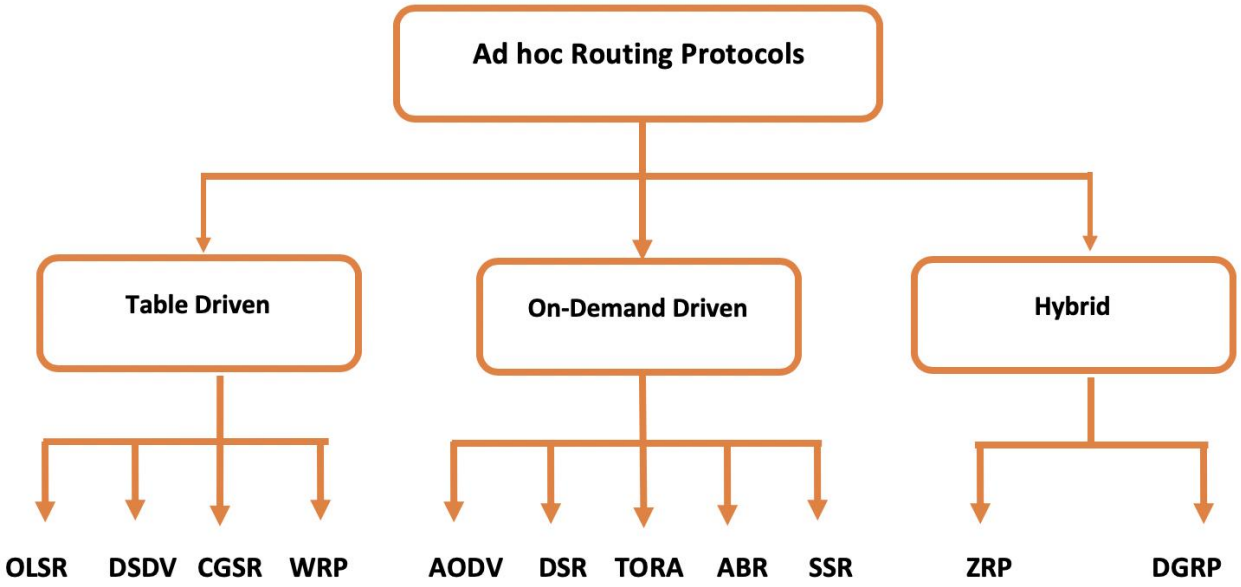

Figure 2 – Ad-hoc Routing Protocol [6]

3.1 Ad-hoc On-Demand Distance Vector (AODV)

AODV is a reactive protocol that establishes routes on demand between nodes. The protocol prevents "Counting to infinity" problem where nodes get updated in a loop [3]. Furthermore, each router has a certain lifetime where connections get discarded once a route expires [7]. There are four basic message types for AODV: Route Request Packet (RREQ), Route Reply (RREP), Route Error (RERR) and HELLO. RREQ is the request message carried by a previous node to form a route to next node. During the route

discovery process, a source triggers an RREQ message to its neighbors to validate the route for packet transmission to destination. Then, RREP from nodes gets unicasted back to original node that generated RREQ. During the course of packet transmission, if an error is detected, RERR message discards the route. Lastly, HELLO message ensures all neighboring nodes to update latest route in case there is a new change in route. Hence, AODV is optimal when there are moving nodes in a network.

Table 1 - AODV Advantages and Disadvantages

Advantage	Disadvantage
Unicast and multicast transmission	High processing demand
Flexible with node movements	Transmission delay due to expired route
Smaller bandwidth for path advertisements	Lengthy time to build routing table

3.2 Destination Sequence Distance Vector (DSDV)

DSDV is the enhanced version of the Bellman-Ford algorithm. It solves the generic problem, count to infinity, that Bellman-Ford algorithm has, ensuring loop-free paths. Each node maintains one routing table for route discovery. The routing protocol employs two update mechanisms to update routing tables for each node for packet transmission: periodic update and trigger update [8]. For periodic update, routing tables in each node get updated periodically at certain time interval, while trigger update broadcasts an update when there is a change in routing table. As updates need to be propagated to each node within the network, the routing protocol is suitable when there are a small number of nodes.

Table 2 - DSDV Advantages vs Disadvantages

Advantage	Disadvantage
Guaranteed loop-free path	Excessive delay as nodes increase
Short path set-up process time	Large power consumption due to frequent routing table updates
Optimal for network with few nodes	

3.3 Dynamic Source Routing (DSR)

DSR utilizes source routing to transmit packets, rather than using routing tables [9]. Each routed packet knows a pre-defined path that it needs to traverse. As the path is determined from the source node, all intermediate nodes do not need to have the latest updates to their routing table. DSR is mainly comprised of two phases when delivering packets from source to destination: route discovery and route maintenance [3]. During the discovery phase, the source determines the most efficient and feasible route by broadcasting a Route Request Packet (RREQ). Once source routing is established and the packet successfully arrives to its destination, it sends a Route Reply (RREP) to the source. Maintenance phase is

initiated when Route Error (RERR) is triggered to inform source about broken route during packet transmission.

Table 3 - DSR Advantages and Disadvantages

Advantage	Disadvantage
Self-organizing without network infrastructure	Poor performance when node movements increase
No need for up-to-date routing information	Not scalable

3.4 Optimized Link State Routing (OLSR)

OLSR is a table-driven routing protocol that exchanges link state information between nodes. The protocol uses “hello” and topology control messages to broadcast state changes to all nodes in the network [10]. Since every node needs to receive a link state update periodically, topology can suffer from tremendous overhead. Yet, a large amount of overhead can be reduced by using Multi Point Relay (MPR) to relay routing messages.

Table 4 - OLSR Advantages and Disadvantages

Advantage	Disadvantage
Use hello and topology control messages	Overhead increases when nodes increase
Reduce control traffic overhead	Long time to find broken link
Small end-to-end delay	Large processing power to find alternate transmission path

4.0 Scenario

The simulation scenario is an industrial warehouse with many worker robots that will be taking sensor reading and reporting back to a base station. These robots are mobile with a total of N number of mobile robots. An M number of data-generating robots and N-M non data-generating robots. The robots not generating data will be used solely for packet routing. All the worker robots are constrained to move within a physical area represented as the warehouse. The mobility of these robots is random their speed is constant. In this scenario, the data-generating nodes will be constantly sending packets to the base station at a specific transmission rate. The routing algorithms used in this simulation are AODV, DSVD, DSR, and OLSR as described in the previous section. Figure 3 illustrates this scenario.

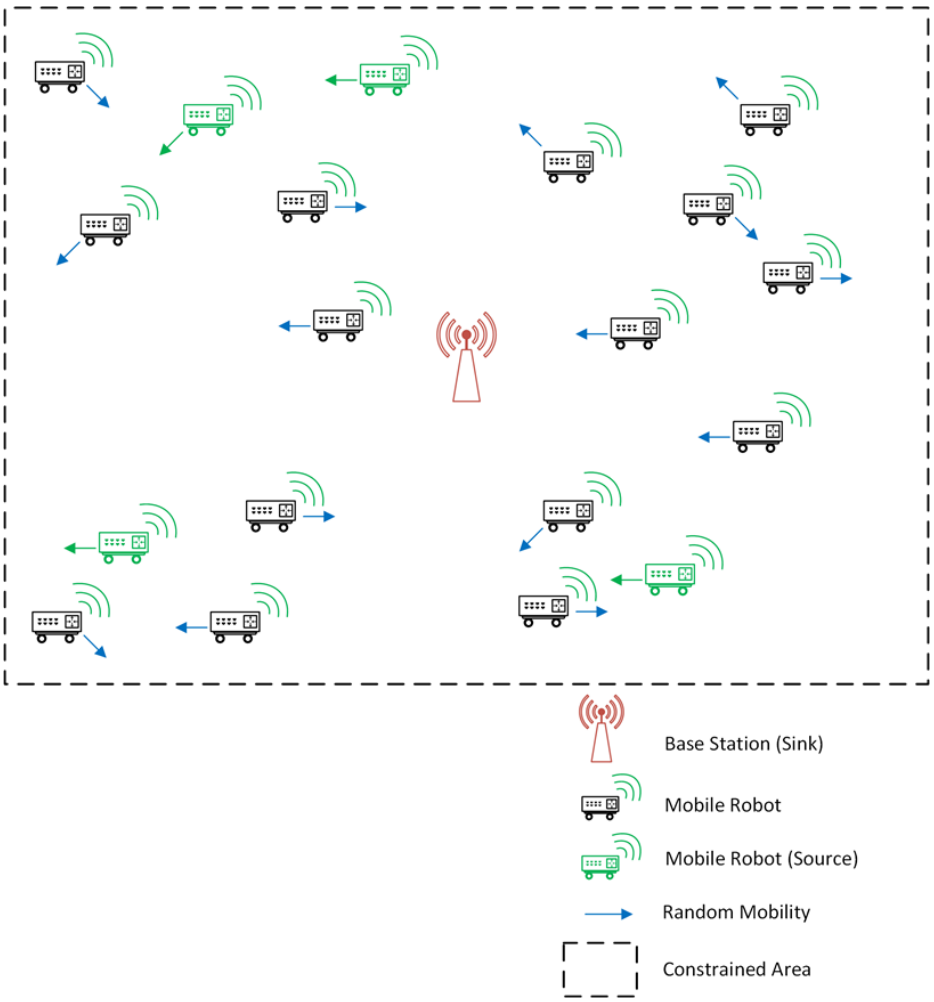

Figure 3 - Industrial Warehouse with Worker Robots

The goal of this scenario is to observe the throughput of the network for each different routing protocol while varying parameters. The parameters varied will be covered in the next section.

5.0 Implementation

This section will describe all the details required to implement the scenario listed above. Included in this section is a description of the network simulation code as well as the wrapper script developed in Python.

5.1 ns-3 Simulation

To develop this simulation, we adapted an open-source ns-3 example written by Justin Rohrer. We modified this code to add a single base station node for collecting data as well as adjusting the number of sources sending data. We added command line arguments to modify the physical area, node transmission power, traffic rate, the number of nodes in the scenario, and the routing protocol to be used. The total simulated network time is 150 seconds for each individual simulation. The list of all simulation parameters that have been held constant can be seen in Table 5. The simulation code can be seen in Appendix A.

Table 5 - Constant Simulation Parameters

Simulation Time	150 s
Data Generating Robots	10
Traffic Type	UDP
Node Movement Speed	20 m/s
WiFi Protocol	802.11b
Propagation Delay Model	Constant
Propagation Loss Model	FriisPropagationLossModel

The simulation generates an output CSV file where each row contains the simulation second, number of packets received during that second, and the receive rate at the base station node. This CSV file will be used by the Python script for extra processing which will be covered in a later section.

5.2 Parameters of Interest

The parameters that were varied during simulation include the physical area of the warehouse, the number of mobile nodes in the warehouse, the transmission power of each node, and the traffic sending rate of each node. For each simulation, one of the parameters would be changed and the others held to their default value. The default values of each parameter can be seen in Table 6.

Table 6 - Default Value of Parameters

Physical Area (m^2)	300 x 1500
Number of Nodes	50
Transmission Power (dBm)	7.5
Traffic Sending Rate (bps)	2048

The range of values for each parameter of interested are listed in Table 7.

Table 7 - List of Parameter Values

Physical Area (m^2)	100x100, 300x300, 500x500, 700x700, 900x900, 1100x1100, 1300x1300, 1500x1500, 1700x1700, 1900x1900, 2100x2100, 2300x2300, 2500x2500
Number of Nodes	20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80
Transmission Power (dBm)	2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30
Traffic Sending Rate (bps)	100, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000
Routing Protocol	OLSR, AODV, DSDV, DSR

5.3 Python Script

To compare the different algorithms average throughput, we developed a wrapper script that could run each simulation with a specific set of parameters. The script has five main components:

- *Comparison of routing protocols when varying the **physical area***
- *Comparison of routing protocols when varying the **number of nodes***
- *Comparison of routing protocols when varying the **transmission power***
- *Comparison of routing protocols when varying the **traffic sending rate***
- *Comparison of routing protocols when varying the **physical area with a low transmission power***

Each comparison will call the ns-3 simulation code and pass the varied parameter along with the other default parameters listed in Table 5. It will run this exact simulation four times using each of the routing protocols. After each simulation, the generated the output CSV file was used to compute the average throughput with the "numpy" Python module. The full script is listed in Appendix B and a general process flow for the full system can be seen below in Figure 4.

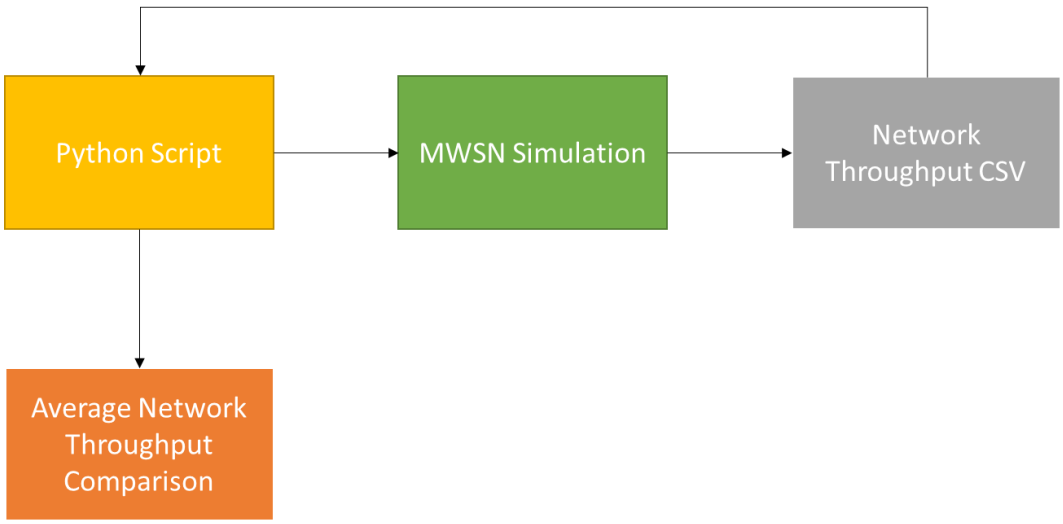

Figure 4 - System Overview

6.0 Results

In this section, we will showcase the results for five different comparisons as described in the above section. Each will show the average throughput versus the four parameters listed in Table 6. The relative performance of each routing protocol will be illustrated and discussed.

6.1 Physical Area Comparison

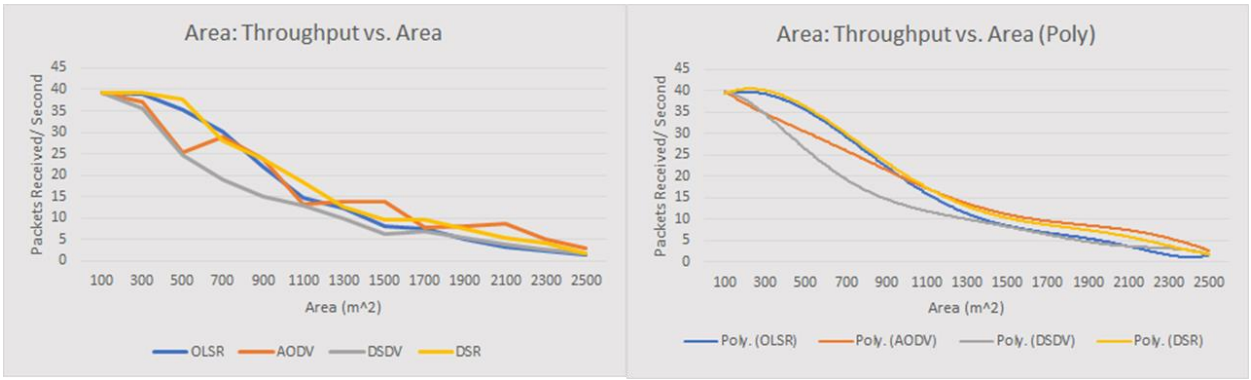

Figure 5 - Area vs Throughput

As you can see in Figure 5, the network throughput is decreasing as the physical area of the warehouse increases. This is expected as the transmission power has been held constant at 7.5 dBm. All protocols perform quite similar in this scenario, however, DSDV seems to have the lowest throughput for any given area. DSR and AODV are performing marginally better than OLSR.

6.2 Number of Nodes Comparison

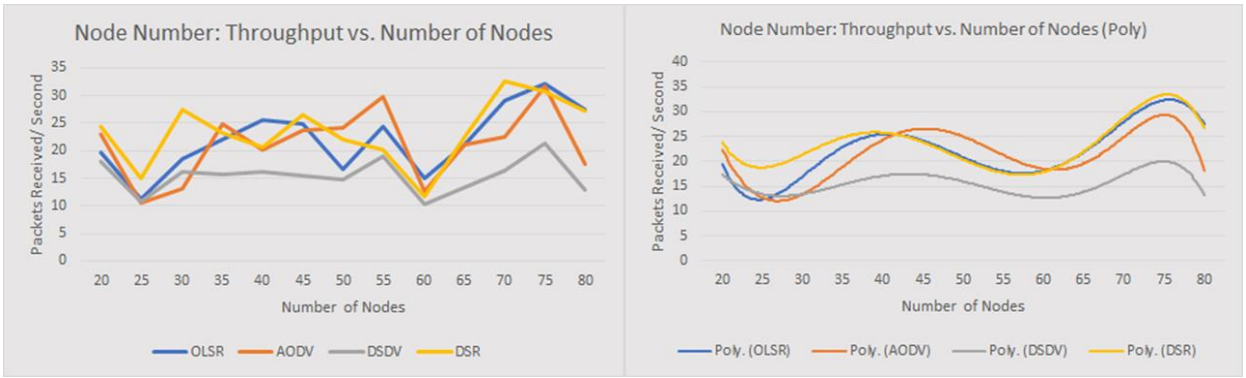

Figure 6 - Number of Nodes vs Throughput

The number of nodes does not seem to have a direct relationship with the network throughput in this scenario. As you can see in Figure 6, there seem to be local minima and maxima in each of the routing protocols. This is something that can be taken into consideration by a system designer. There seems to be a necessary balance between node density and overhead.

6.3 Transmission Power Comparison

Figure 7 - Transmission Power vs Throughput

The transmission power clearly has a direct effect on the network throughput as should be expected. With the area being held constant, increasing the transmission power results in an increase in network throughput. However, this increase does saturate to a certain point around 40 packets/second. DSR reaches this saturation point faster than all other algorithms and DSDV once again takes the longest.

6.4 Traffic Generation Comparison

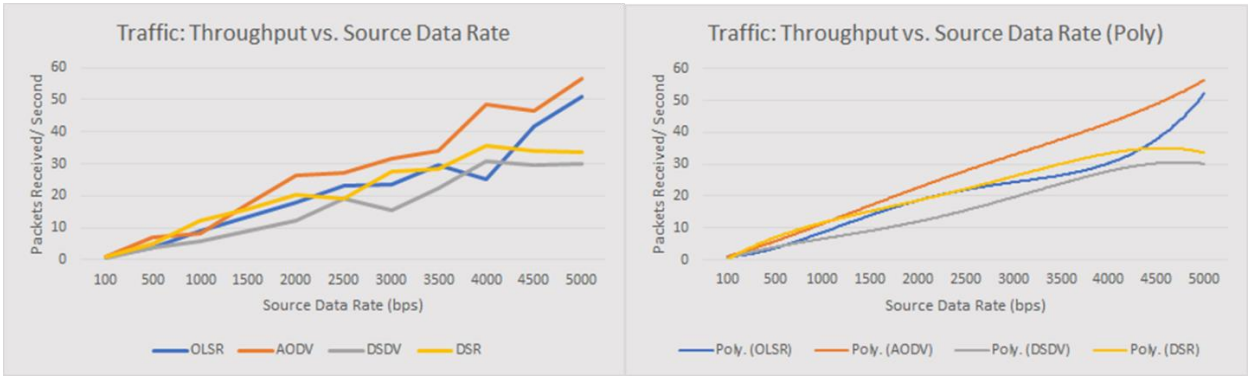

Figure 8 - Traffic vs Throughput

As you can see above, an increase in the source data rate results in an increase in throughput for all routing protocols. AODV and OLSR seem to adapt to high traffic situations much better than DSDV and DSR. DSDV and DSR seem to level off around 30 and 35, respectively. Further simulations with sending rate higher than 5000 bps could possibly let us see a similar saturation in AODV and OLSR.

6.5 Area with Half Transmission Power Comparison

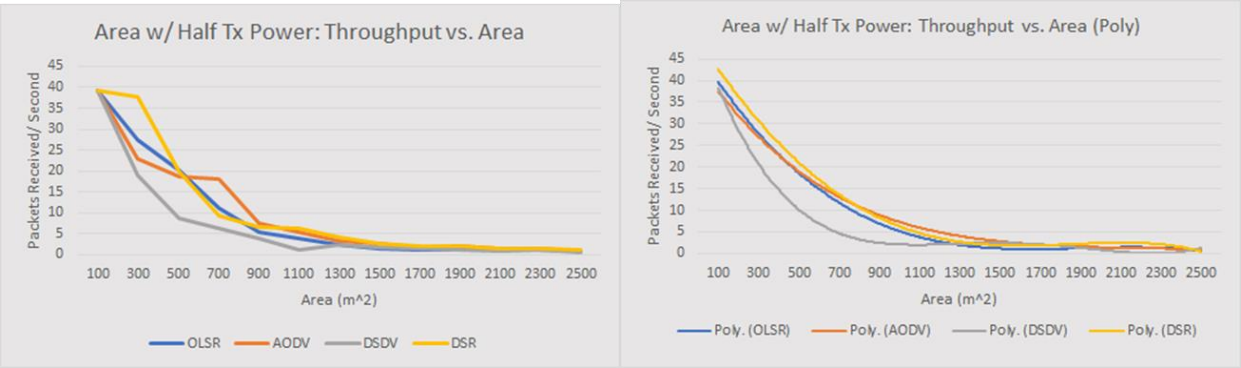

Figure 9 - Area and Half Transmission Power vs Throughput

As you can see in this comparison, the physical area is increasing, but instead of using the default transmission power, it is instead half the default transmission power. A similar result to Figure 5 is seen, however the throughput decreases faster. This is expected with lower transmission power. Once again, all algorithms perform similarly except for DSDV which performs the worst out of the four.

7.0 Conclusion

The overall performance of AODV, OLSR, DSDV, and DSR routing protocols perform quite similar in this MWSN network. What can be seen from the results is that DSDV had the overall worst performance. This is quite noticeable in the high traffic and high number of node conditions pictured in Figure 8 and Figure 6. In contrast, AODV and OLSR excelled in these high traffic and high node conditions. DSR performed well in the high node conditions but could not handle high traffic conditions as you can see in Figure 8.

In general, what we found was that MWSN networks can generally increase the transmission power of a robot sensor node to overcome large area. However, MWSN sensor nodes usually have very constrained power requirements that need to be considered when increasing the transmission power. The transmission power also shows a clear knee point where an increase in power showed no improvement in network throughput. The number of nodes did not show a clear relationship between network throughput. This may be attributed to a necessary balance between node density and the amount of overhead it introduces to the network.

Some challenges for this project included utilizing the ns-3 tool to its full capability. Ns-3 is a very powerful tool but takes time to get familiar with the different libraries and tracing mechanisms. Other challenges included using Virtual Box to develop the ns-3 simulation in a Linux environment. Running over 200 simulations in this project took a significant amount of time.

7.1 Future Work

Possible future work for this project could be done to gain more insight into MWSN networks in an industrial warehouse scenario. Changing the type of traffic from UDP to TCP would be interesting to see how the network deals with retransmission. Further investigation into table versus demand driven protocols. This could include varying the source transmission data rate with time, changing the speed of mobile nodes, and changing the number of nodes in the network with time. Other possibilities include changing the number of data-generating nodes, measuring the average packet delay in the network, and adding a hybrid ad-hoc routing algorithm.

References

[1] V. Jayalakshmi and A. Razak, “A study on issues and challenges in mobile ad hoc networks,” *International Journal of Innovative Research in Computer and Communication Engineering,* vol. 3, no. 9, September, 2015. [Online serial]. Available: http://www.rroij.com/open-access/a-study-on-issues-and-challenges-in-mobile-ad-hoc-networks-IJIRCCE-2015-%200309003.pdf. [Accessed February 9, 2019].

[2] V. Ramasamy, “Mobile wireless sensor networks: an overview,” *ResearchGate,* October, 2017. [Online serial]. Available: https://www.researchgate.net/publication/320266947_Mobile_Wireless_Sensor_Networks_An_Overview. [Accessed February 9, 2019].

[3] S. Lally and L. Trajkovic, “Performance Analysis of Routing Protocols for Wireless Ad-Hoc Networks,” [online document], Available: http://www2.ensc.sfu.ca/~ljilja/papers/Opnetwork2011_lally_final.pdf [Accessed February 15, 2019].

[4] A. Winfield, “Distributed sensing and data collection via broken ad hoc wireless connected networks of mobile robots,” *ResearchGate,* January, 2000. [Online serial]. Available: https://www.researchgate.net/publication/221230049_Distributed_Sensing_and_Data_Collection_Via_Broken_Ad_Hoc_Wireless_Connected_Networks_of_Mobile_Robots. [Accessed February 9, 2019].

[5] U. Burgos, U. Amozarrain, C. Gómez-Calzado and A. Lafuente, “Routing in mobile wireless sensor networks: a leader-based approach,” *MDPI,* July, 2017. [Online serial]. Available: https://www.mdpi.com/1424-8220/17/7/1587/pdf. [Accessed February 9, 2019].

[6] C. Han, Y. Yang and X. Han, “A fast network coding scheme for mobile wireless sensor networks,” *International Journal of Distributed Sensor Networks,* February, 2017. [Online serial]. Available: https://journals.sagepub.com/doi/full/10.1177/1550147717693241. [Accessed February 9, 2019].

[7] Wikipedia, "Ad hoc On-Demand Distance Vector Routing," March, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Ad_hoc_On-Demand_Distance_Vector_Routing. [Accessed April 9, 2019].

[8] Wikipedia, "Destination-Sequenced Distance Vector routing," November, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Destination-Sequenced_Distance_Vector_routing. [Accessed April 10, 2019].

[9] Wikipedia, "Dynamic Source Routing," August, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Dynamic_Source_Routing. [Accessed April 10, 2019].

[10] Wikipedia, "Optimized Link State Routing Protocol," January, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Optimized_Link_State_Routing_Protocol. [Accessed April 10, 2019].

Appendix A: Simulation Code

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

using namespace ns3;
using namespace dsr;

NS_LOG_COMPONENT_DEFINE ("manet-routing-compare");

class RoutingExperiment
{
public:
  RoutingExperiment ();
  void Run (std::string CSVfileName);
  //static void SetMACParam (ns3::NetDeviceContainer & devices,
  //                         int slotDistance);
  std::string CommandSetup (int argc, char **argv);

private:
  Ptr<Socket> SetupPacketReceive (Ipv4Address addr, Ptr<Node> node);
  void ReceivePacket (Ptr<Socket> socket);
  void CheckThroughput ();

  uint32_t port;
  uint32_t bytesTotal;
  uint32_t packetsReceived;

  std::string m_CSVfileName;
  int m_nSinks;
  std::string m_protocolName;
  double m_txp;
  bool m_traceMobility;
  uint32_t m_protocol;
  uint32_t m_nodes;
  uint32_t m_max_width;
  uint32_t m_max_height;
  uint32_t m_rate;
};

RoutingExperiment::RoutingExperiment ()
  : port (9),
    bytesTotal (0),
    packetsReceived (0),
    m_CSVfileName ("manet-routing.output.csv"),
  m_nSinks(10),
  m_txp(7.5),
    m_traceMobility (false),
    m_protocol (2),// AODV
  m_nodes (50),
    m_max_width(300),
    m_max_height(1500),
  m_rate(2048)
{
}

static inline std::string
PrintReceivedPacket (Ptr<Socket> socket, Ptr<Packet> packet, Address senderAddress)
{
```

```
PrintReceivedPacket (Ptr<Socket> socket, Ptr<Packet> packet, Address senderAddress)
{
  std::ostringstream oss;

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

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

void
RoutingExperiment::ReceivePacket (Ptr<Socket> socket)
{
  Ptr<Packet> packet;
  Address senderAddress;
  while ((packet = socket->RecvFrom (senderAddress)))
    {
      bytesTotal += packet->GetSize ();
      packetsReceived += 1;
      NS_LOG_UNCOND (PrintReceivedPacket (socket, packet, senderAddress));
    }
}

void
RoutingExperiment::CheckThroughput ()
{
  double kbs = (bytesTotal * 8.0) / 1000;
  bytesTotal = 0;

  std::ofstream out (m_CSVfileName.c_str (), std::ios::app);

  out << (Simulator::Now ()).GetSeconds () << ","
      << kbs << ","
      << packetsReceived << ","
      << m_nSinks << ","
      << m_protocolName << ","
      << m_txp << ""
      << std::endl;

  out.close ();
  packetsReceived = 0;
  Simulator::Schedule (Seconds (1.0), &RoutingExperiment::CheckThroughput, this);
}

Ptr<Socket>
RoutingExperiment::SetupPacketReceive (Ipv4Address addr, Ptr<Node> node)
{
  TypeId tid = TypeId::LookupByName ("ns3::UdpSocketFactory");
  Ptr<Socket> sink = Socket::CreateSocket (node, tid);
  InetSocketAddress local = InetSocketAddress (addr, port);
  sink->Bind (local);
  sink->SetRecvCallback (MakeCallback (&RoutingExperiment::ReceivePacket, this));
  return sink;
}

std::string
RoutingExperiment::CommandSetup (int argc, char **argv)
{
  CommandLine cmd;
  cmd.AddValue ("CSVfileName", "The name of the CSV output file name", m_CSVfileName);
```

```
  cmd.AddValue ("num_sinks", "Number of sink/source pairs", m_nSinks);
  cmd.AddValue ("traceMobility", "Enable mobility tracing", m_traceMobility);
  cmd.AddValue ("protocol", "1=OLSR;2=AODV;3=DSDV;4=DSR", m_protocol);
  cmd.AddValue ("nodes", "The number of nodes", m_nodes);
  cmd.AddValue ("max_width", "The maximum spatial width for mobile nodes", m_max_width);
  cmd.AddValue ("max_height", "The maximum spatial height for mobile nodes", m_max_height);
  cmd.AddValue ("txp", "Wifi transmission power", m_txp);
  cmd.AddValue ("source_drate", "source's transmission rate", m_rate);
  cmd.Parse (argc, argv);
  return m_CSVfileName;
}

int
main (int argc, char *argv[])
{
  RoutingExperiment experiment;
  std::string CSVfileName = experiment.CommandSetup (argc,argv);

  //blank out the last output file and write the column headers
  std::ofstream out (CSVfileName.c_str ());
  out << "SimulationSecond," <<
  "ReceiveRate," <<
  "PacketsReceived," <<
  "NumberOfSinks," <<
  "RoutingProtocol," <<
  "TransmissionPower" <<
  std::endl;
  out.close ();

  experiment.Run (CSVfileName);
}

void
RoutingExperiment::Run (std::string CSVfileName)
{
  Packet::EnablePrinting ();
  m_CSVfileName = CSVfileName;

  int nWifis = m_nodes;

  double TotalTime = 150.0;
  std::string rate = std::to_string(m_rate) + "bps"; //("2048bps"); 2048bps
  std::string phyMode ("DsssRate11Mbps");
  std::string tr_name ("manet-routing-compare");
  int nodeSpeed = 20; //in m/s
  int nodePause = 0; //in s
  m_protocolName = "protocol";

  Config::SetDefault  ("ns3::OnOffApplication::PacketSize",StringValue ("64"));
  Config::SetDefault ("ns3::OnOffApplication::DataRate",  StringValue (rate));

  //Set Non-unicastMode rate to unicast mode
  Config::SetDefault ("ns3::WifiRemoteStationManager::NonUnicastMode",StringValue (phyMode));

  NodeContainer adhocNodes;
  adhocNodes.Create (nWifis);

  // setting up wifi phy and channel using helpers
  WifiHelper wifi;
  wifi.SetStandard (WIFI_PHY_STANDARD_80211b);

  YansWifiPhyHelper wifiPhy =  YansWifiPhyHelper::Default ();
  YansWifiChannelHelper wifiChannel;
  wifiChannel.SetPropagationDelay ("ns3::ConstantSpeedPropagationDelayModel");
  wifiChannel.AddPropagationLoss ("ns3::FriisPropagationLossModel");
  wifiPhy.SetChannel (wifiChannel.Create ());
```

```
268
269   // Add a mac and disable rate control
270   WifiMacHelper wifiMac;
271   wifi.SetRemoteStationManager ("ns3::ConstantRateWifiManager",
272                                 "DataMode",StringValue (phyMode),
273                                 "ControlMode",StringValue (phyMode));
274
275   wifiPhy.Set ("TxPowerStart",DoubleValue (m_txp));
276   wifiPhy.Set ("TxPowerEnd", DoubleValue (m_txp));
277
278   wifiMac.SetType ("ns3::AdhocWifiMac");
279   NetDeviceContainer adhocDevices = wifi.Install (wifiPhy, wifiMac, adhocNodes);
280
281   MobilityHelper mobilityAdhoc;
282   int64_t streamIndex = 0; // used to get consistent mobility across scenarios
283
284   ObjectFactory pos;
285   std::string X = "ns3::UniformRandomVariable[Min=0.0|Max=" + std::to_string(m_max_width) + ".0]";
286   std::string Y = "ns3::UniformRandomVariable[Min=0.0|Max=" + std::to_string(m_max_height) + ".0]";
287   pos.SetTypeId ("ns3::RandomRectanglePositionAllocator");
288   pos.Set ("X", StringValue (X));
289   pos.Set ("Y", StringValue (Y));
290
291   Ptr<PositionAllocator> taPositionAlloc = pos.Create ()->GetObject<PositionAllocator> ();
292   streamIndex += taPositionAlloc->AssignStreams (streamIndex);
293
294   std::stringstream ssSpeed;
295   ssSpeed << "ns3::UniformRandomVariable[Min=0.0|Max=" << nodeSpeed << "]";
296   std::stringstream ssPause;
297   ssPause << "ns3::ConstantRandomVariable[Constant=" << nodePause << "]";
298   mobilityAdhoc.SetMobilityModel ("ns3::RandomWaypointMobilityModel",
299                                   "Speed", StringValue (ssSpeed.str ()),
300                                   "Pause", StringValue (ssPause.str ()),
301                                   "PositionAllocator", PointerValue (taPositionAlloc));
302   mobilityAdhoc.SetPositionAllocator (taPositionAlloc);
303   mobilityAdhoc.Install (adhocNodes);
304   streamIndex += mobilityAdhoc.AssignStreams (adhocNodes, streamIndex);
305   NS_UNUSED (streamIndex); // From this point, streamIndex is unused
306
307   AodvHelper aodv;
308   OlsrHelper olsr;
309   DsdvHelper dsdv;
310   DsrHelper dsr;
311   DsrMainHelper dsrMain;
312   Ipv4ListRoutingHelper list;
313   InternetStackHelper internet;
314
315   switch (m_protocol)
316     {
317     case 1:
318       list.Add (olsr, 100);
319       m_protocolName = "OLSR";
320     std::cout << "\n OLSR\n";
321       break;
322     case 2:
323       list.Add (aodv, 100);
324       m_protocolName = "AODV";
325     std::cout << "\n AODV\n";
326       break;
327     case 3:
328       list.Add (dsdv, 100);
329       m_protocolName = "DSDV";
330     std::cout << "\n DSDV\n";
331       break;
332     case 4:
333       m_protocolName = "DSR";
```

```
    std::cout << "\n DSR\n";
      break;
    default:
      NS_FATAL_ERROR ("No such protocol:" << m_protocol);
    }

  if (m_protocol < 4)
    {
      internet.SetRoutingHelper (list);
      internet.Install (adhocNodes);
    }
  else if (m_protocol == 4)
    {
      internet.Install (adhocNodes);
      dsrMain.Install (dsr, adhocNodes);
    }

  NS_LOG_INFO ("assigning ip address");

  Ipv4AddressHelper addressAdhoc;
  addressAdhoc.SetBase ("10.1.1.0", "255.255.255.0");
  Ipv4InterfaceContainer adhocInterfaces;
  adhocInterfaces = addressAdhoc.Assign (adhocDevices);

  OnOffHelper onoff1 ("ns3::UdpSocketFactory",Address ());
  onoff1.SetAttribute ("OnTime", StringValue ("ns3::ConstantRandomVariable[Constant=1.0]"));
  onoff1.SetAttribute ("OffTime", StringValue ("ns3::ConstantRandomVariable[Constant=0.0]"));

  Ptr<Socket> sink = SetupPacketReceive (adhocInterfaces.GetAddress (0), adhocNodes.Get (0));
  AddressValue remoteAddress (InetSocketAddress (adhocInterfaces.GetAddress (0), port));
  onoff1.SetAttribute ("Remote", remoteAddress);

  for (int i = 0; i < m_nSinks; i++)
    {


      Ptr<UniformRandomVariable> var = CreateObject<UniformRandomVariable> ();
      ApplicationContainer temp = onoff1.Install (adhocNodes.Get (i + m_nSinks));
      temp.Start (Seconds (var->GetValue (50.0,51.0)));
      temp.Stop (Seconds (TotalTime));
    }

  std::stringstream ss;
  ss << nWifis;
  std::string nodes = ss.str ();

  std::stringstream ss2;
  ss2 << nodeSpeed;
  std::string sNodeSpeed = ss2.str ();

  std::stringstream ss3;
  ss3 << nodePause;
  std::string sNodePause = ss3.str ();

  std::stringstream ss4;
  ss4 << rate;
  std::string sRate = ss4.str ();

  NS_LOG_INFO ("Configure Tracing.");
  tr_name = tr_name + "_" + m_protocolName +"_" + nodes + "nodes_" + sNodeSpeed + "speed_" + sNodePause + "pause_" + sRate + "rate";

  AsciiTraceHelper ascii;
  Ptr<OutputStreamWrapper> osw = ascii.CreateFileStream ( (tr_name + ".tr").c_str());
  wifiPhy.EnableAsciiAll (osw);
 // AsciiTraceHelper ascii;
 // MobilityHelper::EnableAsciiAll (ascii.CreateFileStream (tr_name + ".mob"));

  //Ptr<FlowMonitor> flowmon;
```

```
406     NS_LOG_INFO ("Run Simulation.");
407
408     CheckThroughput ();
409
410     Simulator::Stop (Seconds (TotalTime));
411     Simulator::Run ();
412
413   //  flowmon->SerializeToXmlFile ((tr_name + ".flowmon").c_str(), false, false);
414
415     Simulator::Destroy ();
416   }
417
418
```

Appendix B: Python Script

```python
import numpy as np
import csv
import os


protocols = [1, 2, 3, 4] #OLSR, AODV, DSDV, DSR
csv.register_dialect('d', delimiter=',')
# ##### AREA COMPARISION ######
areas = [100, 300, 500, 700, 900, 1100, 1300, 1500, 1700, 1900, 2100, 2300, 2500] #sqrt meters
csv_rows = [['area', 'OLSR', 'AODV', 'DSDV', 'DSR']]

for area in areas:
  new_csv_row = []
  new_csv_row.append(area)
  for protocol in protocols:
    #create function call
    cmd_line = "./waf --run \"scratch/p --protocol=" + str(protocol)
    cmd_line += " --max_width=" + str(int(area))
    cmd_line += " --max_height=" + str(int(area))
    csv_filename = "p" + str(protocol) + "_a" + str(area) + ".csv"
    cmd_line += " --CSVfileName=" + csv_filename + "\""

    #call function
    # os.system(cmd_line)

    #read this csv
    readdata = csv.reader(open('p1_a300.csv', 'r')) #test
    # readdata = csv.reader(open(csv_filename, 'r'))

    #Comute average throughput
    data = []
    for row in readdata:
      data.append(row)
    data.pop(0)

    q1 = []
    for i in range(len(data)):
      q1.append(int(data[i][2])) #2 is the recieved packets
    new_csv_row.append(float((np.mean(q1))))

  csv_rows.append(new_csv_row)

#write .csv
with open('area_comparision.csv', 'wb', newline='') as csvFile:
    writer = csv.writer(csvFile, dialect='d')
    for row in csv_rows:
      writer.writerow(row)

csvFile.close()

# ##### NODE COMPARISION ######
nodes = [20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80] #number of mobile nodes
csv_rows = [['number of nodes', 'OLSR', 'AODV', 'DSDV', 'DSR']]

for node in nodes:
  new_csv_row = []
  new_csv_row.append(node)
  for protocol in protocols:
    #create function call
    cmd_line = "./waf --run \"scratch/p --protocol=" + str(1)
    cmd_line += " --nodes=" + str(node)
    csv_filename = "p" + str(1) + "_n" + str(node) + ".csv"
    cmd_line += " --CSVfileName=" + csv_filename + "\""

    #call function
    os.system(cmd_line)

    #read this csv
    #readdata = csv.reader(open('aodv-manet-routing.output.csv', 'r')) #test
```

```
    #readdata = csv.reader(open('aodv-manet-routing.output.csv', 'r')) #test
    readdata = csv.reader(open(csv_filename, 'r'))

    #Comute average throughput
    data = []
    for row in readdata:
      data.append(row)
    data.pop(0)

    q1 = []
    for i in range(len(data)):
      q1.append(int(data[i][2])) #2 is the recieved packets
    new_csv_row.append(float((np.mean(q1))))

  csv_rows.append(new_csv_row)

#write .csv
with open('node_comparision.csv', 'w', newline='') as csvFile:
    writer = csv.writer(csvFile, dialect='d')
    for row in csv_rows:
      writer.writerow(row)

csvFile.close()

# ##### TXP COMPARISION ######
txps = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30] #transmit power in dBm
csv_rows = [['txp', 'OLSR', 'AODV', 'DSDV', 'DSR']]

for txp in txps:
  new_csv_row = []
  new_csv_row.append(txp)
  for protocol in protocols:
    #create function call
    cmd_line = "./waf --run \"scratch/p --protocol=" + str(protocol)
    cmd_line += " --txp=" + str(txp) + ".0"
    csv_filename = "p" + str(protocol) + "_t" + str(txp) + ".csv"
    cmd_line += " --CSVfileName=" + csv_filename + "\""

    #call function
    os.system(cmd_line)

    #read this csv
    # readdata = csv.reader(open('aodv-manet-routing.output.csv', 'r')) #test
    readdata = csv.reader(open(csv_filename, 'r'))

    #Comute average throughput
    data = []
    for row in readdata:
      data.append(row)
    data.pop(0)

    q1 = []
    for i in range(len(data)):
      q1.append(int(data[i][2])) #2 is the recieved packets
    new_csv_row.append(float((np.mean(q1))))

  csv_rows.append(new_csv_row)

#write .csv
with open('txp_comparision.csv', 'w', newline='') as csvFile:
    writer = csv.writer(csvFile, dialect='d')
    for row in csv_rows:
      writer.writerow(row)

csvFile.close()

##### SENDER DATA RATE COMPARISION ######
drates = [100, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000] #data rate in bps
csv_rows = [['drate', 'OLSR', 'AODV', 'DSDV', 'DSR']]
```

```
csv_rows = [['drate', 'OLSR', 'AODV', 'DSDV', 'DSR']]

for drate in drates:
  new_csv_row = []
  new_csv_row.append(drate)
  for protocol in protocols:
    #create function call
    cmd_line = "./waf --run \"scratch/p --protocol=" + str(protocol)
    cmd_line += " --source_drate=" + str(drate)
    csv_filename = "p" + str(protocol) + "_d" + str(drate) + ".csv"
    cmd_line += " --CSVfileName=" + csv_filename + "\""

    #call function
    os.system(cmd_line)

    #read this csv
    # readdata = csv.reader(open('aodv-manet-routing.output.csv', 'r')) #test
    readdata = csv.reader(open(csv_filename, 'r'))

    #Comute average throughput
    data = []
    for row in readdata:
      data.append(row)
    data.pop(0)

    q1 = []
    for i in range(len(data)):
      q1.append(int(data[i][2])) #2 is the recieved packets
    new_csv_row.append(float((np.mean(q1))))

  csv_rows.append(new_csv_row)

#write .csv
with open('txp_comparision.csv', 'w', newline='') as csvFile:
    writer = csv.writer(csvFile, dialect='d')
    for row in csv_rows:
      writer.writerow(row)

csvFile.close()

##### LOW TXP + AREA COMPARISION ######
areas = [100, 300, 500, 700, 900, 1100, 1300, 1500, 1700, 1900, 2100, 2300, 2500] #sqrt meters
csv_rows = [['area', 'OLSR', 'AODV', 'DSDV', 'DSR']]

for area in areas:
  new_csv_row = []
  new_csv_row.append(area)
  for protocol in protocols:
    #create function call
    cmd_line = "./waf --run \"scratch/p --protocol=" + str(protocol)
    cmd_line += " --txp=" + "3.0"
    cmd_line += " --max_width=" + str(int(area))
    cmd_line += " --max_height=" + str(int(area))
    csv_filename = "p" + str(protocol) + "_a" + str(area) + "_t" + ".csv"
    cmd_line += " --CSVfileName=" + csv_filename + "\""

    #call function
    os.system(cmd_line)

    #read this csv
    # readdata = csv.reader(open('aodv-manet-routing.output.csv', 'r')) #test
    readdata = csv.reader(open(csv_filename, 'r'))

    #Comute average throughput
    data = []
    for row in readdata:
      data.append(row)
    data.pop(0)
```

```
204         data.pop(0)
205
206         q1 = []
207         for i in range(len(data)):
208             q1.append(int(data[i][2])) #2 is the recieved packets
209         new_csv_row.append(float((np.mean(q1))))
210
211     csv_rows.append(new_csv_row)
212
213 #write .csv
214 with open('area_low_t_comparision.csv', 'w', newline='') as csvFile:
215     writer = csv.writer(csvFile, dialect='d')
216     for row in csv_rows:
217         writer.writerow(row)
218
219 csvFile.close()
220
```