Indiana State University

Sycamore Scholars

Fall 12-1-2017

Distributed Computing in Internet of Things (Iot) Using Mobile Ad Hoc Network (Manet): a Swarm Intelligence Based Approach

John Selvadurai
Indiana State University

Follow this and additional works at: https://scholars.indianastate.edu/etds

Recommended Citation

Selvadurai, John, "Distributed Computing in Internet of Things (Iot) Using Mobile Ad Hoc Network (Manet): a Swarm Intelligence Based Approach" (2017). *Electronic Theses and Dissertations*. 67.
https://scholars.indianastate.edu/etds/67

DISTRIBUTED COMPUTING IN INTERNET OF THINGS (IoT) USING MOBILE AD HOC

NETWORK (MANET): A SWARM INTELLIGENCE BASED APPROACH

_____

A Dissertation

Presented to

The College of Graduate and Professional Studies

College of Technology

Indiana State University

Terre Haute, Indiana

_____

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

_____

by

John Selvadurai

December 2017

Keywords: IoT, MANET, Distributed Computing, Technology Management, ACO, PSO, ABC

CURRICULUM VITAE

JOHN SELVADURAI

EDUCATION

Indiana State University   2017

    Ph.D.: Technology Management, Digital Communications

East Carolina University   2016

    MS.: Network Technology, Digital Communications

California State University   2009

    MBA.: Business Administration

California State University   2006

    MS: Computer Science

American University of Asia   2002

    BS: Computer Science

EMPLOYMENT HISTORY

Innovation Fellow

    Iterate.ai, Denver, Colorado.

Software Development

    SunGard Public Sector, Chico, California.

PUBLICATIONS

Selvadurai, J. (2013). Legal and Ethical Responsibilities in Mobile Payment Privacy. International Journal of Scientific and Technology Research. June, 2013.

Selvadurai, J. (2013). A Natural Language Processing based Web Mining System for Social Media Analysis. International Journal of Scientific and Research Publications. January 2013.

Selvadurai. J. (2012). A Mobile Commerce Architecture Based on Location Based Services and Social Media Monitoring. International Journal of Scientific and Engineering Research. September 2012.

Selvadurai. J. (2006). A Performance Enhancing Web Service Architecture based on Priority Scheduling Mechanism. California State University. 2006.

COMMITTEE MEMBERS

Committee Chair: Patrick Appiah-Kubi, Ph.D.

    Chair, Database Systems Technology

    Assistant Professor, Department of Information & Technology Systems

    University of Maryland University College

Committee Co-Chair: Mehran Shahhosseini, D.Eng.

    Associate Professor, Department of Applied Engineering and Technology Management

    College of Technology

    Indiana State University

Committee Member: Xiaolong Li, Ph.D.

    Associate Professor, Department of Electronics and Computer Engineering Technology

    Indiana State University

Committee Member: Erol Ozan, Ph.D.

    Associate Professor, Department of Technology Systems

    East Carolina University

ABSTRACT

Internet of Things (IoT) is a fast-growing technological trend, which is expected to revolutionize the world by changing the way we do things. IoT is a concept that encourages all the electronic devices to connect to the internet and interact with each other. By connecting all these devices to the internet, new markets can be created, productivity can be improved, operating costs can be reduced and many other benefits can be obtained. In IoT architecture, often sensors and aggregators collect data and send to a cloud server for analyzing via the traditional cloud-server model. This client-server architecture is not adequate to fulfill the growing requirements of IoT applications because this model is subjected to cloud latency.

This research proposed a distributed computing model called Distributed Shared Optimization (DSO) to eliminate the delay caused by cloud latency. DSO is based on swarm intelligence where algorithms are built by modeling the behaviors of biological agents such as bees, ants, and birds. Mobile Ad-hoc Network (MANET) is used as the platform to build distributed computing. The infrastructure-less and leader-less features of MANET make it the ideal candidate to build IoT with swarm intelligence. To test the theory, this research also built a simulation program and conducted multiple simulations on both DSO and client-server models. The simulation data was analyzed by descriptive statistics and One-Way ANOVA. This research found that there is a significant difference in computing time between DSO and client-server models. Further, Multiple-Regression technique was conducted on DSO simulation data to identify the effect sensors and data had towards DSO computing time.

PREFACE

The purpose of this dissertation is to propose a distributed computing mechanism in IoT environment that can model swarm intelligence behaviors. The results of this study can be applied to build efficient data processing IoT applications.

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to my committee chair Dr. Patrick Appiah-Kubi and committee co-chair Dr. Mehran Shahhosseini. Their continuous support and encouragement led me to accomplish this exciting reward in research.

I also would like to thank my committee members Dr. Xiaolong Li and Dr. Erol Ozan for their relentless guidance and suggestions in this research journey.

Finally, I would like to dedicate my research to my dad late Dr. C.C.K. Selvadurai who highly motivated me to achieve a doctoral degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Internet of Things (IoT) is a concept that refers to the collection of devices (things) that are connected to the internet and interact with each other. Identification numbers, names or addresses could identify each device. In IoT theory, any type of device can be connected to the internet. Those IoT devices can be either home environment devices such as television, microwave, and coffee maker or large machines such as jet engines and factory machinery (Verma, 2016). By connecting all the devices to the internet, information flows to every device seamlessly. Figure 1 describes this concept that all the devices are connected to the internet and each other. IoT can also be considered as a global network that allows communication from human to human, human to things and things to things (Madakam et al., 2015). This creates a major breakthrough in productivity because many actions can be completed without much human interaction.

Applications of IoT are countless. For example, in a home environment, a smart refrigerator can track the expiry dates of food items through sensors and could place an order to the grocery shop if the level goes below a certain limit. In a similar way, home appliances, security cameras, doors, temperature monitors and personal computers can be connected to the internet to provide automated home operations. For elders, IoT applications could provide assisted living through sensors monitoring their health and home safety. In the event of an

emergency, first responders, through sensors, could be called automatically.

IoT applications are not just limited to home environments. When cars, traffic signals, street lamps and roads are connected to the internet, smart decisions can be made to avoid traffic congestion. For example, personal calendar planning applications can retrieve real-time traffic data through sensors in the streets to determine the best time to travel. Similarly, a city's waste, water, and power can be managed efficiently through IoT. This concept leads to smart cities and smart grids.
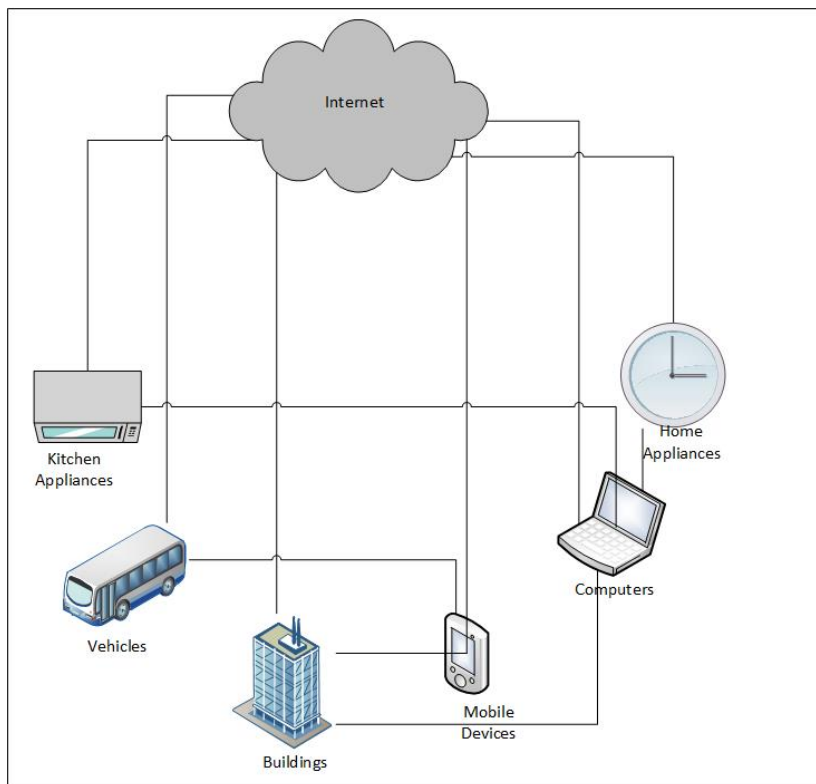
Figure 1. Internet of Things Overview.

The concept of IoT is very broad. In general, IoT structure can be broken into four major layers: sensors, communication, computation and service (Chen, 2012). Sensors collect data. Communication units send this data to computation units for processing. Based on the data analysis, services take action. It is not necessary to have a device for each layer. One or two

devices can function all four components of IoT. For example, a mobile phone could act both as a sensor and a communication unit to send data to the cloud server, which could act as both computation and service units.

By connecting anything to the internet, IoT creates enormous amounts of data. Cisco's Fourth Annual Global Cloud Index Study estimates the data generated by IoT will reach 403 Zeta bytes per year in 2018 (Cisco, 2016). A zeta byte is a trillion of gigabytes and this amount is expected to increase exponentially in the following years when IoT applications become more popular. Usually, sensors generate raw data that passes to cloud servers via aggregators. For an efficient IoT application, large amounts of data must be continuously transferred to the cloud while cloud servers continuously conduct analysis and process on that data. The efficiency of transferring and processing huge amounts of data depends on the communication channel from device to internet and computation capability of cloud servers.

The success of IoT applications relies on how quickly cloud servers can interpret raw data to meaningful information and complete the action to fulfill the purpose. Although client-server architecture is widely considered in implementing IoT applications, it is not always the best fit to address the growing demand in IoT. The major concern in using client-server architecture in IoT applications is the impact of cloud latency. The term cloud latency refers to the time delay in sending a request to the cloud and receiving the response. There are many factors that affect cloud latency. One notable factor that causes cloud latency is the number of multi-hopping required to travel from device to server. Cloud server locations are spread around the world in many different geographical locations. Data packets need to hop multiple servers around the world to reach the target cloud server from the device. Each hopping process could delay the overall operation.

Cloud latency is a major concern in cloud-based businesses. The cost of cloud latency is very high. According to Sharp (2012), a half-second delay could cause Google's traffic drop by 20% and a tenth of a second delay could cause Amazon's sales to drop by 1%.

In certain IoT applications, receiving quick responses from the cloud is very critical. Applications in transport and healthcare industries are time sensitive. A good example of this scenario can be found in self-driving cars. There are numerous sensors built in self-driving cars to identify and analyze the surrounding environments. According to Google, one gigabyte of data is generated for every second their self-driving car travels (Rijmenam, 2016). Based on the massive amount of data collected from sensors, decisions must be made instantly. When the self-driving car is traveling at high speeds, any time delay in the decision-making is very critical. Waiting on cloud servers to receive some decisions creates latency and may result in the failure of the purpose of the application. Therefore, the common model in IoT that depends on cloud servers to respond for every action is not adequate to serve time-sensitive applications. To better utilize the computing power of cloud servers, the focus must be on mainly processing machine learning and artificial intelligence techniques rather than interpreting raw data sent from sensors.

This research identified the time delay issue with the traditional cloud-server model and proposed the distributed computing model to eliminate time delay in latency. The proposed model conducted computing locally between sensors rather than sending raw data to cloud servers. As a side impact, this distributed computing model also helps to offload computing resources from cloud servers to sensors.

Problem Statement

The problem this research addressed is that client-server model causes delays in time-sensitive IoT applications. The main reason for this problem to exist is the cloud latency which delays the response to reach the devices. This problem underscores the need for distributed computing in the IoT environment; however, there is no efficient distributed computing model to reduce response time delay in IoT computing.

Research Theory

IoT networks span across various types of devices. IoT applications often require smartphones, wireless sensors, RFID, and Bluetooth enabled devices to join together to perform data gathering and information processing. Mostly the sensors in IoT are mobile and continuously change their positions. This pattern is highly visible in vehicular networks where sensors in cars collect data of the environment. In these cases, collected data must be sent to aggregators so they can be passed to cloud servers. Mobile Ad-hoc Networks (MANET) can perform this type of continuous data collecting and presenting functionalities profoundly. MANET is a collection of self-organizing wireless nodes that are functioning without any centralized administration (Songfan et al., 2015). MANETs do not require communication infrastructure to function. There is no centralized server in MANET. Nodes in MANET can join together to create a network on demand. Because no infrastructure needs to be built, MANETs can be deployed easily with short notice. This infrastructure-less feature makes MANET an ideal candidate to implement IoT applications. This research uses MANET to model sensors in IoT.

MANET nodes are interconnected with each other. This pattern is diagramed in Figure 2. Because MANET does not have a centralized server to route the communication, each node in

MANET act as both the end node and router. Since nodes in MANET can join and leave the network at any time, the topology is subjected to continuous change. Because of this agility, communication routing from one node to another node is difficult (Kandari, 2011); however, there are protocols in place to handle this challenge. In certain cases, swarm intelligence is applied to build algorithms and protocols in MANET environments.
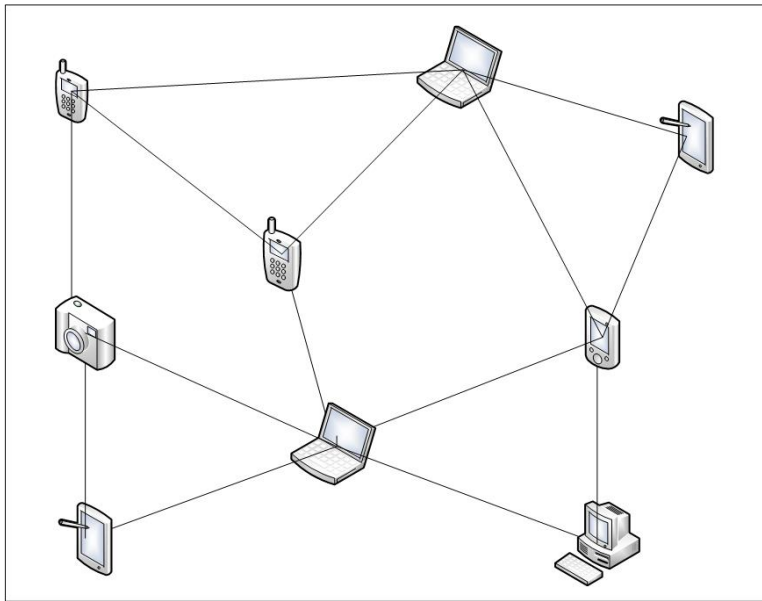
Figure 2. Mobile Ad-hoc Network (MANET)

Swarm intelligence describes the study of how a group of relatively simple agents through their collective interaction accomplish tasks that are far beyond the capabilities of a single agent. Swarm intelligence techniques are inspired by bee swarms, ant colonies, and bird flocks. This is a form of decentralized intelligence in which no central authority makes a decision or directs agents. Each individual agent follows simple rules and cooperates with other agents in the environment in order to complete the tasks. Decentralized control, collaborative learning, and high exploration ability are the main characteristics of this intelligence model. In this intelligence, all the members function autonomously and act upon their available information.

From that available information, they are able to decide their behavior and take appropriate actions.

A good example of swarm intelligence is the colony of ants. Ant colonies are generally social insect societies, which present a highly structural social organization. As a result of this organization, ant colonies can accomplish complex tasks that far exceed the individual capabilities of a single ant. Ant algorithms are derived from the observation of real ant behavior and use these models as a source of inspiration for the solution of optimization and distributed control problems. Ant Colony Optimization is a technique that finds the shortest path to a target using the concepts of how ants find food in the shortest path to their colony (Dorigo and Stutzle, 2004). A foraging ant deposits chemical pheromone on the ground, which increases the probability other ants will follow the same path. Over the time, the pheromone trail will evaporate. The longer paths take longer time for ants to travel and the pheromones have a higher probability to evaporate. In contrast, shorter paths will take a shorter time for ants to travel and the pheromone trail will stay active throughout the time needed to travel. Therefore, ants as collective agents choose to travel the trail that contains the most active pheromones. This specific behavior shows indirect communication can help to achieve self-organization. This original idea has evolved to solve a wider class of numerical and computer science problems.

Similar to ants, birds have a flocking behavior in which they achieve tremendous coordination among member agents without a permanent centralized figure. Without a designated leader, bird flocks fly to the desired destination and stay away from dangers. Figure 3 shows a diagram where birds are flying with coordination. From an outsider's perspective, bird flocks seem to have a leader. In reality, there is no leader in the flock and all the members are following simple basic rules to achieve the coordination.
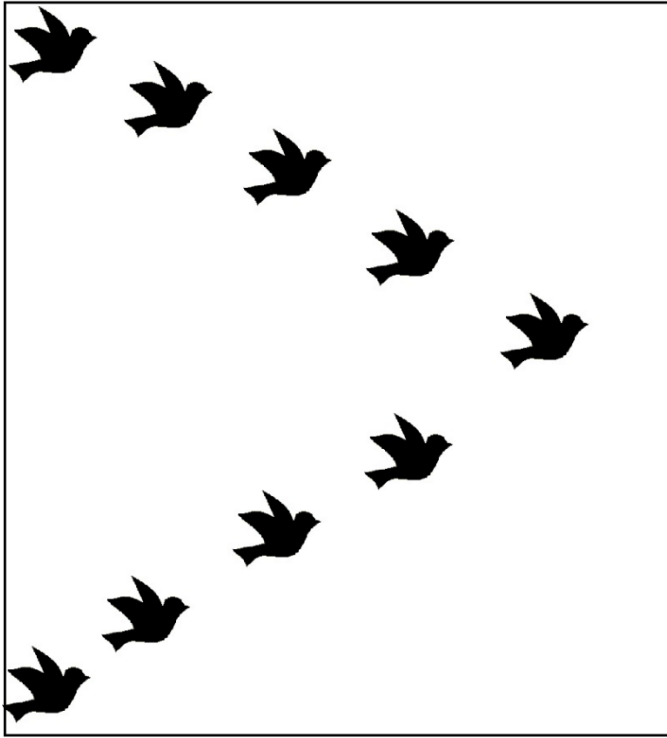
Figure 3. Birds Flocking

Scientists explored this flocking behavior and proposed a flocking model that has three rules (Su et al., 2009):

1) Separation: Avoiding collision with nearby agents.

2) Alignment: Matching velocity with nearby agents.

3) Cohesion: Staying close to nearby agents.

Using these three simple rules, birds accomplish flying long distances. This research is intended to propose a distributed computing model called Distributed Shared Optimization (DSO) based on the flocking theory for IoT applications. This approach is similar to how the ant colony optimization (ACO) technique is used widely in technology systems such as MANET routing to find the shortest path. In this research, flocking theory based DSO is used to employ

distributed computing in MANET based IoT applications. The flocking theory concept of agents coordinating without a centralized figure fits with the requirements of IoT applications and the topology of MANET.

Statement of Purpose

The purpose of this research is to reduce the computing time in IoT applications by proposing a distributed computing model based on swarm intelligence. MANET framework was selected to build the proposed model because of its infrastructure-less and dynamic characteristics. Similarly, swarm intelligence theory was selected to develop the algorithms for the model because of its ability to complete complex tasks without a centralized figure.

Statement of Significance

This research contributes to the body of knowledge in IoT architecture, MANET framework, and network management. The concept of IoT is relatively new and there is a growing demand for applications based on IoT. However, traditional client-server models are producing longer response delay due to cloud latency. Therefore, there is a need for IoT sensors to conduct computing in a distributed way locally without the interaction from cloud servers. This research provides value to the future IoT applications by improving the computing time and offloading the work from cloud servers.

Research Goals and Objectives

The main goal of this research is to reduce the computing delay from cloud servers in IoT applications. To achieve this goal, this research proposes a swarm intelligence based distributed computing model using MANET topology and specifically applying flocking theory to implement the distributed computing model.

Therefore, this research is guided by the following research objectives:

1. Reducing the delay caused by cloud latency by introducing distributed computing to IoT environment.

2. Proposing swarm intelligence-based model called Distributed Shared Optimization (DSO) to assist distributed computing.

Organization of the Research

This research is organized in the following manner:

1. Proposing a flocking theory-based algorithm for distributed computing in IoT applications.

2. Building a simulation tool that implements the prototype of the distributed computing model as well as the client-server model.

3. Testing the algorithm's theory in the built prototype of the simulation tool.

4. Running simulations in both the distributed model and the client-server model of the tool.

5. Applying simulation data to a statistical model to identify how the proposed model affects the computing time of the application.

6. Applying the test data to the statistical model to identify the relationship between IoT application components such as the number of sensors and size of data.

Research Questions

The research questions of this study are as follows:

Q_1: Does the proposed swarm intelligence based Distributed Shared Optimization (DSO)

model reduce the total computing time of the application?

Q_2: How does the number of sensors affect the computing time of the distributed

computing model?

Q_3: How does the size of data being processed affect the computing time of the

distributed computing model?

Q_4: How do both sensors and the size of data being processed together affect the

computing time of the distributed computing model?

Hypotheses

In the process of finding answers to the research questions, the following hypotheses

were established.

Null Hypotheses

H_{01}: There is no difference in computing time of the application when the proposed distributed

computing model is used.

$\mu_{\text{computing time of DSO model}} = \mu_{\text{computing time of client-server model}}.$

H_{02}: There is no relationship between the number of sensors and the computing time in

distributed computing model.

H$_{03}$: There is no relationship between the size of data being processed and computing time in distributed computing model.

H$_{04}$: There is no relationship with computing time by the combination of sensors and the size of data being processed together in distributed computing model.

Alternative Hypotheses

H$_{A1}$: There is a difference in computing time of the application when the proposed distributed computing model is used.

μcomputing time with proposed distributed computing model $\neq$ μcomputing time without proposed distributed computing model.

H$_{A2}$: There is a relationship between the number of sensors and computing time in distributed computing model.

H$_{A3}$: There is a relationship between the size of data being processed and computing time in distributed computing model.

H$_{A4}$: There is a relationship with computing time by the combination of sensors and the size of data being processed together in distributed computing model.

Statement of Methodology

This research is conducted in two phases. The purpose of the first phase is to develop the proposed distributed computing model called Distributed Shared Optimization based on flocking theory. MANET architecture is used as the platform for this model. The first phase also included the development process of the simulation tool to test the theory. In the second phase, this research employs a quasi-experimental design to find the answers to the research questions.

In the quasi-experimental design, the MANET simulator executions were conducted in two sets. One set is the experimental group and the other set is the control group. The treatments are the number of data processing attempts via MANET nodes in the environment.

Data Collection

Both experimental group and control group had 40 simulations. In each simulation, the same treatment was given to both groups. That means both groups received an equal number of MANET nodes and an equal size of data to compute in each simulation. The resulting total computing time for each simulation is collected.

Independent variables: Number of sensors/nodes, Size of the data (bits).

Dependent variable: Computing time. (Total time is taken by the nodes/sensors to complete processing the given data.)

Data Analysis

This research employs quantitative research methods to analyze the data collected from simulations. Analysis of Variance (ANOVA) is used to analyze and identify the relationship between variables. Through the data analysis, the answers to the research questions are understood.

Assumptions

Following assumptions were made in this research.

- The proposed distributed computing model based on flocking theory assumes the distance covered by flocks is similar to data processed by distributed MANET nodes.

- MANET nodes in the simulator are sensors. In other words, each node in the simulator has a single sensor and a processor.

- MANET simulation tool represents the real-world MANET implementation for IoT applications.

- The number of simulations executed in the research is adequate to conduct data analysis and derive the conclusion.

- Computing process and data type specified in the simulation tool represents the real world IoT applications.

Limitations

This research has the following known limitations.

- The number of sensors and size of the data are the only influencing factors considered in the distributed computing in MANET for IoT. Any other factors are considered outside the scope of this research.

- The results of the research are limited to the number of simulations conducted.

- The standard statistical limitations of ANOVA apply to this research as well.

Definitions of Terms

The following list of terms illustrates the technical terminology used in this research.

ABC: Artificial Bee Colony is a swarm intelligence based algorithm that models bees' behavior of finding food sources to find the optimum solution.

ACO: Ant Colony Optimization is a swarm intelligence based algorithm that finds the shortest path to a destination in a similar way to how ants forage for food.

ANOVA: A statistical analysis that tests the means of two or more populations are equal.

Bluetooth: A communication protocol that allows devices to communicate using radio transmissions within a certain range.

Flocking Theory: A part of swarm intelligence studies which models the self-organized, decentralized, collective behavior of birds like agents.

DSO: Distributed Shared Optimization is a swarm intelligence based theoretical model proposed by this research to optimize distributing computing in MANET platform.

IDE: Integrated Development Environment is a development tool that can be used to develop software programs.

IoT: Internet of Things is a concept with an emphasis on connecting all the devices to the internet.

MANET: Mobile Ad-hoc Network is a self-configuring, infrastructure-less, de-centralized mobile network.

NFC: Near Field Communication is communication protocol that enables two devices to communicate using electromagnetic waves when they are in very close proximity.

PSO: Particle Swarm Optimization is a swarm intelligence based algorithm that follows bird flocking behavior to find the optimum solution.

Swarm Intelligence: A collective, decentralized, self-organized behavior of biological and artificial agents.

t-test: A statistical analysis to compare the means of two populations.

CHAPTER 2

LITERATURE REVIEW

This chapter provides the summary of the available literature in the research area. First, the literature related to the technologies of IoT and MANET is discussed. Then the available research literature in swarm intelligence-based protocols and algorithms are explored in detail.

Internet of Things (IoT)

The concept of Internet of Things (IoT) was derived by enhancing everyday physical things by adding an electronic device that can provide intelligence and connectivity to the internet (Kopetz, 2011). Everyday physical things could be anything we use in our daily life. In a home environment, they can be the television, microwave, alarm clock or refrigerator. On a farm, the things could be farm resources such as machinery and equipment. In transportation, they can be cars, trains, streetlights, traffic signals, etc. The idea is to allow any physical thing to intelligently gather and share data through the internet. The ever-reducing cost and size of electronic devices are making this concept to become real at a faster pace.

Many different technologies are used to implement connectivity between devices and the internet in IoT. Radio Frequency Identification (RFID), Wireless Fidelity (Wi-Fi), Bluetooth, and Near Field Communication (NFC) are a few of these technologies (Madakam, 2015). NFC uses electromagnetic waves to communicate while Bluetooth uses radio transmission. The main

advantage of NFC over Bluetooth based devices is the ability of NFC to connect devices without pairing (Shah & Yaqoob, 2016). Internet Protocol (IPv6) is used for internet connectivity in IoT. IPv6 can support up to 2^{128} unique addresses. Since IoT allows any daily life physical thing to be connected to the internet, the demand for unique physical addresses is very high. IPv6 can satisfy this demand.

Even though many technologies are used in IoT, very few standards are in practice. In order to specify some formal descriptions about IoT, the National Institute of Standards and Technology (NIST) has published a document in 2016 (Voas, 2016). This publication specified building blocks or primitives of IoT in which larger systems can be built. The NIST's list of IoT primitives is Sensor, Aggregator, Communication Channel, External Utility, and Decision Trigger.

1. Sensor: The purpose of the sensors is to measure physical properties such as location, sound, weight, light, movement, etc. Sensors could directly connect to the internet or could connect to aggregators.

2. Aggregator: Aggregators are basically software implementations that could transform raw data to aggregated data. This functionality is an important step in big data.

3. Communication Channel: This is the medium the data can be transmitted. This includes the physical aspect as well as virtual aspects such as protocols and software implementations.

4. External Utility: This could be either software or hardware components. Cloud services are examples of external utilities.

5. Decision Trigger: Decision trigger implements the final result that the IoT application is intended for.

Chen et al. (2015) described the idea of distributed computing in IoT using video sensing network. In their research, they also elaborated two case studies to show the impact of distributed computing. Through their case studies, they argued that the computing power of the sensors determines the required bandwidth from sensors to aggregators. Similarly, the computing power of aggregators determines the required bandwidth from aggregators to the cloud servers. Thus, the bottleneck of IoT flow is the communication channel from sensors and aggregators to cloud servers. Therefore, conducting most of the computing on sensors and aggregators could avoid these bottlenecks and would lead to efficient networks. Due to the recent advancement of silicon technology, more computing power can be embedded into sensors and aggregators. By increasing computing power in sensors and aggregators, most of the computation can be done in sensors and aggregators and less workload could be required by cloud servers.

In an effort to bring the distributed concept to IoT, Cardozo et al. (2016), proposed architecture to conduct distributed sensing in IoT. They also implemented a case study in an agricultural area to evaluate the functionalities of the proposed architecture. Their proposal was the solution to bridge the gap between high-level IoT applications and the management of physical devices in gathering sensor data. Their solution provided a middleware to support data acquisition and management to IoT devices. This middleware architecture is managed by rules and driven by events. In their solution, they have implemented a rules engine to perform the treatments in the shortest possible time.

MANET and IoT

MANET provides a dynamic platform for IoT devices to connect and communicate. In MANET, mobile nodes can communicate without any fixed infrastructure like cellular networks.

MANET also does not have any centralized servers to coordinate the nodes. Without the infrastructure and centralized server, nodes in MANET directly establish the peer-to-peer type of connections with neighboring nodes within their transmission range. To deliver data packets outside the transmission range, mobile nodes use the multi-hop principle (Dowla, 2004). In multi-hop, source nodes use a series of intermediate nodes to relay the data packets to the destination node. Therefore, all the mobile nodes in MANET framework are required to store and forward the data packets of other nodes. Because of this nature, MANET nodes are considered both the router and the host in the network.

In MANET, nodes are free to join and leave the network at any given time. Also, nodes can freely move their locations. Therefore, the topology of MANET changes unpredictably in the environment (Umamaheswari & Radhamani, 2012). This autonomous and infrastructure-less nature of MANET makes it the ideal candidate to implement IoT sensors.

The concept of combining MANET and IoT is gaining momentum in the research fields. Many types of research have been done in MANET based IoT such as providing distributed medium access control to IoT-enabled MANETs (Ye & Zhuang, 2016).

Cloud Latency

Cloud computing offers numerous benefits such as flexibility, elasticity, and scalability to IoT applications. Usually, IoT sensors gather data and send to an aggregator in the environment. The aggregator then passes data to cloud servers for data processing. Upon the completion of data processing, results or any instructions to take actions could be sent back to the aggregator. Cloud computing provides economical, highly efficient and fault tolerant services to the end users. In the current setup, a cloud is a group of interconnected computers where all the entities

are interconnected through the internet (Srivastava & Singh, 2016). Data is stored across

multiple servers in the cloud and can be retrieved from anywhere in the world. While cloud

computing gave the flexibility to access data from anywhere, it also introduces latency to retrieve

such data. Cloud latency is a term used to describe the time delay to get a response from cloud

servers.

Bali and Khurana (2013) provided few reasons for cloud latency to occur. Mainly, cloud

latency is caused by the combination of propagation delay, congestion delay, node delay and

processing delay. Propagation delay is the time taken for the head of a signal to travel from

source to destination. Congestion delay is related to congestion caused by a node when it is

carrying more data than it can handle. Processing delay is the time taken by routers to process the

data.

In the current cloud setups, destination points are not fixed. That means a request from

the source may have to travel across the planet through various communication channels and

links to reach the destination. Therefore, it is very common that the requests to the cloud might

experience many congestion, propagation, and processing delays. The cloud latency varies

depending on the number of routers and servers a request needs to hop.

Besides the network causes, virtualization and big data can also add delay to cloud

responses. In the current cloud architecture, it is very popular to run servers on virtualized

machines. This adds another layer of delay when requests are required to travel through a series

of virtual servers (Strom & Zwet, 2015). In big data environments, applications employ

computing in servers distributed across the world. Although this pattern produces flexibility and

efficiency, it also introduces another layer of cloud latency to pass data between servers that are

geographically distanced (Strom & Zwet, 2015). The massive amount of big data stored in the

cloud must be analyzed and queried quickly to return the response. Often, those massive queries take considerable time. When IoT aggregators send data to process, the response from the cloud is subjected to cloud latency due to the above reasons.

Swarm Intelligence based Framework

Swarm intelligence is a methodology to solve numerical optimization problems by modeling biological swarm behaviors (Gui et al., 2016). Algorithms and protocols were developed based on swarm intelligence to solve real-world technical problems.

Karaboga (2005) proposed that two fundamental concepts are required to determine whether a behavior is based on swarm intelligence. These concepts are self-organization and division of labor. Self-organization is a set of mechanisms that establish basic rules to interact between agents of the system. These rules also ensure the agents of the system to purely act on local information without any knowledge of the global pattern. Division of labor is the property that ensures specialized agents perform different tasks simultaneously. This property is important in distributed computing frameworks because simultaneous tasks performed by coordinated agents are more efficient than sequential tasks performed by uncoordinated agents in the traditional systems.

This research studies the following frameworks to develop the proposed Distributed Shared Optimization model.

1. Ant Colony Optimization

2. Particle Swarm Optimization.

3. Artificial Bee Colony.

Ant Colony Optimization (ACO)

Ant Colony Optimization is a methodology proposed by Dorigo & Stutzle (2004) to solve optimization problems by modeling the ants' behavior for foraging food. In real life, ants establish the shortest path from the food source to their nest. Ants release a type of chemical called pheromones to the immediate environment for communication with other ants. This chemical substance evaporates as the time increases (Hlaing & Khine, 2011).
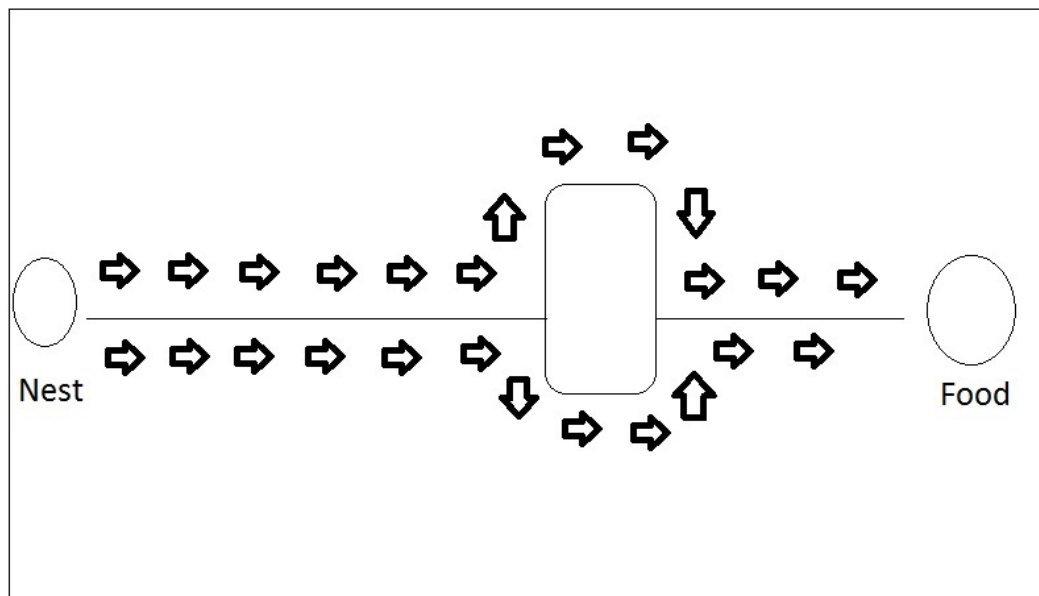
Figure 4. Ant Colony Optimization – All Paths

At the beginning, ants search for food in random order. Figure 4 shows ants moving from the nest to the food in different paths. As they travel through the paths, they release pheromones to the immediate environment and other ants can follow that route (Hlaing & Khine, 2011). Ants also have the tendency to choose the shortest path. As many ants began to choose the shortest path, the shortest path becomes high in pheromone density. On the other hand, pheromones in

other longer paths, which are then less traveled, begin to evaporate. This prompts more ants to choose the shortest path based on the pheromone density as it is shown in Figure 5.
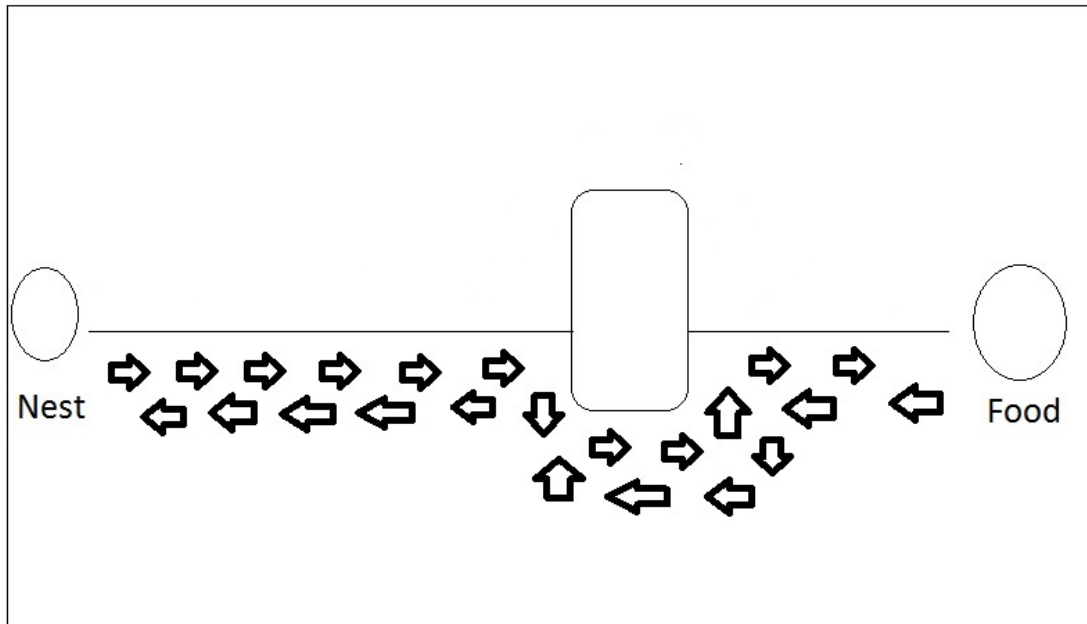
Figure 5. Ant Colony Optimization – Shortest Path

ACO is widely used as a MANET routing protocol (Roy et al., 2012). In MANETs, the network topology constantly changes due to the mobility of the nodes. It is challenging to send data through multi-hoping when the nodes are changing their position. ACO is used in these situations to find the shortest path from the source to the destination node. ACO adopts positive feedback and self-organization (Liu et al., 2014).

There exist many implementations of ACO for MANET. Gunes et al. (2002) described a routing algorithm based on ACO. In their routing algorithm, they established two agents called forward ant and backward ant. The purpose of forwarding ant is to establish a pheromone track from source to destination node. In contrast, backward ant establishes a pheromone track from destination to the source node. These ants have a small packet with a unique sequence number. Each node is able to identify packets based on the source address and the sequence number. At

the beginning, a sender node broadcasts forward ant to all the neighboring nodes and neighboring nodes relay the forward ant to their neighbor nodes. When each node receives the forward ant packet, it creates a record in the routing table. This routing table entry contains pheromone values which determine the number of hops the forward ant needed to reach the node. When the forward ant reaches the destination node, it is destroyed and the backward ant is created to send back to the sender. When a backward ant reaches the sender, the sender has the information about the shortest path to send packets. This algorithm dynamically creates routing paths in MANET and establishes the shortest paths for effective communication.

Particle Swarm Optimization

Kennedy & Eberhart (1995) proposed Particle Swarm Optimization (PSO), which is a simple and effective algorithm based on swarm intelligence. PSO models leaderless animal groups such as bird flocking and fish schooling to guide the particles towards an optimized solution (Wahab et al., 2015). In bird flocking and fish schooling, everyone moves randomly to find food. One of the members should be closer to the food and the rest of them will move closer to that member. Through this continuous communication and coordination, the flock achieves its desired destination.

Bird flocking behavior has three simple mechanisms (Su et al., 2009). They are alignment, cohesion, and separation. Figure 6 describes these behaviors in graphical form. Alignment is the behavior that maintains speed with other flock mates and moving towards the direction where others are moving. Cohesion is a behavior that causes an agent to move towards the center of the mass, which is the average position of other flock mates within a radius. Separation is similar to cohesion but separation allows an agent to steer away from other flock

mates to avoid a collision if the average position becomes too close. Using these three simple

guidelines, birds flock towards the desired destination.
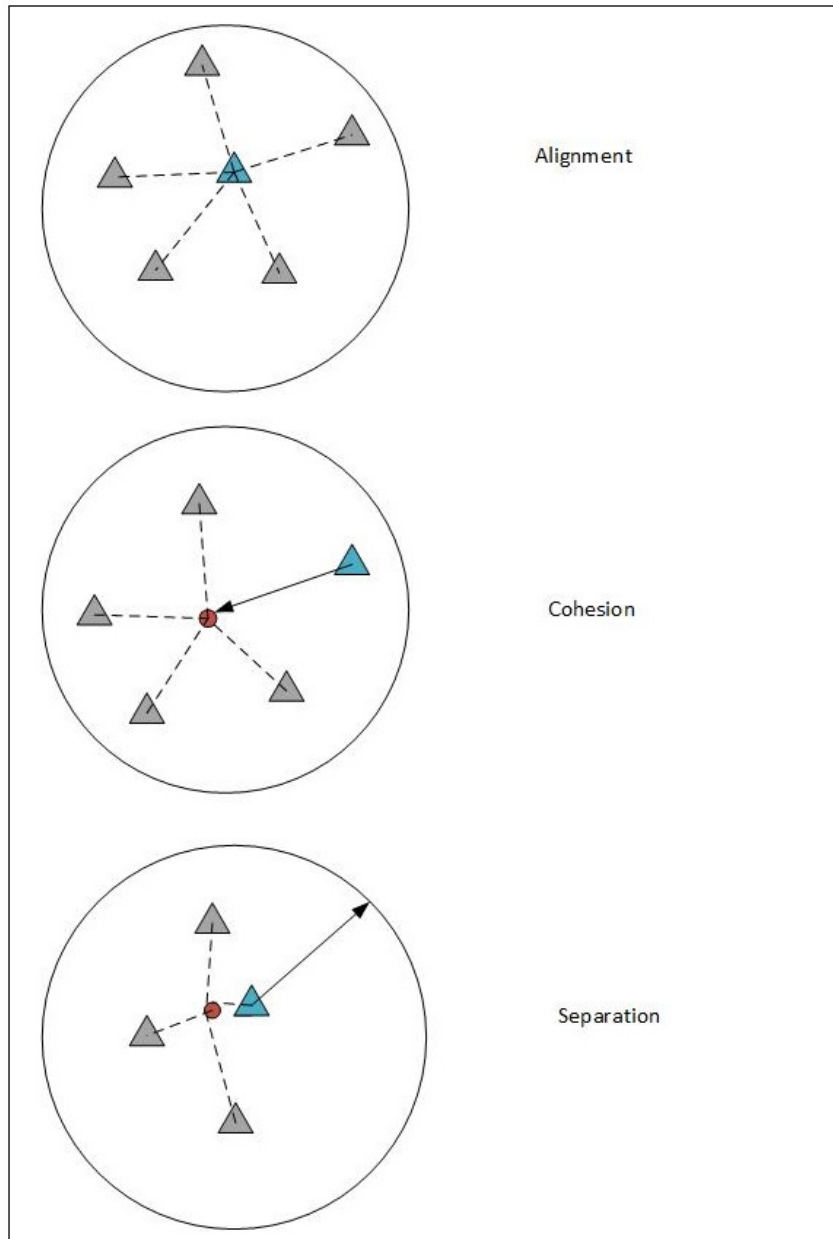
Figure 6. Flocking Behaviors

PSO models these flocking behaviors by considering a swarm of particles in the

problem's search space. Each particle in the swarm represents a potential solution to the main

problem. The objective is to fly through the problem's search space until it finds the best solution. Each particle's direction and distance are calculated by a velocity equation. By moving towards the particle that represents the optimal solution, the solution to the problem will eventually be found (Rini et al., 2011). Each particle's movement is guided by its local best-known position. There is a global best-known position that is updated continuously based on the local best-known position of all the particles. Upon knowing the global best-known position, all the particles move towards the global best-known position. This process is continued until finding the best solution for the problem (Chung, 2005). This is similar to how birds follow the direction of one member who has the knowledge of the food until the entire flock reaches the destination. But the knowledge of the food doesn't rely on one specific member. As the flock moves around the search space, any member could become the closest member to the solution and its local best becomes the global best.

Figure 7 describes a generic implementation of PSO in the form of pseudo code. PSO algorithm is easier to implement and fewer parameters are required to be set. Further, PSO is widely used in networking, machine learning, image processing and signal processing (Wahab et al., 2015).

```
For each Particle
    Initialize particle
End

Do
    For each particle
        Calculate local best value
        If calculated local best value is greater than previous current local best values in history
            Set calculated local best as new current local best
        End If
    End For each

    Choose the particle that has best current local best among all the particles and assign its value
    as global best value

    For each particle
        Calculate particle velocity
        Update particle position
    End For each

While maximim iterations reached or optimal solution is found
```

Figure 7. Particle Swarm Optimization – Pseudo Code

Artificial Bee Colony

Karaboga (2005) proposed Artificial Bee Colony (ABC), which is one of the recent algorithms in swarm intelligence. ABC models the behavior of honeybees when they seek a food source. The bees are categorized into three groups based on the tasks they do.

    i)   Scouts: They are responsible for exploring the surroundings of the beehive to look for a food source.

    ii)   Worker Bees: They exploit the food source and carry information about the food source such as distance and location with them.

    iii)   Onlookers: They look for food sources from the information shared by workers and other bees in the nest.

The important process in a bee colony is the exchange of information between bees. Biologically, this information is exchanged between bees through a special type of dancing.

Based on the type of dance, bees determine whether to incorporate or abandon a food source (Crawford et al., 2014).

ABC algorithm mimics this bee behavior to find the optimum solution. In ABC, the solution to the optimization problem is represented by the position of the food source. An artificial bee moves in the search area to find food sources using the information from its previous experiences and its fellow bees. Some artificial bees fly randomly until they find a food source. When they find a food source, they update the position of the food source and delete the previous food source position if the new position is more accurate than the previous positions (Crawford et al., 2014). They repeat this process until they find the best solution or until they reach the maximum iterations allowed. Thus, the pseudo code of ABC algorithm is similar to Figure 8.

```
Initialize Bees

Do
    Select sites for local search
    Send bees to the selected sites
    Evaluate the fitness level of bees
    Select the best fitness level
    Assign the remaining bees the selected best fitness
    Update the optimum solution
While maximim iterations reached or optimal solution is found
```

Figure 8. Artificial Bee Colony – Pseudo Code

ABC is easy to implement and very flexible. It requires two control parameters. They are the number of maximum cycles and the size of the colony. Adding and removing a bee can be done easily without making major modifications to the implementation. ABC is widely used in networking, scheduling and image processing (Wahab et al., 2015).

CHAPTER 3

METHODOLOGY

This chapter discusses the research methodology of this study. The purpose of this experimental quantitative research was to propose and understand a distributed computing model based on swarm intelligence that could reduce the total computing time in IoT applications. Therefore, this research was conducted in two phases; phase one was to propose the new model and phase two was to conduct statistical analysis to understand the model. In that sense, the first phase developed DSO algorithm and a simulation program. The purpose of the simulation program was to test DSO in MANET platform. The second phase was involved with collecting data from simulation and conducting statistical analysis to find answers to research questions.

Restatement of Purpose

The purpose of this research was to propose a distributed computing model for IoT applications based on swarm intelligence using mobile ad hoc networks (MANET) as the platform.

Thus, this research proposed a distributed computing model called Distributed Shared Optimization (DSO) based on flocking behaviors of birds.

Distributed Shared Optimization

Distributed Shared Optimization (DSO) is an optimization framework to support

distributed computing in IoT applications based on flocking theory of swarm intelligence. DSO

models the behaviors of birds flocking to mechanisms in distributed computing. Flocking theory

lists three simple behaviors in biological agents to coordinate their moving process. These three

behaviors are Cohesion, Alignment, and Separation (Su et al., 2009). This research implemented

three distributed computing mechanisms from these three flocking behaviors. Table 1 describes

three flocking behaviors and their corresponding distributed computing mechanisms proposed by

this research.

Table 1

Flocking Rules and corresponding DSO Mechanisms

Flocking Theory Rules	Rule Description	DSO Mechanism
Cohesion	Moving towards the center of the mass of nearby agents.	A node joining the distributed computing framework.
Alignment	Maintaining the same speed and moving towards the same direction of others.	Maintaining distributed computing with other nodes by sharing the workload.
Separation	Steering away from other flock mates to avoid a collision.	A node leaving the distributed computing framework and notifying others.

DSO maintains a global table in a distributed framework that holds the information about

each node in MANET and its availability to participate in distributed computing. A copy of this

table is maintained at every node in MANET so all the nodes know which nodes are available to

process distributed computing and which are not. This is an important aspect of DSO because

MANET nodes join or leave the network anytime without the coordination of a centralized server. Therefore, distributed computing needs to know which nodes are available and which nodes are not. When a node joins the network, the global table is updated with the new node's information. Similarly, when a node leaves the network, the global table is updated to reflect the unavailability of that node.

DSO implements three simple mechanisms from flocking theory rules. Cohesion in flocking theory suggests an agent fly towards the center of the mass of nearby agents. This rule is essential in joining together with rest of the flock. In DSO, cohesion implies updating the global table when a node joins the network. Through this process, the new node allows the rest of the nodes to know that it is available for distributed processing and is willing to share the workload.

Separation is another flocking rule that allows an agent to steer away from the rest of the flock mates to avoid a collision. In DSO, separation is implemented as the process of a node to steer away from MANET. When the node leaves MANET, it notifies the global table about the departure. This would trigger the rest of the nodes in the distributed computing to take necessary actions to manage the workload with available nodes. The leaving node's workload can also be transferred to another node in the distributed computing framework.

Alignment is a flocking behavior that encourages agents to maintain the same speed as the rest of the flock and move towards the direction where everyone else is heading. DSO models this behavior to encourage nodes to share computing workload of nearby nodes and compute towards rest of the nodes. Figure 9 describes this sharing process in detail.
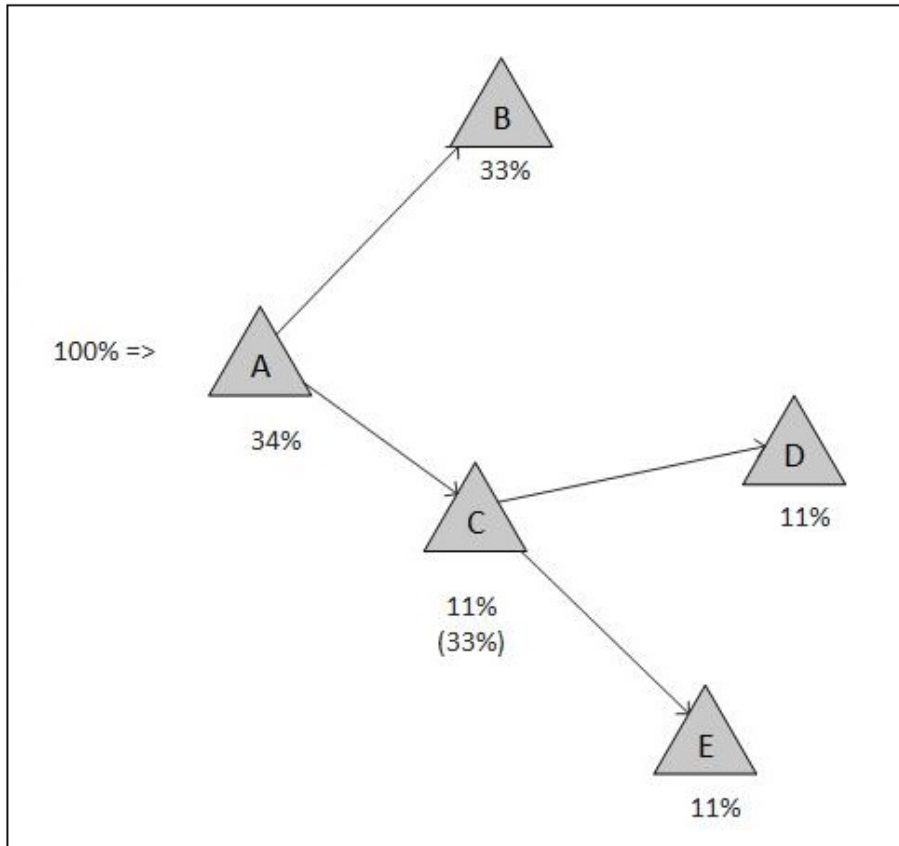
Figure 9. DSO – Functionality

As it is shown in Figure 9, when node A needs to compute a task, it checks the global

table to see the nearby nodes that are available for distributed computing. In this case, node A

sees node B and C are available for distributed computing. Therefore, the task is divided into

thirds to process by nodes A, B and C. Numerically, node A gets approximately 34% of the task

while nodes B and C get 33% each.

Upon receiving distributed tasks to compute, nodes B and C will check their global tables

to see if there are any nodes available to share the process. Since node B does not have any nodes

connected in that network other than node A, node B begins to process its computing task. Node

C has two more nodes (D and E) connected in the network besides node A. Therefore, node C's

given computing task of approximately 33% is divided into thirds to share among nodes C, D

and E. So, nodes C, D, and E get approximately 11% of the task each to compute. This sharing process continues until the main task becomes a manageable size in nodes or until a specified number of sharing processes are completed. Figure 10 describes the flow of the DSO model.
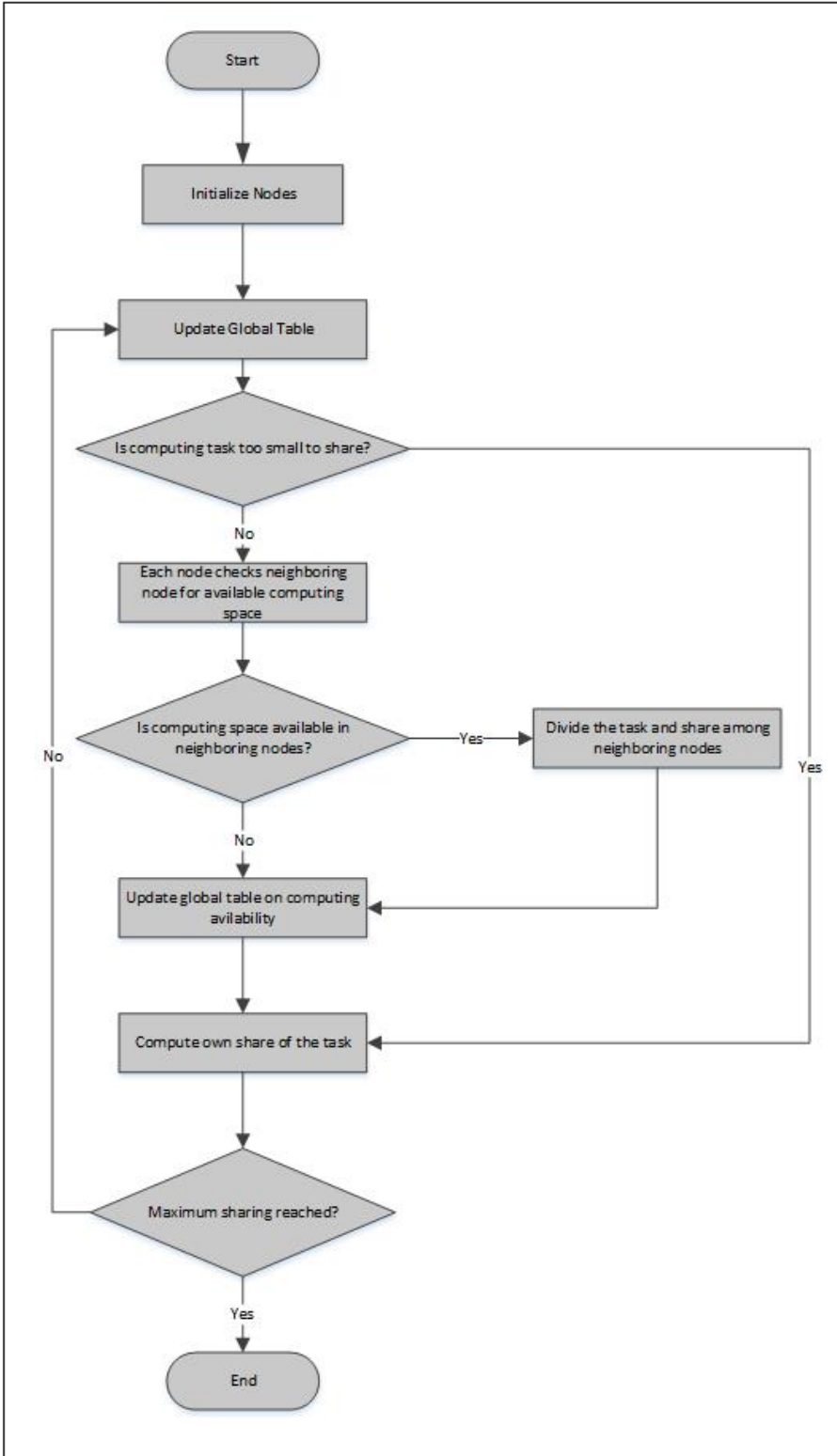
Figure 10. Flowchart of DSO

As figure 10 suggests, the overall flow of DSO is based on distributing and sharing the computing task. At first, the sensors in the environments are initialized along with the global table that holds the distributed computing availability of each node. DSO implementation is better served by recursive functionality. When each node needs to process a task, it checks its neighboring nodes for any available computing space by checking the global table. If any neighboring nodes with available computing space are found, the computing task will be divided among all the nodes. Upon receiving the computing task, each neighboring node updates the global table about their new availability and looks for their neighboring nodes to share the task. In this way, the computing task is divided into smaller chunks until the task becomes manageable to compute faster by all the nodes. In certain environments where there are too many sensors and too small of tasks, it is not efficient to divide and share too many times. In those cases, it is recommended to fix the maximum number of dividing and sharing in the environment. Figure 11 shows the pseudo code for DSO processing.

```
Initialize Nodes
Initialize Global Table

DSO Procedure
Begin

    Update Global Table
    Check computing task size
    If computing task can be divided Then
        Divide the computing task equally between neighboring nodes and self node
        Call DSO Procedure for each neighboring node to compute parts of the task
        compute the part of the task assigned to self node
    Else
        compute the remaining task
    End if

End
```

Figure 11. Pseudo Code for DSO

Research Design

This research proposed the theory of the DSO model to reduce the total computing time in IoT applications. In order to investigate the effects of this model, a statistical analysis was conducted. For the purpose of this statistical analysis, a network simulator was built to conduct simulations for both DSO and client-server models. This network simulator was able to produce a MANET platform with a given number of sensors. The functionality of the simulator was to process the given size of data on the built MANET platform. Both the DSO and client-server models were executed in the simulator for various configurations.

Research Questions

This research was intended to find answers to the following research questions.

Q_1: Does the proposed swarm intelligence based Distributed Shared Optimization (DSO) model reduce the total computing time of the application?

Q_2: How does the number of sensors affect the computing time of DSO model?

Q_3: How does the size of data being processed affect the computing time of DSO model?

Q_4: How do both sensors and the size of data being processed together affect the computing time of the distributed computing model?

Hypotheses

In order to find answers to the research questions, this research constructed the following hypotheses.

Null Hypotheses

H_{01}: There is no difference in computing time of the application when the proposed distributed computing model is used.

$\mu_{\text{computing time of DSO model}} = \mu_{\text{computing time of client-server model}}.$

H$_{02}$: There is no relationship between the number of sensors and the computing time in the distributed computing model.

H$_{03}$: There is no relationship between the size of data being processed and computing time in the distributed computing model.

H$_{04}$: There is no relationship with computing time by the combination of sensors and the size of data being processed together in distributed computing model.

Alternative Hypotheses

H$_{A1}$: There is a difference in computing time of the application when the proposed distributed computing model is used.

$\mu_{\text{computing time with proposed distributed computing model}} \neq \mu_{\text{computing time without proposed distributed computing model}}.$

H$_{A2}$: There is a relationship between the number of sensors and computing time in the distributed computing model.

H$_{A3}$: There is a relationship between the size of data being processed and computing time in the distributed computing model.

H$_{A4}$: There is a relationship with computing time by the combination of sensors and the size of data being processed together in distributed computing model.

Research Methodology

This research is a quantitative experimental research that employed both control group and experimental group. In both groups, the same number of sensors and the same size of data

were given in each simulation. In the control group, the client-server model was implemented but the treatment of the DSO model was absent. In the experimental group, the DSO model was implemented. In total, 40 simulations were executed where each simulation consisted the same number of sensors and size of data. Each simulation was built with randomly created MANET nodes where distances between nodes were varied. Each simulation contained 12 nodes and 2500 bits of data to process. From the simulations, the total computing time in each model, their corresponding number of sensors and processed data sizes were all recorded.

Data collected from simulations were subjected to statistical analysis to determine the answers to research questions. One-way ANOVA was conducted to identify if there is any significant difference in computing time between DSO and client-server models. With subsequent research findings, multiple regression analysis was executed in DSO model's data to explore any significant relationship between the number of sensors and the data sizes on the computing time. In order to find the relationship between sensors and computing time in DSO, each simulation was implemented with the constant data size of 2500 bits and the number of sensors was incremented in every attempt. Similarly, to find the relationship between data size and DSO computing time, the number of sensors was kept at a constant level of 12 and the data size was incremented in every attempt.

Instrumentation

This research first explored the available network simulators to implement DSO. As part of this exploration process, two network simulators were configured and analyzed. First, this research configured OMNeT++ and planned to conduct testing for DSO. Figure 12 shows the basic configuration of two network nodes.
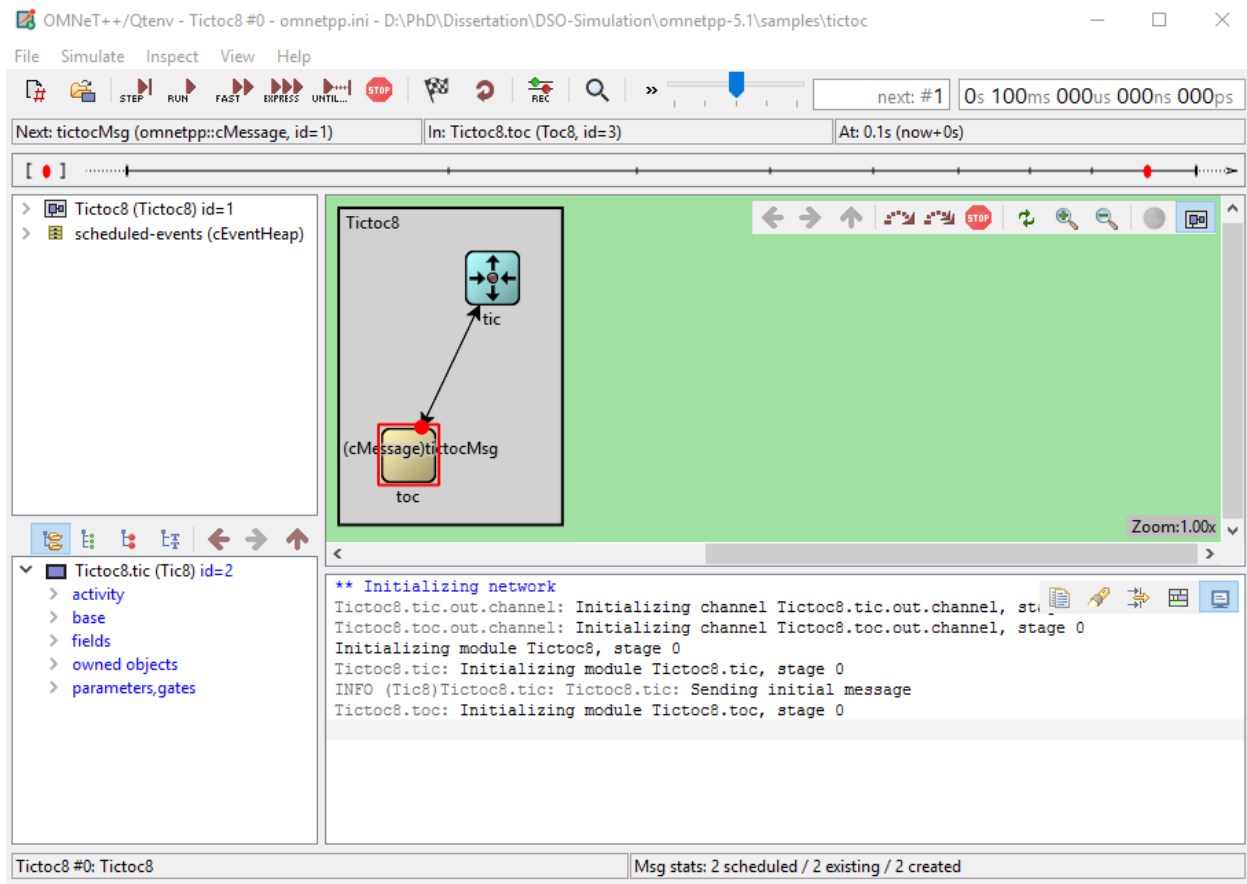
Figure 12. OMNeT++ Simulator

OMNeT++ is a discrete event simulator that supports many different network models (Varga & OpenSim, 2016). For specialty networks such as MANET, there are frameworks built in open source. Through the configuration, this research learned that MANET framework from OMNeT++ cannot be directly used to support big data processing. A significant programming effort was required to enhance the current MANET framework to support big data processing. Also, OMNeT++ uses NED language to define network modules. However, NED language does not provide flexibility to construct a new framework like DSO.

Due to the above shortcomings in OMNeT++ simulation, this research moved to explore Riverbed Modeler as a potential simulator tool for DSO. Riverbed Modeler provides easy to use

network simulation for many different network structures. However, the academic version of

Riverbed Modeler allows up to 20 mobile nodes to construct a network. This is a restricting

factor in this research because having 20 mobile nodes in DSO simulation is an inadequate

model to test the theory. However, Riverbed Modeler can be used to validate other network

simulators.
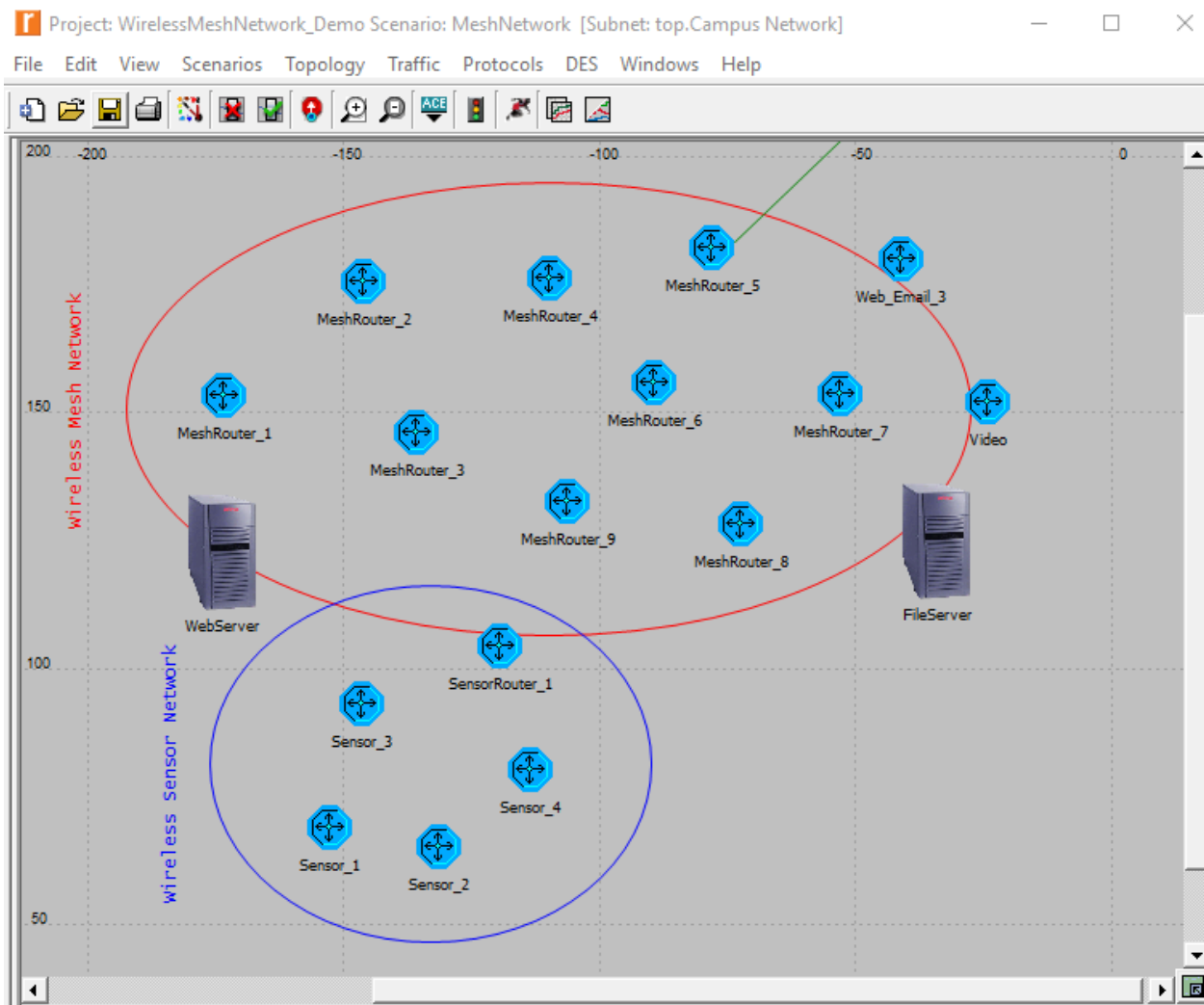
Figure 13. Riverbed Modeler Simulator

After extensive analysis of both the simulators, this research found both the simulators

were not suitable for implementing DSO. This exploration highlighted the need for a specialized

simulator that can implement DSO and provide test data. The following table shows the side by side comparison of tested simulators and DSO simulator.

Table 2

Simulator comparison

	OMNet++	Riverbed Modeler	DSO Simulator
Big data support	No	No	Yes
Ability to connect to external engines such as Spark	No	No	Yes
Programming language limitations	NED (A topology description language)	Proto-C (Enhanced version of C/C++)	C# (Very flexible)
Mobile Node limitation	No limit	Maximum 20 mobile nodes are allowed in academic version.	No limit
Support	No support	Limited support	Homegrown solution
Open source	Yes	No (Academic version available with limited features)	Yes
Time required to incorporate a new framework to the simulator	Approximately 8 months	Approximately 1 year.	Approximately 4 weeks
Level of configuration required to run a simulation	Very high (Needed to configure Mac and Windows libraries)	Relatively less configuration required	Very less. (No overhead configuration because it runs exclusively for DSO)

This research built a DSO simulator to conduct both DSO and client-server simulations. This simulator is capable of producing a given number of MANET nodes for simulation. The procedure of the simulation was to process certain size of data in a given method. The simulator can be configured to run either DSO or the client-server model. The size of the data can also be configured on the simulator. The purpose of this simulator was to conduct data processing simulation in a MANET platform using either DSO or the client-server method. Figure 14 shows the simulator when it was built with 40 sensor nodes and was ready to process 4000 bits of data. In this figure, both DSO and client-server models were executed and their corresponding computing times were displayed. According to the figure, the DSO model took approximately 5.166 seconds while the client-server model took 1 minute and 39.427 seconds. This simulator also provides individual computing time of each sensor node in DSO model. For troubleshooting purposes, neighboring nodes for each MANET node were also provided.

This simulator builds MANET nodes in a random order according to the number of sensors entered in the text box. When the 'Build MANET' button is clicked, random MANET nodes are placed on the 920x580 pixel plane and neighboring nodes are connected to each other. This simulator assumes each MANET node's coverage is 150-pixel length on all the sides and any neighboring nodes within that vicinity get connected together. Besides sensor nodes, the simulator adds one aggregator to the plane. The aggregator's purpose is to aggregate data gathered from the sensor nodes and to transmit to the cloud servers. For identification purposes, the aggregator is coded in red and sensor nodes are coded in blue. For troubleshooting purposes, the simulator also displays the nodes and the list of their neighboring nodes in a grid view.
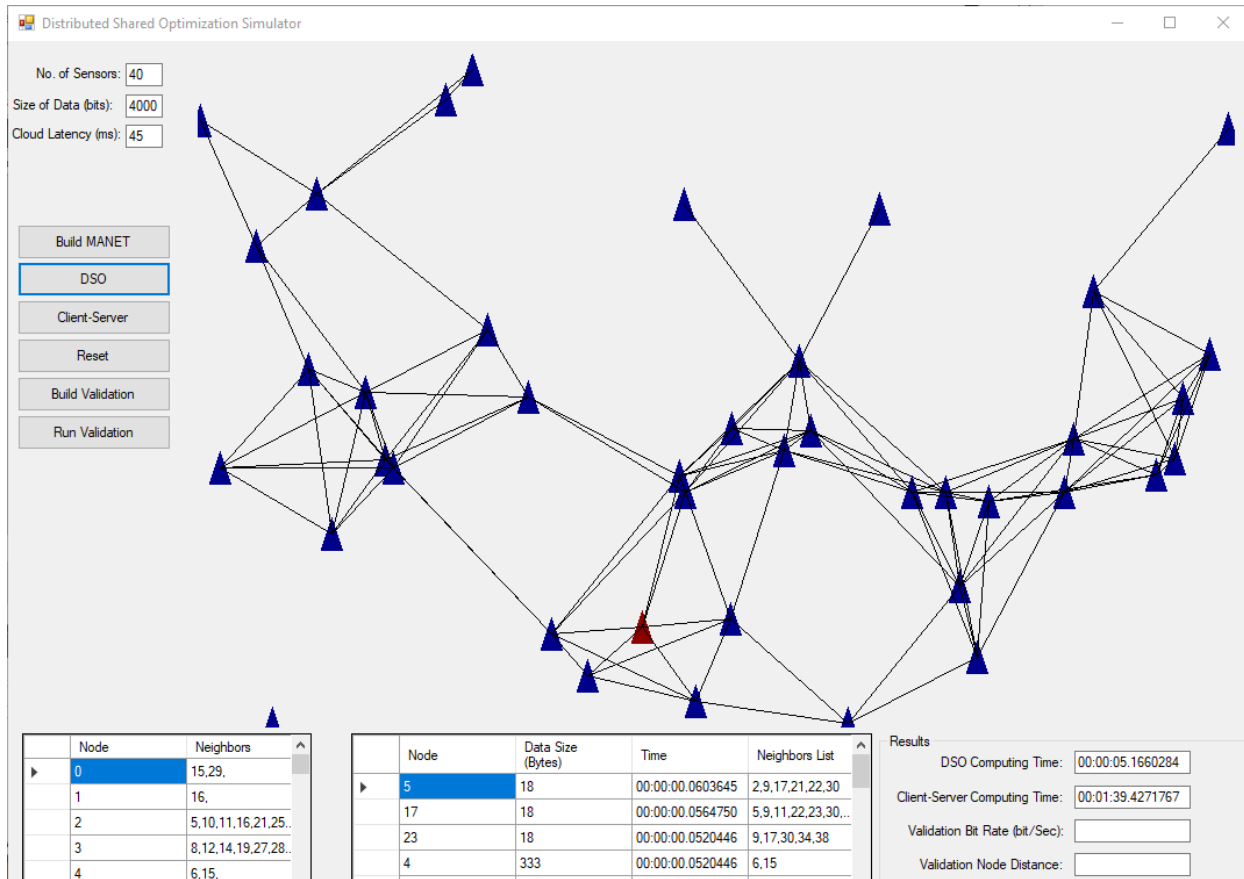
Figure 14. DSO Simulator

Software components of this simulator tool were built using .NET technologies. C# was used as the programming language and Visual Studio 2015 was used as the IDE platform. Multi-Threading was applied in DSO implementation to simulate real-world distributed computing. The entire source code of this simulator is given in Appendix B.

For simulation purposes, this research took an assumption that to process each bit of data, one millisecond of computing time is required. This procedure closely simulated the real-world applications. Therefore, this simulator spent one millisecond of computing time for every data bit given.

DSO simulation begins when the 'DSO' button is clicked. According to the theory, a

global table is initialized. This global table contains every node in that environment and the list of neighboring nodes of each node. Every simulation begins with Node 0. In the DSO model, Node 0 first checks its available neighboring nodes from the global table. Based on the availability, it divides the data to equal bytes among neighboring nodes, including itself. If the data cannot be equally divided between nodes, Node 0 takes one byte more for computation and the rest will be divided among the neighboring nodes. Upon sending divided data to neighboring nodes, Node 0 begins computing its own share of data. At the same time, neighboring nodes begin sharing their portion of the data among their neighbors. Sharing and processing occur recursively through all the nodes in the environment. This process continues until the sharing portion of the data becomes 50 bytes. If the sharing portion of the data is less than 50 bytes, the sharing node processes the entire portion instead of sharing among neighboring nodes. This limit is enforced because sharing too little data is not efficient enough in the DSO model. In real-world applications, MANET nodes can begin processing their portion of data parallel to each other. In order to simulate this parallel processing, this simulation implemented multi-threading in which each node's computing is a separate programming thread and all the threads are independent of the rest of the threads. Therefore, each node's parallel computing occurs at the same time.

The client-server simulation begins when the 'Client-Server' button is clicked. In the client-server model, Node 0 sends entire data to the aggregator. Since node sensors are created randomly, each simulation might have Node 0's and aggregator's positions in various places of the 920x580 pixel plane. The data travels from Node 0 to the aggregator through multi-hoping neighboring nodes. This functionality is the characteristic of the MANET platform. Once the aggregator receives the data, it sends the data to the cloud servers for computing. In that time, the

given cloud latency is applied to the processing time. Cloud latency can also be configured on the simulation tool. At the end of the simulation, the total computing time is displayed for the DSO model and the client-server model.

Instrument Validation

This research took extensive effort to validate the simulator's functionality. The goal of the validation process was to ensure the MANET simulation in DSO simulator aligns with industry standards. In order to run validation, this research selected Riverbed Modeler as the industry standard simulation tool. Riverbed Modeler's MANET model was configured to measure the bit rate through a selected node.
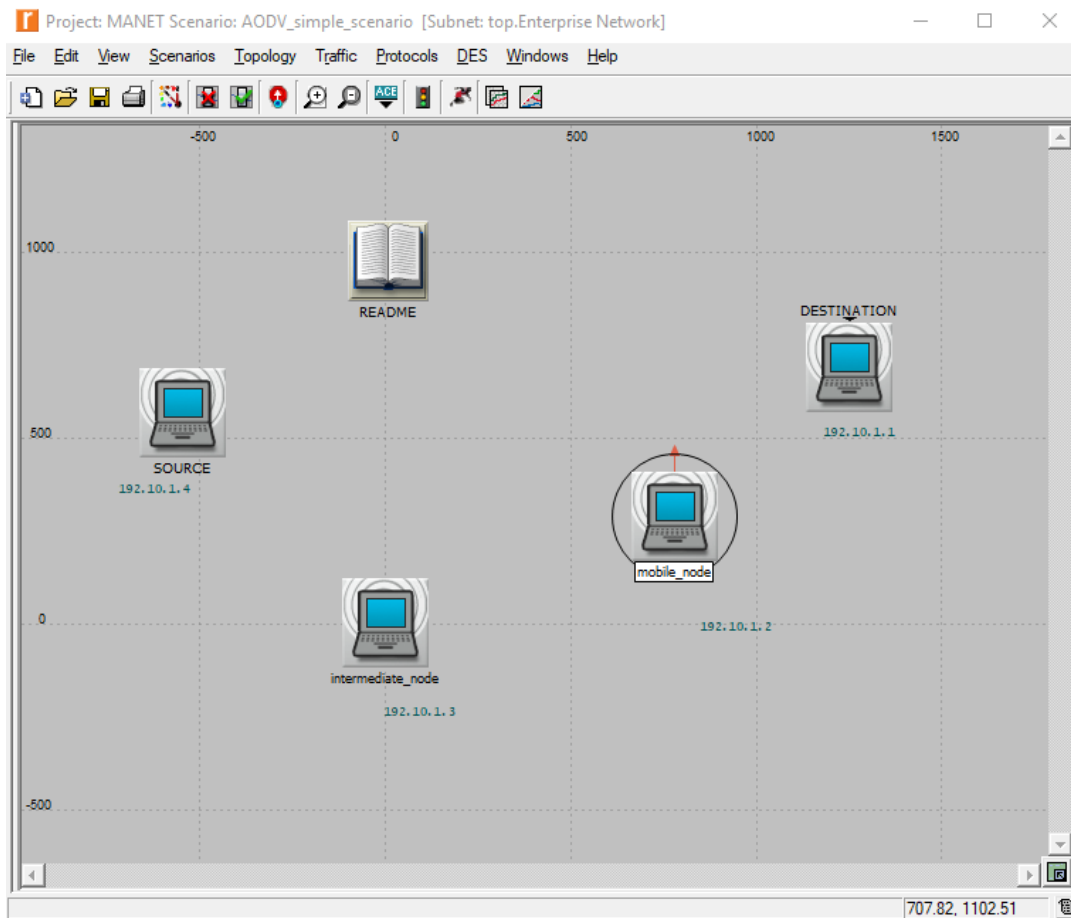
Figure 15. Validation Simulation – Riverbed Modeler

As it is shown in Figure 15, MANET model in Riverbed Modeler was configured with 4 nodes. The mobile node's distance from other nodes was randomly changed and the bit rate of the mobile node was collected for each distance. This bit rate data was statistically compared with DSO simulator node's bit rate data. Figure 16 shows DSO simulator's MANET model with 4 nodes and bit rate data. DSO simulator's MANET model was also randomly changed with various distances and the bit rate through the mobile node was measured.
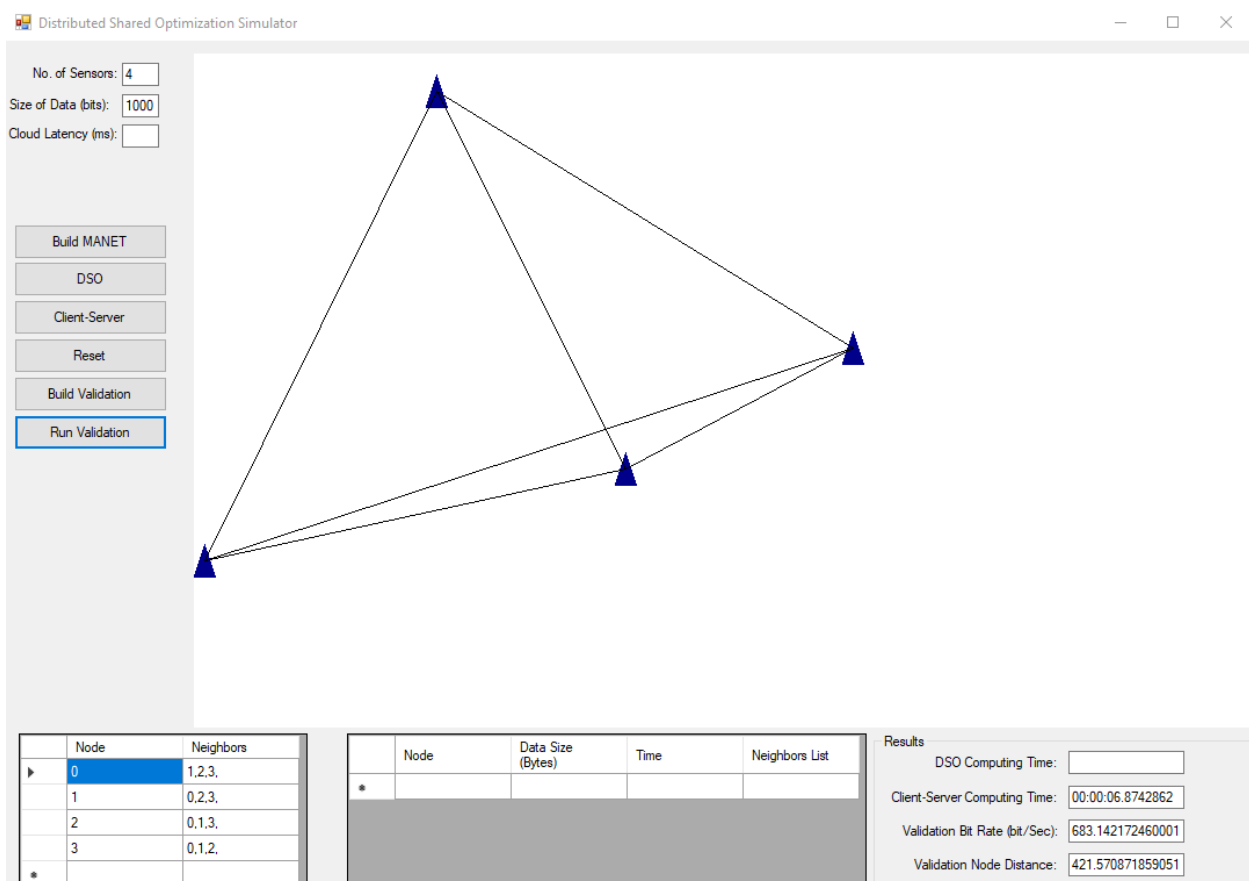
Figure 16. Validation Simulation – DSO Simulator

Bit rates from both Riverbed Modeler and DSO Simulator were collected and tabulated in Appendix A. Following hypotheses were considered to understand both the groups.

H_o = Population variances of both the groups are equal.

H_A = Population variances of both the groups are not equal.

Collected data was subjected to Independent Samples T-Test. Table 3 shows the results from T-Test.

Table 3

Independent Samples Test for Validation

		F	Sig.	t	Sig. (2-tailed)	Mean Difference	Std. Error Difference
BitRate	Equal variances assumed	0.003	0.954	-0.165	0.870	-3.275	19.80658
	Equal variances not assumed			-0.165	0.870	-3.275	19.80658

According to Table 3, the *p-value* is 0.954 which is greater than 0.05. Therefore, the null hypothesis was accepted and alternative hypothesis was rejected. That means both groups are very similar in their bit rate variances. This test validated that the DSO simulator provides the similar bit rate and network functionality as Riverbed Modeler when the configurations were made identically.

Data Collection

The network simulation tool was executed for 40 different cases and the data was recorded in Microsoft Excel. Each simulation case contained the same number of sensors and sizes of data. 12 sensors and 2500 bits of data were maintained in all 40 simulation cases. In each simulation case, sensors were placed randomly with various distances between them. Each simulation case was also ensured that one aggregator node was placed along with a given

number of sensors. Because sensors nodes were coded in blue and aggregator nodes were coded in red, it was possible to visually validate that the aggregator node was connected to at least one sensor node in the environment. This validation is important because the aggregator must be connected to one of the sensors in the environment of the client-server model in order to send the data to the cloud.

The data collection process also involved setting the cloud latency for the simulation. This research used Cedexis's (Cedexis, 2016) cloud services report for the US region to determine the value for cloud latency. Figure 17 shows the top 20 cloud platforms and their latency durations. This report was generated according to the data available on December 03rd of 2016. Using this report, this research took the median of the top 20 cloud latencies in the US and used it in this simulation. The median of top 20 cloud latencies in the US is 65 milliseconds. Therefore, this simulation was set to run based on a cloud latency of 65 milliseconds. When the simulation in the client-server model was executed, the data sent from aggregator to cloud server waited for 65 milliseconds to simulate cloud latency.
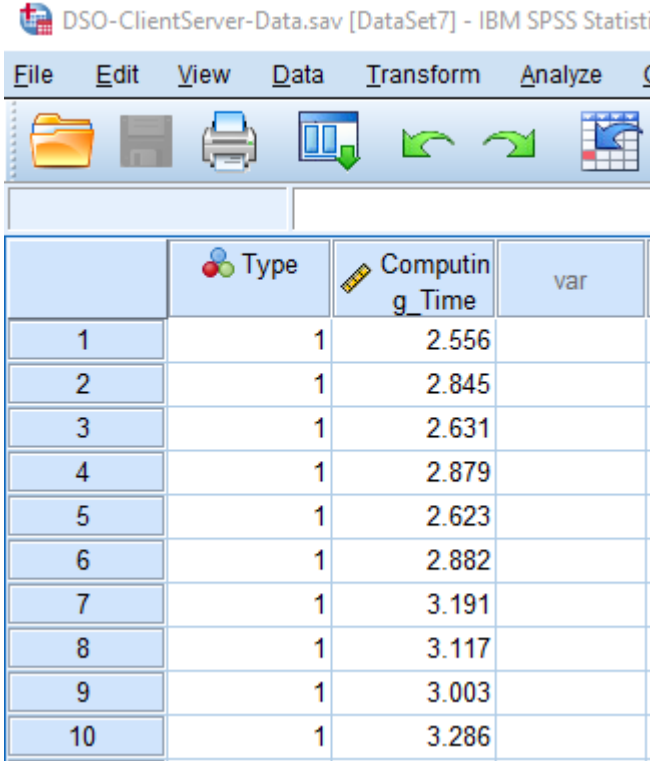
Figure 17. Cedexis Cloud Services Report – US

Upon visual validation of sensors and aggregators, both DSO and client-server models were executed and the corresponding variables of interest were recorded. Table 4 describes each variable of interest.

Table 4

Variable of Interests

Variable	Type	Measure	Range
Type (Model)	Categorical	Number	1 and 2
Sensors	Continuous	Number	4 – 35
Data Size	Continuous	Bits	600 – 6800
Computing Time	Continuous	Seconds	0.836 – 34.527

This research used IBM SPSS Statistics version 24 for data analysis. Therefore, data gathered from simulations were initially recorded in Microsoft Excel and then transferred to SPSS for analysis. Figure 18 shows the first 10 records of data from SPSS.

Figure 18. Data in SPSS

The Variable Type is a categorical variable, which denotes whether the simulation model is DSO or client-server. In order to present the variable to statistical analysis, the Variable Type was coded to 1 and 2. In that coding, 1 means DSO and 2 means client-server.

Statistical Analysis

To find the answer to the first research question, one-way analysis of variance (ANOVA) method was conducted on the variables of interests. Similarly, to find the answers to the second third, and fourth research questions, Multiple Regression was applied to the variables of interests. Independent variables of this research were Type, Sensors, and Data Size. The dependent variable of this research was Computing Time. Prior to the analysis, assumptions of ANOVA were satisfied.

CHAPTER 4

RESULTS

This chapter describes the results of the experiments conducted by this research. The

main purpose of this quantitative research study was to propose the distributed computing model

to eliminate delay from cloud latency. In that purpose, this research proposed DSO model which

uses swarm intelligence techniques to conduct computing locally. Also, this research

implemented the simulation tool to test the proposed model. Through significant number of

simulations, data regarding sensors, data size and computing time were gathered. These data

were subjected to statistical analysis in order to understand the answers to the research questions.

The simulation tool was built to run both DSO and client-server models. As input

parameters, the number of sensors and the size of data were entered. Based on these values, a

MANET platform was constructed in the tool and both DSO and client-server models were

executed. The total computing time for each model was measured. These values were recorded in

Microsoft Excel spreadsheet initially for review purposes. After careful review, collected data

were entered into IBM SPSS Statistics software version 24.

Descriptive Analysis

Following sections provide descriptive analysis for both DSO model and client-server

models.

Descriptive Analysis - DSO Model

Table 5 and Table 6 show the descriptive data for DSO model.

Table 5

DSO Model – Descriptive Analysis I

	Number of Records	Range	Minimum	Maximum
Computing Time (Seconds)	40	2.449	1.569	4.018

Table 6

DSO Model – Descriptive Analysis II

	Mean	Standard Deviation	Variance
Computing Time (Seconds)	2.87885	0.403436	0.163

In total, 40 simulations were conducted in DSO model. Each simulation was conducted

with 12 sensors. The data size was also kept at a constant level of 2500 bits.

Hence, the shortest computing time recorded for DSO model was 1.569 seconds. The

longest computing time recorded was 4.018 seconds. The range of computing time in all 40

simulations was 2.449 seconds. The mean for DSO Model's computing time was 2.87885

seconds. Also, the standard deviation and variance for computing time of DSO Model were

0.403436 and 0.163 respectively. Figure 19 describes the frequency distribution of computing time of DSO model.
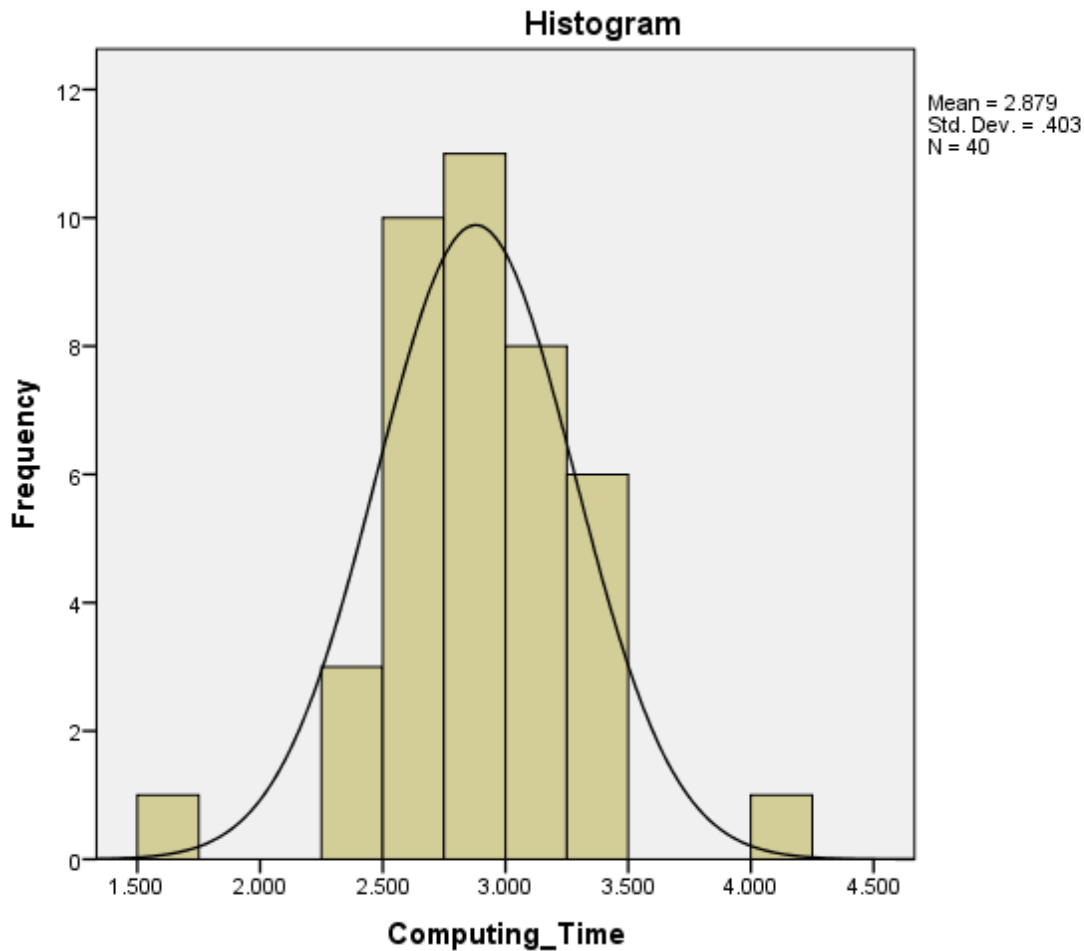
Figure 19. Frequency Distribution of Computing Time of DSO model.

Descriptive Analysis – Client-Server Model

Client-Server model was also subjected to 40 simulations. Each simulation was identical to DSO model's simulation. Table 7 and Table 8 show the descriptive data of client-server model simulation.

Table 7

Client-Server Model – Descriptive Analysis I

	Number of Records	Range	Minimum	Maximum
Computing Time (Seconds)	40	21.626	12.901	34.527

Table 8

Client-Server Model – Descriptive Analysis II

	Mean	Standard Deviation	Variance
Computing Time (Seconds)	25.75830	5.682394	32.290

The number of sensors used in these simulations was 12. The data size used was 2500 bits. These numbers are identical to DSO simulation. The main difference in descriptive statistics between DSO and client-server models was found in computing time. The shortest computing time recorded for the client-server model was 12.901 seconds. The longest computing time recorded was 34.527 seconds. The range of computing time in all 40 simulations was 21.626 seconds. The mean for client-server model's computing time was 25.75830 seconds. Additionally, the standard deviation and variance for computing time of client-server model were 5.682394 and 32.290 respectively. Figure 20 describes the frequency distribution of computing time of client-server model.
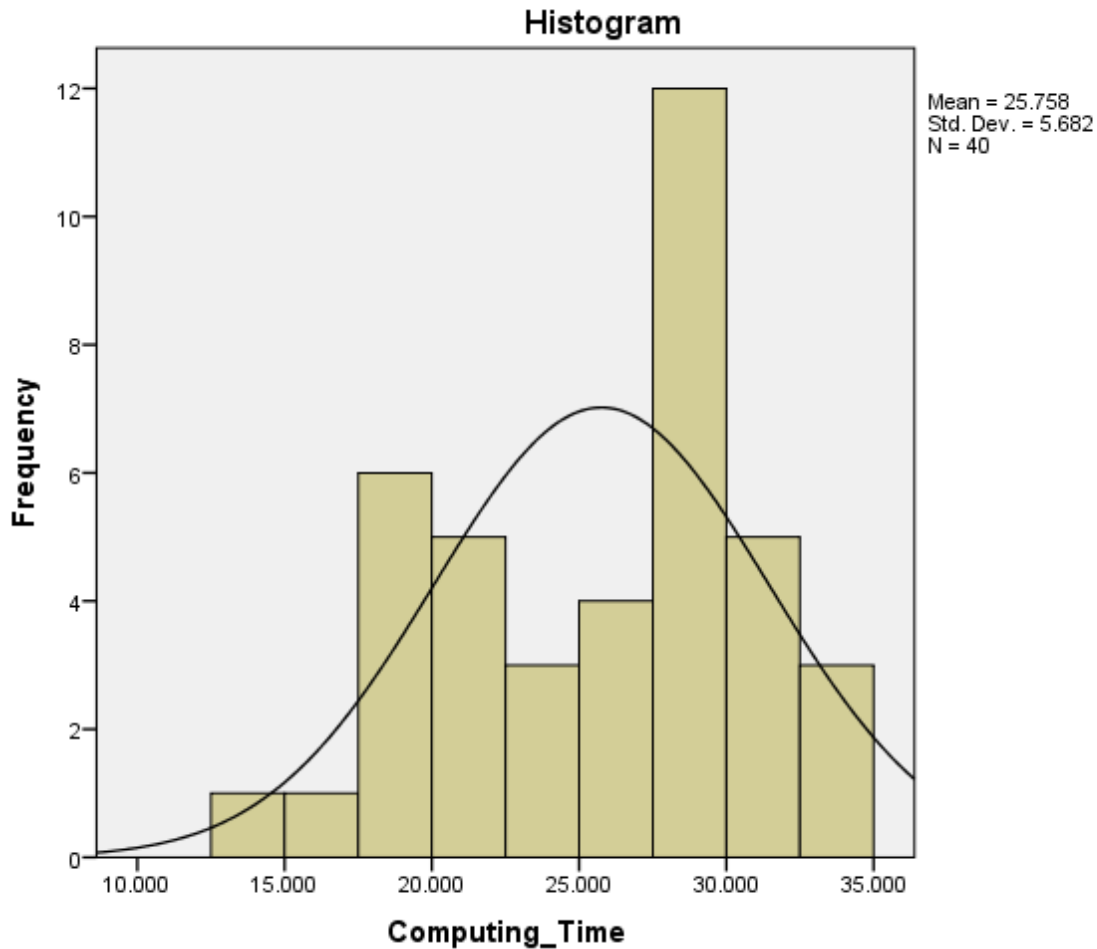
Figure 20. Frequency Distribution of Computing Time of Client-Server model.

The comparison of means from both DSO and client-server models provided a clear view of the computing time. The mean of DSO model's computing time was 2.87885 seconds and the mean of client-server model's computing time was 25.75830 seconds. This showed the time taken to compute in DSO model is significantly lower than the time taken to compute in a client-server model. Further, comparing standard deviations in computing time of both DSO and client-server model revealed that client-server model had significantly higher standard deviation and variance. That indicated client-server model's computing time readings were spread out from

mean significantly compared to the computing time in DSO model.

One-way ANOVA

Besides descriptive analysis, this research also used one-way ANOVA to determine whether there are any statistically significant differences in the means of computing time between DSO and client-server models. ANOVA is able to determine the degree of differences among multiple means without increasing the Type I error. In research, Type I error (alpha) is a false rejection of null hypothesis when the null hypothesis is actually true. Similarly, Type II error is incorrectly not rejecting a null hypothesis when the null hypothesis is actually false. This research sets the Type I error to 0.05. That means this study has 95% confident not to reject a true null hypothesis. Table 9 displays ANOVA computations for computing time between DSO and client-server models.

Table 9

ANOVA Table for Computing Time

	Sum of Squares	df	Mean Square	F	Sig.
Between Groups	10469.385	1	10469.385	645.215	.000
Within Groups	1265.642	78	16.226		
Total	11735.027	79			

In this ANOVA test, the sum of squares of computing time between groups was 10469.385. Since degrees of freedom (df) between groups is 1, the mean square was also

10469.385. Sum of squares within groups was 1265.642. In this case, the degrees of freedom (df) within groups was 78. Therefore, the mean square was 16.226 (calculated by 1265.642/78). The total sum of squares was 11735.027. Finally, this ANOVA test produced F-score 645.215 with a p-value of 0.000. (The complete p-value is 1.815E-39).

The Test of Homogeneity of Variances tests the equality of the variances in the groups. Table 10 describes the output from Test of Homogeneity of Variances. According to Table 10, a p-value of this test is 0.000 which is less than the alpha designated for this test. Therefore, the variances are not equal.

Table 10

Test of Homogeneity of Variances

Levene Statistic	df1	df2	Sig.
113.194	1	78	.000

Figure 21 shows the graphical representation of the difference between mean in DSO and client-server models. It was observed DSO model's computing time mean was more than 20 seconds less than the client-server model's computing time mean.
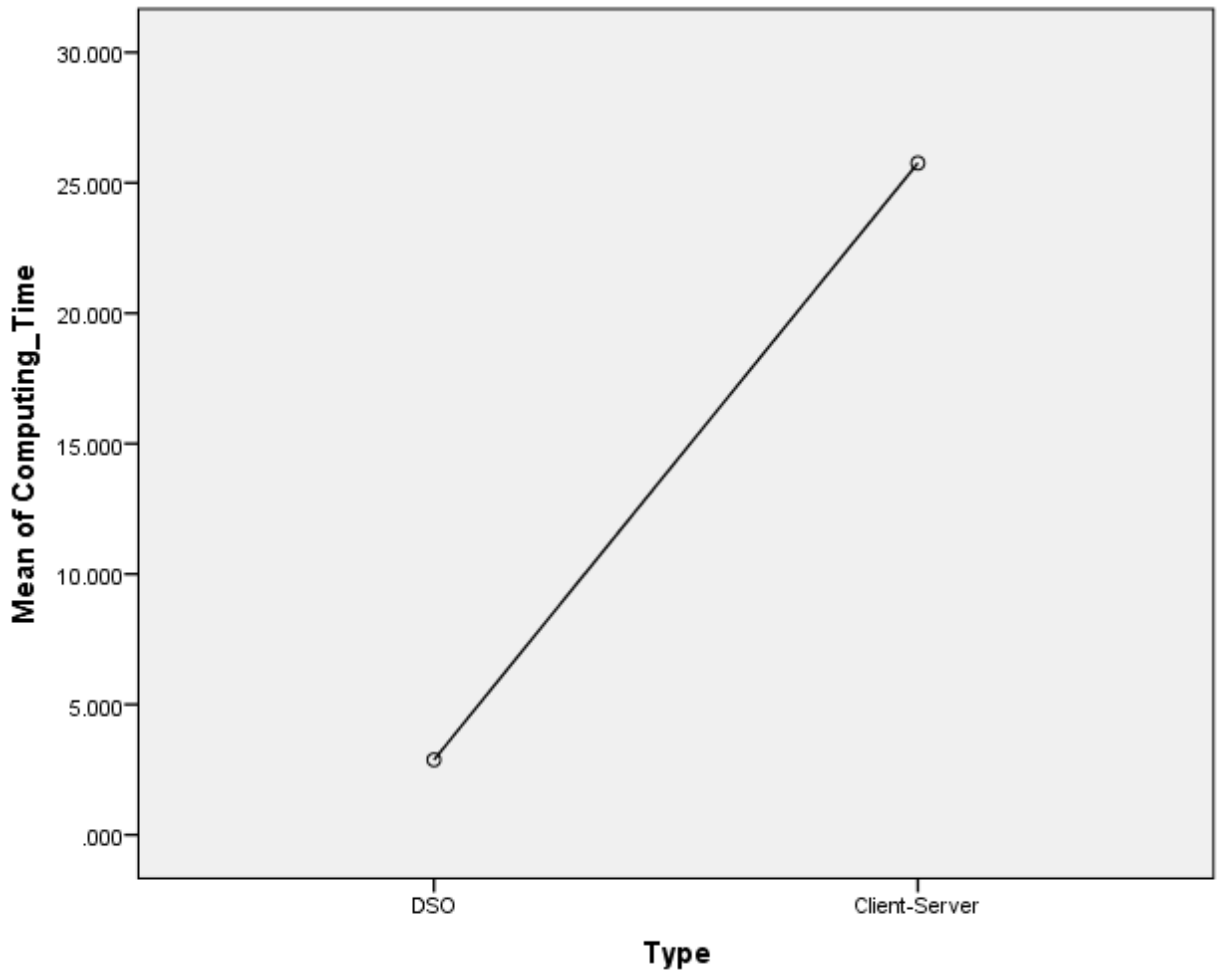
Figure 21. Means of DSO and Client-Server models

Hypotheses and Research Question Analysis

Null Hypotheses and alternative hypotheses are restated below.

H_{01}: There is no difference in computing time of the application when the proposed distributed computing model is used.

μcomputing time of DSO model $= \mu$computing time of client-server model.

H_{A1}: There is the difference in computing time of the application when the proposed distributed computing model is used.

μcomputing time with proposed distributed computing model $\neq$ μcomputing time without proposed distributed computing model.

The p-value in this research from ANOVA calculation is 1.815E-39. This p-value is significantly less than the alpha level of this research which is 0.05. Therefore, the null hypothesis is rejected and the alternative hypothesis is accepted. Thus, there is a statistically significant difference in computing time means between DSO and client-server models.

Through this statistical analysis, first research question of this research was answered. The research question and the answer were given below.

Q_1: Does the proposed swarm intelligence based Distributed Shared Optimization (DSO) model reduce the total computing time of the application?

A_1: Yes. The computing time taken by DSO model is significantly less than the computing time taken by client-server model.

Multiple Linear Regression

Second and third research questions examined the relationship of variables in DSO model. For the purpose of finding answers to these two questions, multiple regression analysis was conducted in DSO model variables. Multiple-regression technique helps to predict a variable from one or more independent variables. Therefore, multiple linear regression was conducted to predict computing time from sensors and data size in DSO model.

Sensor-Computing Time Analysis

First linear regression analysis was conducted in sensors data to predict the computing time in DSO model.

Table 11

Linear Regression results for Sensors and Computing time

R	R^2	Adjusted R^2	Standard Error of the Estimate	F Change	Sig. F Change
0.243	0.059	0.27	9.251	1.876	0.181

The multiple correlation coefficient R shows the correlation between computing time and the number of sensors. The R-value of 0.243 indicates a weaker correlation between computing time and the number of sensors. R-squared indicates how close the data are to the regression line. An R-squared value of 0.059 shows that 5.9% of data points are closer to the regression line. Adjusted R^2 is a modified R^2 that was adjusted based on the predictors and subjects in the model (Frost, 2013). The adjusted R^2 value is 0.27 which is not close to R^2 indicates there was a significant change in R^2. The standard error of the estimate shows the variability of computing time around the regression line. This measure of standard error of the estimate 9.251 is the standard deviation of the data points distributed around the regression line. The ANOVA conducted to test the significance of R^2 indicates that significance of F(1.876) is 0.181 which is greater than 0.05.

Figure 22 displays the scatter plot for depended variable computing time in DSO model. Through the visual assessment, it was observed that most of the residuals were falling between +2 and -2. That indicates that residuals were almost standardized. Another observation noted was that scatter plots were not concentrated towards a specific point.

Figure 22. Scatter Plot from Sensor-Computing Time Regression

Figure 23 shows P-P Plot of Computing Time resulted from regression with Sensors in DSO Model. Through the visual assessment, it was determined the data points were mostly distributed around the regression line. However, it was also observed that in some areas the variance was higher compared to the rest of the points in the graph.
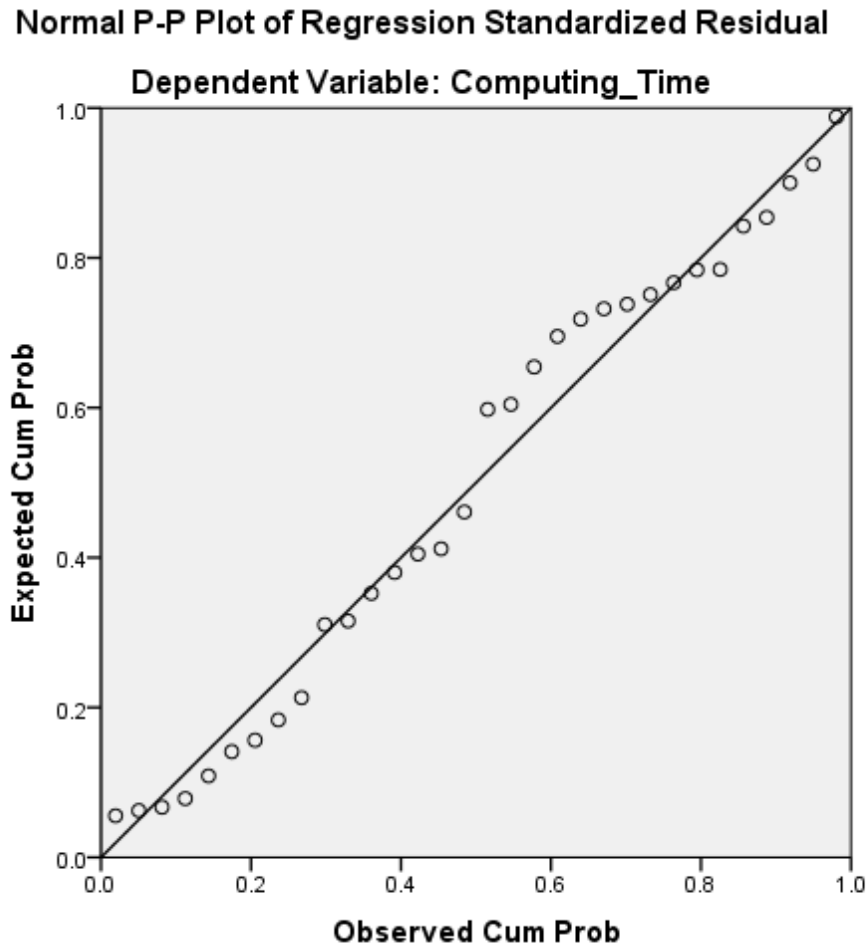
Figure 23. P-P Plot from Sensor-Computing Time Regression.

Table 12 describes the relationship between computing time and the number of sensors.

Table 12

Coefficients Table from Sensor-Computing Time Regression

	B	Standard Error	t	Sig.
Constant	2.324	0.235	9.883	0.000
Sensors	0.015	0.011	1.370	0.181

The coefficient table from regression technique revealed t = 1.370 and p = 0.181. Since this p-value is greater than 0.05, Sensors are not significant predictors to predict computing time in DSO models. Therefore, the second null hypothesis was accepted and the second alternative hypothesis was rejected.

Thus,

H_{02}: There is no relationship between the number of sensors and the computing time in distributed computing model.

was accepted and

H_{A2}: There is a relationship between the number of sensors and computing time in distributed computing model.

was rejected.

Through the evaluation of the second hypothesis, the second research question was also answered.

Q_2: How does the number of sensors affect the computing time of the distributed computing model?

A_2: There is a weaker relationship between the number of sensors and computing time in DSO model. Further, the affection of the number of sensors on computing time cannot be stated in an equation.

Further, this research conducted an extensive experiment to understand the limitation of sensors on DSO mechanism. In that procedure, sensors were added incrementally to the simulation model until the tool breaks. The computing time from each iteration was recorded and tabulated in Appendix A. The recorded data were analyzed to understand the impact of sensors. Figure 24 shows the graph obtained from the sensors incremental process.
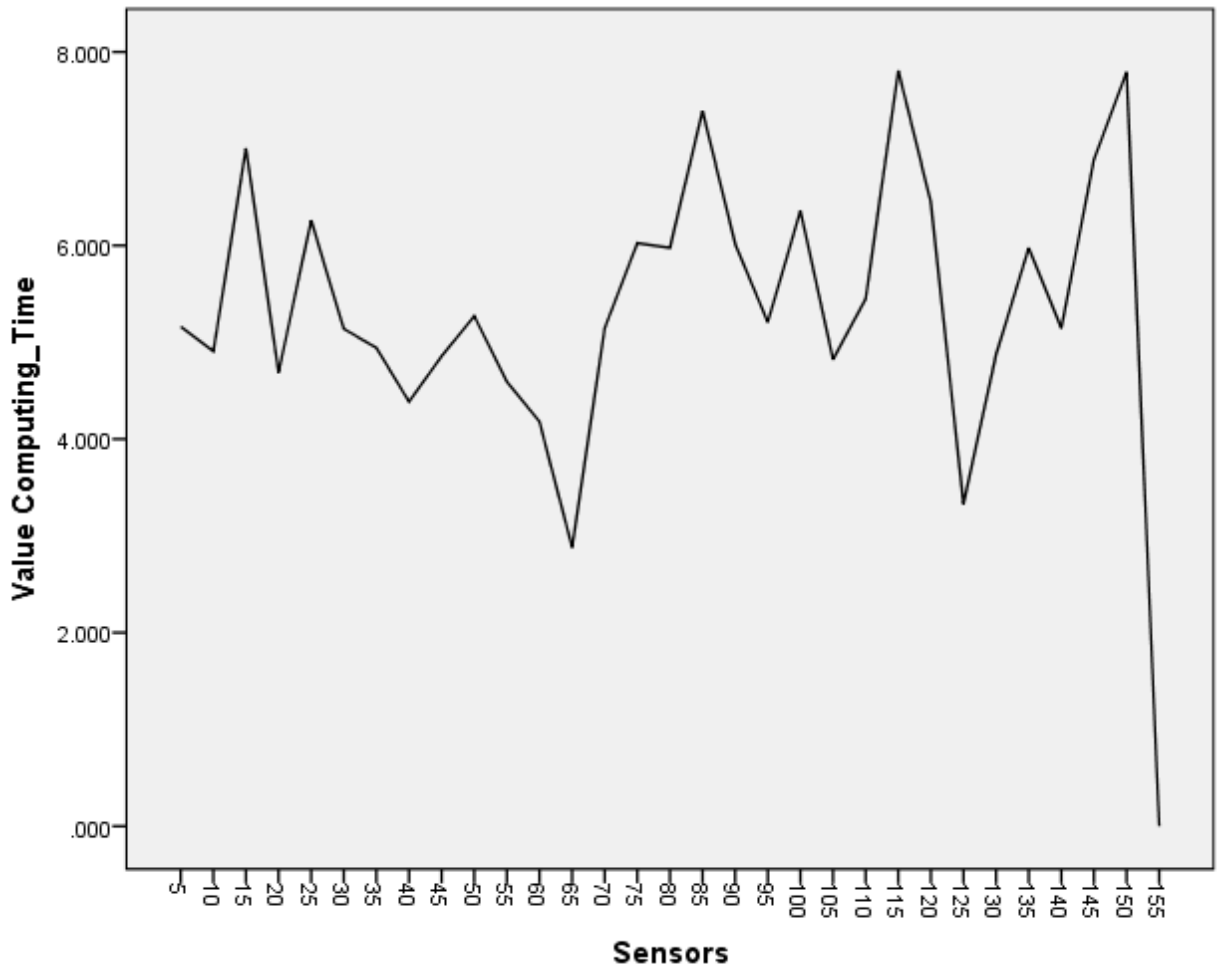
Figure 24. Sensor Limit on Simulation Tool.

According to the graph obtained in Figure 24, the computing time ranged between 2.5 seconds and 8 seconds regardless of the number of sensors applied. During this experiment, the data size was kept at a constant level of 5000 bits. This graph also explained the effect of sensors on DSO mechanism. There was no pattern observed between the number of sensors and DSO mechanism. When the number of sensors was increased systematically, computing time did not show a steady increase or decrease. This procedure proved there was no observable pattern between sensors and computing time in DSO mechanism.

Another observation from this experiment was the breaking point of the simulation tool. When the number of sensors reached 155, the simulation tool was unable to continue the process. It is not the limitation of DSO mechanism. This is a limitation of the simulation tool which heavily depends on the hosting computer's resources. During this specific experiment, the hosting computer's resources reached the maximum limit and that restricted the simulation tool not to proceed beyond 155 sensors.

Data Size-Computing Time Analysis

Second linear regression analysis was conducted in data size to predict the computing time in DSO model.

Table 13

Linear Regression results for Data Size and Computing time

R	R^2	Adjusted R^2	Standard Error of the Estimate	F Change	Sig. F Change
0.985	0.969	0.968	0.306467	951.598	0.000

Correlation coefficient R's value of 0.985 indicates a strong correlation between computing time and data size. R-squared indicates how close the data are to the regression line. An R-squared value of 0.969 shows that 96.9% of data points are closer to the regression line. The adjusted R^2 value is 0.968 which is very close to R^2. This indicates there was no significant change in R^2 based on the predictors and subjects in the model. The measure of standard error of the estimate 0.306467 is the standard deviation of the data points distributed around the regression line8]8. The ANOVA conducted to test the significance of R^2 indicates that significance of F=951.598 is 0.000 which is less than 0.05.

The scatter plot described in Figure 25 shows the data points from the regression analysis on data size and computing time in DSO model. The visual assessment on the plot revealed that residuals were standardized because all the residuals were falling between +2 and -2. However, it was also observed that data points on the scatter plot were concentrated towards -2. Also, the data points were scattered and spread towards +2.
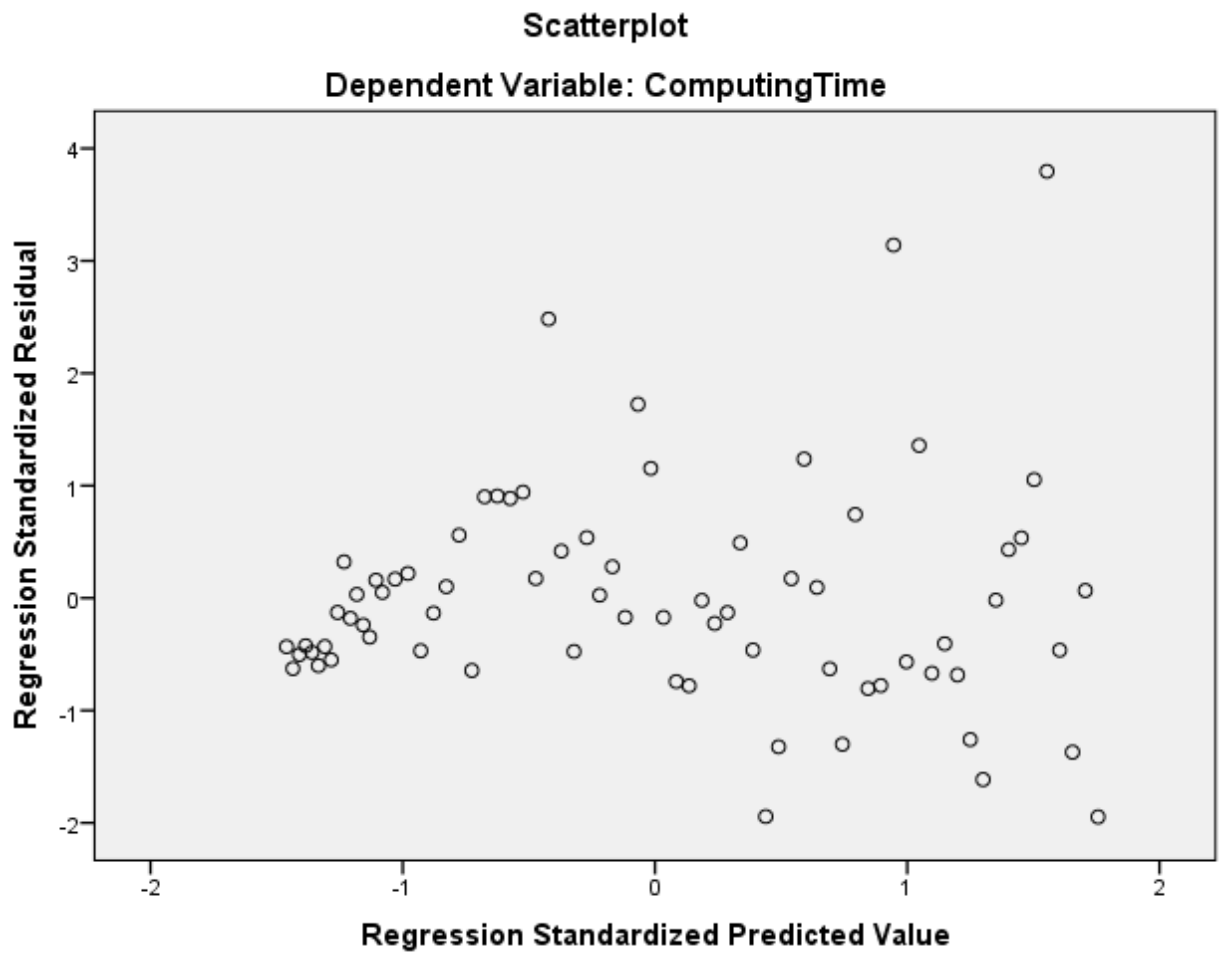
Figure 25. Scatter Plot from Data Size-Computing Time Regression

Figure 26 shows P-P Plot of Computing Time resulted from the regression with data size in DSO Model. The visual assessment showed the data points were distributed around the

regression line. However, it was also observed that some data points had greater variances from

the regression line.

Normal P-P Plot of Regression Standardized Residual

Dependent Variable: ComputingTime
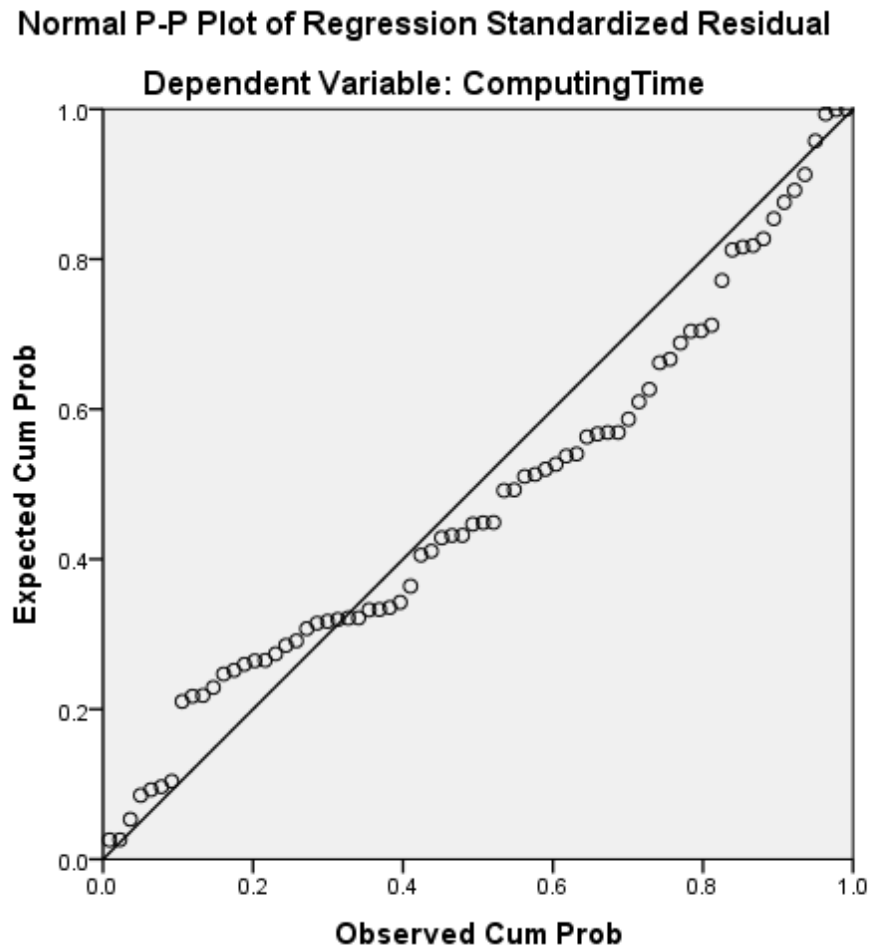
Figure 26. P-P Plot from Data Size-Computing Time Regression.

The relationship between computing time and data size was described in Table 14.

Table 14

Coefficients Table from Data Size-Computing Time Regression

	B	Standard Error	t	Sig.
Constant	0.501	0.121	4.130	0.000
Data Size	0.001	0.000	30.848	0.000

The coefficient table from regression technique revealed $t = 30.848$ and $p = 0.000$. Since this p-value is less than 0.05, data size was a significant predictor to predict computing time in DSO models. Therefore, the third null hypothesis was rejected and the third alternative hypothesis was accepted.

Thus,

H_{03}: There is no relationship between the size of data being processed and computing time in distributed computing model.

was rejected and

H_{A3}: There is a relationship between the size of data being processed and computing time in distributed computing model.

was accepted.

Through the evaluation of the third hypothesis, the third research question was also answered.

Q_3: How does the size of data being processed affect the computing time of the distributed computing model?

A_3: There is a strong relationship between the data size and computing time in DSO

model. Further, the affection of data size on computing time can be stated as

Computing Time = 0.001 x (Data Size) + 0.501

Sensors-Data Size-Computing Time Analysis

This research also conducted multiple regression on all three variables of DSO namely

sensors, data size and computing time. The purpose was to understand the effect of both sensors

and data size together on computing time of DSO model.

Table 15

Multiple Regression results for Sensors, Data Size and Computing time

R	R^2	Adjusted R^2	Standard Error of the Estimate	F Change	Sig. F Change
0.958	0.918	0.916	0.625979	337.547	0.000

The R-value of 0.958 indicated a strong correlation between computing time and the

combination of the number of sensors and data size. The standard error of the estimate 0.625979

is relatively higher compared to data size only regression model. However, this value is less than

the value from sensor only regression model. The ANOVA conducted to test the significance of

R^2 indicates that significance of F(337.547) is 0.000 which is less than 0.05.

The scatter plot described in Figure 27 shows the data points from the regression analysis

on sensors, data size and computing time in DSO model. The visual assessment on the plot

revealed that residuals were standardized because all the residuals were falling between +2 and -

2. Another observation made was the pattern of data points where more data points were
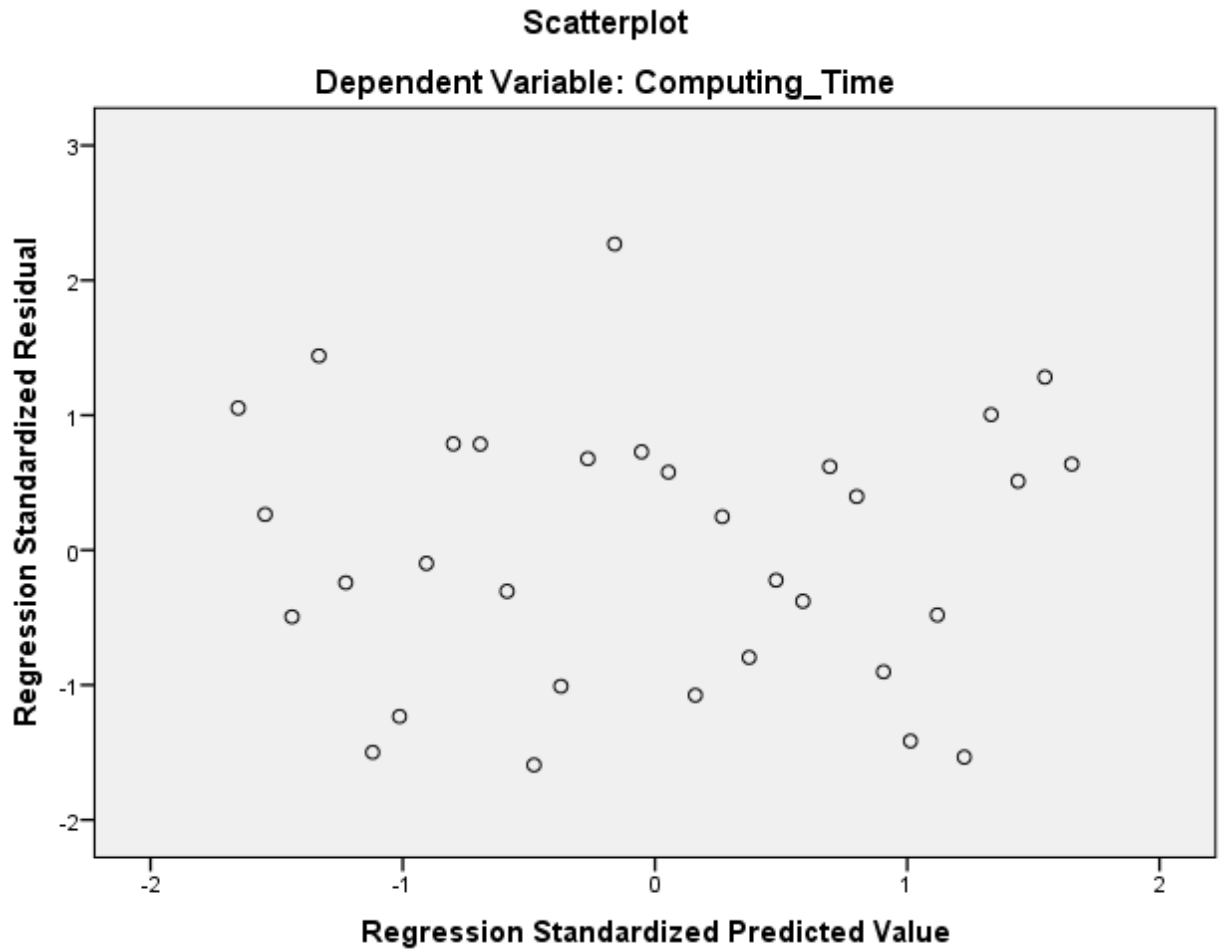
concentrated on the top part of the plot.
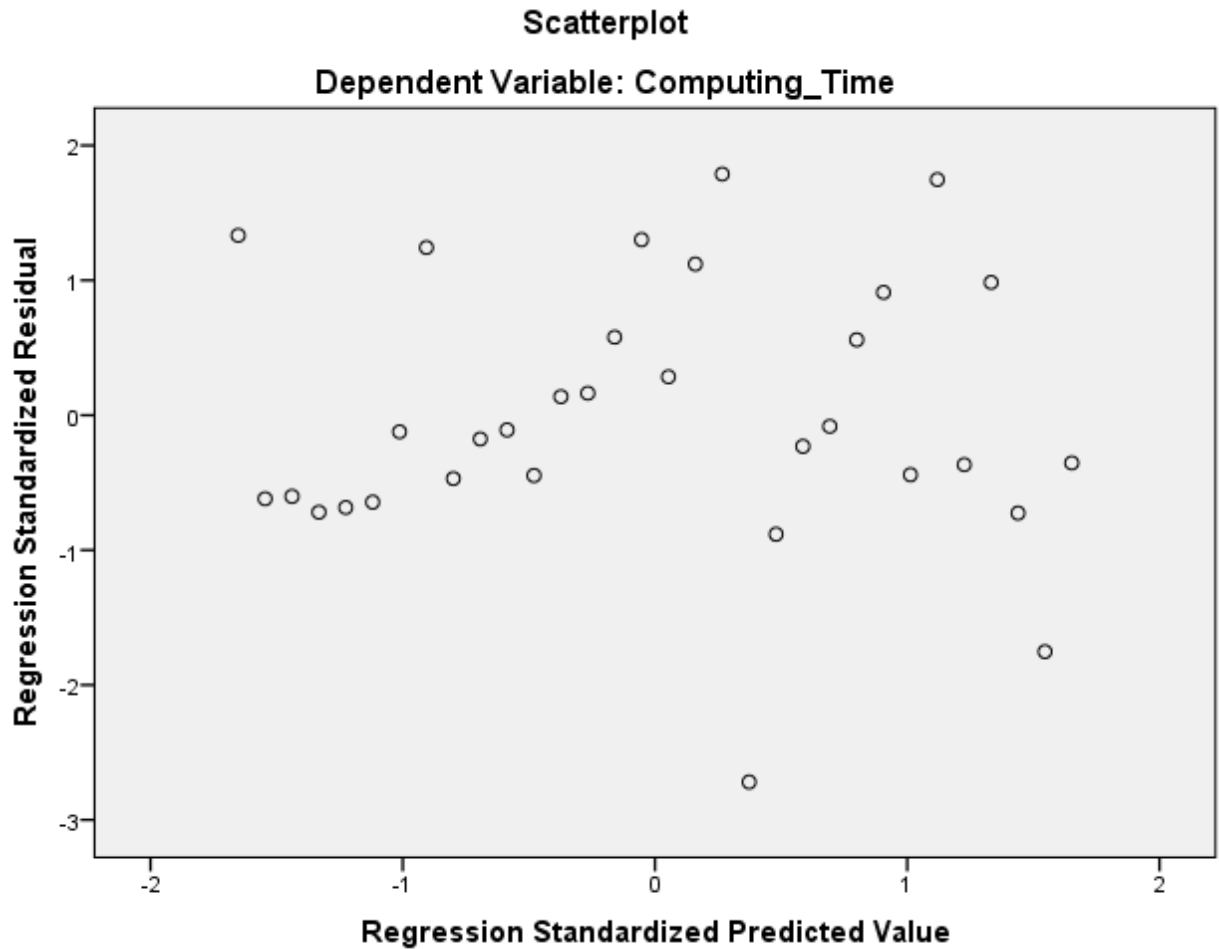
Figure 27. Scatter Plot from Sensors-Data Size-Computing Time Regression

Figure 28 shows P-P Plot of Computing Time resulted from the regression with sensors and data size in DSO Model. The visual assessment showed the data points were distributed around the regression line. This pattern was somewhat similar to the P-P Plot obtained from data size regression.
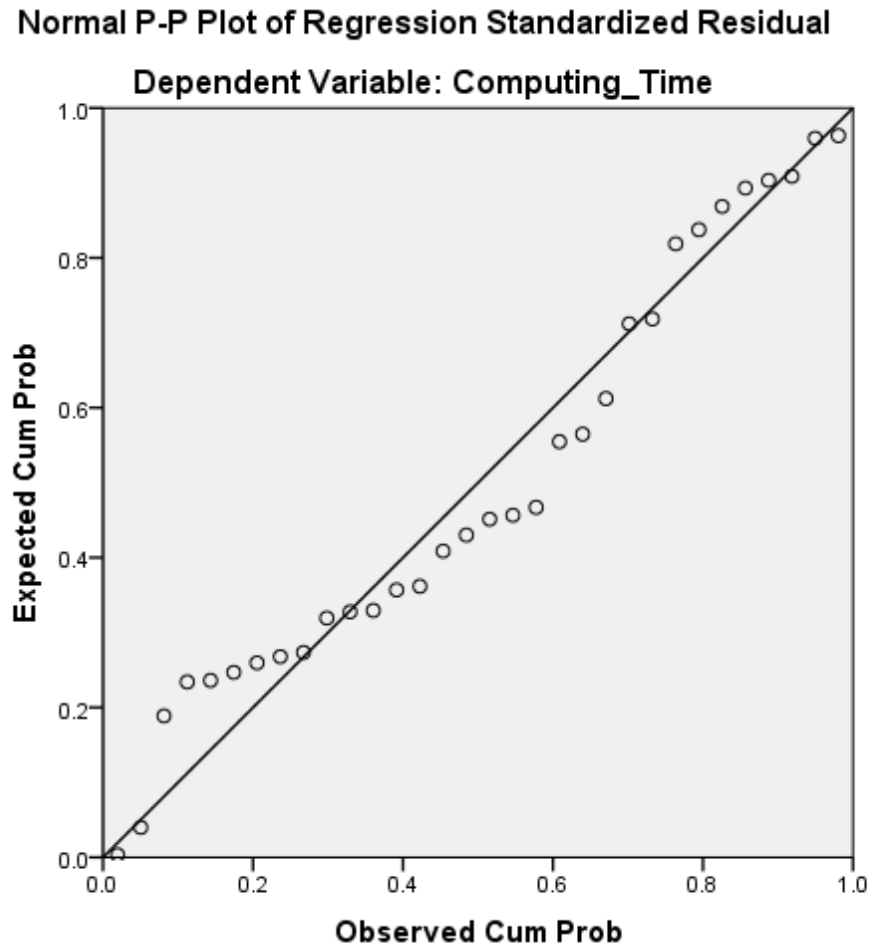
Figure 28. P-P Plot from Sensors-Data Size-Computing Time Regression

The overall model showed a strong relationship. However, coefficient table provided more details to understand the impact on computing time by the combination of sensors and data size together.

Table 16

Coefficients Table from Sensors-Data Size-Computing Time Regression

	B	Standard Error	t	Sig.
Constant	0.473	0.248	1.908	0.066
Data Size	0.001	0.000	18.372	0.000

The coefficient table from regression technique revealed for data size t = 18.372 and p = 0.000. This p-value is less than 0.05 alpha value. Hence, the coefficient table removed sensors from the final table because data size is highly correlated compared to sensors. If that happens, SPSS automatically drops the lesser correlated variable. That shows data size is a more accurate predictor of computing time compared to sensors. In other words, when both variables were applied in prediction, the regression model indicated a strong relationship. This relationship was obtained by removing sensors from the model. However, the strength of this relationship is relatively less compared to the strength of the relationship when data size alone was applied. Therefore, the fourth null hypothesis was rejected and the fourth alternative hypothesis was accepted.

Thus,

H_{04}: There is no relationship with computing time by the combination of sensors and the size of data being processed together in distributed computing model.

was rejected and

H_{A4}: There is a relationship with computing time by the combination of sensors and the size of data being processed together in distributed computing model.

was accepted.

Through the evaluation of the fourth hypothesis, the fourth research question was answered.

Q_4: How do both sensors and the size of data being processed together affect the computing time of the distributed computing model?

A_4: The combination of sensors and the size of data being processed does statistically affect the computing time in distributed computing model. Although, sensors were not used as the predictor, the combination of both sensors and data sizes has an impact on DSO mechanism. Further, the affection by the combination of data size and sensor on computing time can be stated as

Computing Time = 0.001 x (Data Size) + 0.473

CHAPTER 5

CONCLUSIONS AND RECOMMENDATIONS

This research proposed a distributed computing model to reduce the delay caused by cloud latency in IoT applications. Also, this research prepared a simulation program for the distributed model and investigated how various parameters affect the performance of the model. This chapter focuses on the research conclusions and limitations and recommendations for future research in this area.

Conclusions on the Research Hypotheses and Questions

This research primarily raised four research questions to find the performance of DSO model in reducing computing time. Each research question was supported by a hypothesis to conduct research analysis. The following section describes each research question, supported hypothesis, and research conclusion.

Research Question 1

Does the proposed swarm intelligence based Distributed Shared Optimization (DSO) model reduce the total computing time of the application?

Null Hypothesis 1

There is no difference in computing time of the application when the proposed distributed computing model is used.

Alternative Hypothesis 1

There is a difference in computing time of the application when the proposed distributed computing model is used.

Conclusion 1

ANOVA analysis indicated that there was a significant difference in computing time between DSO and client-server models. Further, the mean of DSO model's computing time was 2.87885 milliseconds and the mean of client-server model's computing time was 25.75830 milliseconds. This was 88.82% reduction in computing time by DSO when only sensors and data size were considered as influencing factors. Therefore, DSO model reduces the computing time of IoT significantly.

Research Question 2

How does the number of sensors affect the computing time of the distributed computing model?

Null Hypothesis 2

There is no relationship between the number of sensors and the computing time in distributed computing model.

Alternative Hypothesis 2

There is a relationship between the number of sensors and computing time in distributed computing model.

Conclusion 2

Linear regression analysis on the sensor data of DSO model indicated no relationship between computing time and sensor count. Both the multiple regression correlation coefficient (R) value and the coefficient of determination (R^2) value revealed a very weak correlation between computing time and the number of sensors. Also, when the sensors and computing time

values were analyzed in a graph, there was no clear pattern was revealed. Further analyzing the coefficient table asserted number-of-sensors is not a reliable predictor of computing time in IoT environment.

Research Question 3

How does the size of data being processed affect the computing time of the distributed computing model?

Null Hypothesis 3

There is no relationship between the size of data being processed and computing time in distributed computing model.

Alternative Hypothesis 3

There is a relationship between the size of data being processed and computing time in distributed computing model.

Conclusion 3

Linear regression analysis on the data sizes of DSO model indicated a strong linear relationship between computing time and data sizes. Both the multiple regression correlation coefficients (R) value and the coefficient of determination (R^2) value revealed a strong correlation between computing time and size of the data. Further analyzing the coefficient table asserted data size is a reliable predictor of computing time in IoT environment. The impact of data size on the DSO model can be expressed by the equation shown below.

Computing Time = 0.001 x (Data Size) + 0.501

Research Question 4

How do both sensors and the size of data being processed together affect the computing time of the distributed computing model?

Null Hypothesis 4

There is no relationship with computing time by the combination of sensors and the size of data being processed together in distributed computing model.

Alternative Hypothesis 4

There is a relationship with computing time by the combination of sensors and the size of data being processed together in distributed computing model.

Conclusion 4

Multiple regression analysis on both the data sizes and the number-of-sensors of DSO model indicated a strong linear relationship between computing time and the combination of sensors and data sizes. Both the multiple regression correlation coefficient (R) value and the coefficient of determination (R^2) value revealed a strong correlation between computing time and the combination of both sensors and size of the data. However, analyzing the coefficient table revealed this strong relationship was obtained by considering only data size values. Because the relationship between data size and computing time was very strong, the relationship from sensors was negligible. Overall, the combination of number of sensors and size of the data can be used as predictors of computing time even though data size's influence is much greater than the sensor's influence.

Discussion of Significant Results

The demand for IoT applications is increasing exponentially in the recent past. As the demand increases, requirements for effective IoT applications also grow. One of the growing requirements in IoT ecosystem is the need for short response time for time sensitivity applications. Traditionally, data from IoT sensors are sent to cloud for processing. The response

from the cloud is sent back to IoT application. Cloud latency plays a major role in determining the response time for IoT applications. In North America, the average cloud latency is 65 milliseconds. In time-sensitive applications such as IoT devices in self-driving cars, every millisecond is crucial for decision-making. Therefore, there is a need for reducing the delay caused by cloud latency in IoT applications.

This research proposed distributed computing in IoT environment to reduce the delay caused by cloud latency. Through distributed computing, data processing can be conducted in the IoT environment instead of sending data to the cloud. In order to construct distributed computing, this research proposed a swarm intelligence based distributed system called DSO. In DSO model, IoT sensors function in a way similar to how biological agents such as ants, birds, and bees function in a swarm. DSO specifically modeled birds flocking to simulate distributed computing. Bird flocking follows three basic procedures called alignment, cohesion, and separation. Using these three guidelines, birds are able to fly towards a target destination as a group. DSO uses these three guidelines to process data as a group and reaches the goal of completing the computing. This research also proposed the algorithm for DSO that can be employed in IoT environments.

In order to test the proposed DSO model, this research built a simulation program that simulates a MANET based IoT environment. In this simulation program, both DSO and client-server models were employed and functionality was recorded. Through the statistical analysis using gathered data, this research identified the computing time was reduced when DSO model was employed. The reduction of computing time was 88.82%. This is a significant reduction where client-server model's computing time was reduced to below half when DSO model was employed. By conducting computing in the local environment, DSO did not use cloud resources.

This eliminated the need to experience cloud latency. The processing burden of data is shared by IoT sensors in the local environment. Therefore, DSO allows parallel processing which also reduces the time significantly. Because of the technological advancements in the hardware, it was possible to build sensors that could do the complicated processing. From the cost perspective, it is also affordable to construct sensors that have the high processing power.

Further, this research identified the relationship between sensors, data sizes and computing time in DSO. The number of sensors did not influence the computing time. Based on conventional knowledge, there should be a relationship between the number of sensors involved and processing data. However, this research did not identify any relationship between sensors and computing time in DSO. Because of this revelation, this research assumed there was another factor related to the sensors which was influencing the computing time severely. After analyzing the network setups, this research assumed the distance between the sensors could be the factor that was influencing the computing time. However, the distance between the sensors was not considered as a factor in this research. Therefore, this research recommends the distance between sensors must be considered in the future researches of DSO.

Hence, this research identified a strong relationship between data size and computing time. The relationship between data size and computing time is given below.

Computing Time = 0.001 x (Data Size) + 0.501

A strong relationship was identified between computing time and data size. This equation allowed to estimate the computing time based on data required to process at a given time. Through this equation, researchers could identify the computing time for various data sizes in order to understand the processing scope of DSO.

Interestingly, this research also found that the combination of both sensors and data size can be used as an estimator to find computing time but it was mainly influenced by data size. Because the influence of data size is much greater than the influence by the sensors, the resulting impact can only consider data size. Further research is required to understand more about this combination.

Limitation of the Research

One of the main limitation of this research is the simulation tool's scope in MANET construction. The simulation tool was designed to build the MANET nodes in a two-dimensional (2D) panel. Thus, the nodes were randomly placed based on x and y-axis distances. Also, the neighboring nodes connections were established based on x any y-axis distances. Therefore, the data gathered from simulations were based on x and y-axis distances only. This could have limited the calculations. It would be beneficial to expand the simulation tool to construct MANET nodes in three-dimensional (3D) with x, y, and z-axes to denote the distances.

Another limitation of this study is the hosting computing of the simulation tool. Even though the simulation was designed to conduct parallel processing for DSO, the efficiency of the parallel processing depends on the hosting computer's resources. A high processing powered CPU with a large volume of RAM would provide more accurate results. This research was conducted in a 2.8 GHz Processor with 8 GB RAM. This configuration was enough to run the simulation for this study. However, in the event of conducting more simulation cases for a large volume of data, it is desirable to add more hardware resources to the simulation hosting computer.

The final limitation of this research is the adaptability from simulation to real-time environments. The data and calculations were proving the efficiency of DSO theory in a simulated environment. The results could be slightly different when MANET is constructed in a real environment to conduct the research. The results from the real environment could be affected by external factors such as weather. The scope of this research did not include the external factors that could affect the research calculations. Future researches can be conducted to measure the effects of external factors to DSO.

Applications of DSO

The main application of DSO will be in the technology of self-driving cars. In the functionality of self-driving cars, there are thousands of sensors around the vehicle to understand the environment. The gathered data about the environment are often sent to cloud systems to interpret for reliable information. This information is critical to making the decisions in the self-driving vehicle. The requests made to the cloud systems are subjected to cloud latency. Since cloud systems could be anywhere in the globe, there is not much control in reducing the cloud latency by sending requests only to those servers that are closer to the vehicle. In the present infrastructure of cloud architecture, it is almost impossible to reduce the cloud latency by limiting the servers the request needs to hop to get to the target server. Any delay in certain response could lead to the major catastrophic event in self-driving vehicles.

In these situations, employing DSO in self-driving vehicles could reduce the latency tremendously. By distributing the processing to the neighboring nodes, self-driving vehicles could avoid sending requests to the cloud. Even though some requests need absolute access to the cloud, most of the requests can be diverted to process in the local environment. Technology

advancements are producing sensors with higher processing powers. Therefore, distributing the workload among IoT sensors is becoming more possible.

Another application of DSO is enabling self-sustainable networks that are independent of cloud access. At the time of growing concerns about privacy, there is a need to run applications without accessing many interactions with the cloud. By employing DSO, even home networks with IoT can share most of the processes within the network and reserve cloud access for absolutely necessary actions. As IoT implementations are growing in the smart city and smart grid solutions, the application of not depending too much on cloud servers off-loads huge network traffic off from the infrastructures.

Risks in DSO Mechanism

The main concern in implementing DSO in IoT applications is the over usage of resources in sensors. The primary task of the sensors is sensing data but not computing data. Many resources such as memory and computing power are allocated for sensing purpose only. However, DSO allocates some part of memory and computing power to compute the data. The concern is whether this allocation would affect the performance of sensor's primary task. With the advancement of sensor technology, newer sensors are produced with high memory and power. The cost of the sensors also going down as the technology of the sensors improves. Therefore, the concern about resource allocation will not be an issue in the future implementations. Another similar concern is the power consumption of numerous sensor devices. If the overall power consumption of all the sensors exceeds a certain level, the IoT environment could be subjected to power interruption. However, these sensors consume the

relatively very small amount of power compared to large cloud servers. Therefore, power consumption will be very less when computing is conducted through IoT sensors.

Another risk in DSO mechanism is the handling of sensitive data. In the cloud-server model, the sensitive data needs to travel only to the cloud server from a device. Therefore, the security risk is only in one area. In DSO mechanism, the data is distributed and processed in each sensor. Therefore, the risk of being breached is very high and that risk is scattered all around the IoT environment.

Restricting Factors of DSO

DSO mechanism heavily depends on the physical network obstructions. Since IoT environment could be scattered in any geographical area, the sensors are subjected to physical interruptions such as natural disasters and accidents. In the client-server model, most of the time the data resides in the cloud. Therefore, the data is safe when natural disasters occur. In DSO model, the data is scattered around IoT sensors in the physical environment. This characteristic exposes the continuity of DSO function to greater risk.

Another restricting factor is the distance between sensors. Since this is a MANET based IoT environment, the sensors are mobile. Due to the constant movement of sensors, the data transfer between sensors could vary in time. Also, when the distance exceeds the reachability of other IoT sensors, the data transmission could be interrupted.

Security of IoT sensors also restricts certain environments' ability to participate in DSO. Because DSO depends on the sharing characteristic of each sensor in the network, if individual sensors are programmed not to participate in data process, DSO of that environment will be greatly affected. Since IoT development is in its early stages, not many sensors are highly

secured. However, this pattern will change when IoT applications become more common. At that time, sensors will have very high security to protect data breach. Even passing through security levels will add another layer of time delay in overall computing time.

The final restricting factor is the type of sensors used in DSO environment. The advanced sensors have high memory and processing power while most of the sensors in the current market do not have such resources. The success of DSO mechanism depends on the ability of sensors to process a large volume of data. This requirement requires high-powered advanced sensors. Even though increasingly advanced sensors are produced at a lower cost, the current market is still below the expectation.

Recommendations for Future Research

This research provided significant implications for the future studies. Since this research simulation of DSO is limited to two-dimensional setup, the main recommendation for future researchers is to enhance the simulation to three-dimensional setup. It is important to see how three-dimensional simulation setup impacts the results in DSO's functionality. By focusing on three-dimensional simulation, the results would be much closer to real IoT environment. Besides the three-dimensional enhancement, the simulation tool can be expanded in the following ways to examine DSO theory in a more closer way.

- Simulation tool's minimum data size can be allowed to set by the user. In that way, various simulations with different minimum data size can be tested.
- Expanding the simulation tool to run on multiple computers in a distributed method. This enhancement will bring the simulation tests closer to real-world IoT environments.

- Connecting the simulation tool with big data processing engines such as Spark to measure the time taken for big data queries.

- Tool output can be enhanced to provide graphs to understand the patterns.

Apart from the enhancements to the simulation tool, this research can be modified to use real-world environment instead of using simulation environment. The simulation environment was built to mimic real-world environment. However, some external factors such as weather were not accounted for transmission happening from remote locations. Therefore, it is recommended to conduct this research by reading data from actual sensors built on MANET. Also, this research strongly recommends including security measures in the future research designs. The security measures may change the computing time data significantly.

Summary

This research began the task of proposing a distributed computing model for IoT to reduce the computing time. Cloud latency was the main cause for delaying computing time. This research proposed DSO as a framework for reducing the computing time by conducting computing among IoT sensors. As the proof of this theory, this research conducted an experimental design to test DSO theory on MANET. A simulation tool was built to assist the experiment. The gathered data from the experiment were subjected to statistical analysis. From the statistical analysis, this research concluded that there is a significant computing time reduction in DSO model compared to the traditional cloud structure such as client-server model. Based on this conclusion, this research proposes the distributed computing among IoT devices as an alternative to depending on cloud servers for all the actions. The massive amount of data

produced by IoT sensors can be processed in the local environment instead of sending to the cloud server. This approach will produce very effective and faster computations in time-critical devices such as self-driving cars. As a future recommendation, big data processing engines can also be implemented in the local environment to speed up this process.

REFERENCES

Bali, M.S., and Khurana, S. (2013). *Effect of Latency on Network and End-User Domains in Cloud Computing*. Paper presented at 2013 International Conference on Green Computing, Communication and Conservation of Energy (ICGCE).

Bellavista, P., Cardone, G., Corradi, A., and Foschini, L. (2013). Convergence of MANET and WSN in IoT Urban Scenarios. *IEEE Sensors Journal*. 13(10). 3558-3567.

Cardozo, A., Yamin, A., Xavier, L. Souza, R., Lopes, J., and Geyer, C. (2016). *An Architecture proposal to distributed sensing in Internet of Things*. Paper presented at 2016 1st International Symposium on Instrumentation Systems, Circuits, and Transducers (INSCIT). Retrieved from IEEE Xplore Digital Library. (Accession Number 7598208).

Caro, G., Ducatelle, F. and Gambardella, L. (2008). *Ant Colony Optimization for Routing in Mobile Ad Hoc Networks in Urban Environments*. (Technical Report No. IDSIA-05-08). Manno, Switzerland: Dalle Molle Institute for Artificial Intelligence. Retrieved from http://people.idsia.ch/~gianni/Papers/IDSIA-05-08.pdf

Cedexis, (2016). *CDN & Cloud Performance by Country*. Retrieved from https://www.cedexis.com/get-the-data/country-report/

Chen, Y. (2012). *Challenges and Opportunities of Internet of Things*. Retrieved from IEEE Xplore Digital Library. (Accession Number 12592062)

Chien, S., Chan, W., Tseng, Y., Lee, C., Somayazulu, V. and Chen, Y. (2015). *Distributed computing in IoT: System-on-a-chip for smart cameras as an example*. Paper presented at 20th Asia and South Pacific Design Automation Conference. Retrieved from IEEE Xplore Digital Library. (Accession Number 14984378).

Chung, Y. (2005). *A Comparison Study of Particle Swarm Optimization Algorithms in Data Clustering.* (Master's Thesis). California State University, Long Beach, California. Retrieved from ProQuest Thesis and Dissertation Database. (Accession Number 1486378).

Cisco, (2015). *Cisco Global Cloud Index: Forecast and Methodology: 2014-2019.* [White Paper]. Retrieved from Cisco website:

http://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud_Index_White_Paper.pdf

Crawford, B., Soto, R., Cuesta, R. and Paredes, F. (2014). *Application of the Artificial Bee Colony Algorithm for Solving the Set Covering Problem.* The Scientific World Journal, doi:http://dx.doi.org/10.1155/2014/189164

Dorigo, M. and Stutzle, T. (2004). *Ant Colony Optimization.* Cambridge, Massachusetts: The MIT Press.

Dowla, F. (2004). *Handbook of RF and Wireless Technologies.* Mobile Ad Hoc Networks (pp 59-87). Burlington, MA: Newnes.

Frost, J. (2013). *Multiple Regression Analysis: Use Adjusted R-Squared and Predicted R-Squared to include the correct number of variables.* Retrieved from

http://blog.minitab.com/blog/adventures-in-statistics-2/multiple-regession-analysis-use-adjusted-r-squared-and-predicted-r-squared-to-include-the-correct-number-of-variables

Gui, T., Ma, C., Wang, F., and Wilkins, D. (2016). *Survey on Swarm Intelligence based routing protocols for wireless sensor networks: An extensive study.* Paper presented at 2016 IEEE International Conference on Industrial Technology (ICIT). Retrieved from IEEE Xplore Digital Library. (Accession Number 16035649).

Gunes, M. Sorges, U. and Bouazizi, I. (2002). *ARA – The Ant-Colony Based Routing Algorithm for MANETs*. Paper presented at International Conference on Parallel Processing Workshops. Vancouver, Canada. Retrieved from IEEE Xplore Digital Library. (Accession Number 7456242).

Hlaing, Z. and Khine, May. (2011). *An Ant Colony Optimization Algorithm for Solving Traveling Salesman Problem.* Paper presented at 2011 International Conference on Information Communication and Management, Singapore. Retrieved from http://www.ipcsit.com/vol16/11-ICICM2011M029.pdf

Kandari, S. and Pandey, M. (2011). *Simulation based review for analyzing the performance of routing protocols for MANETs.* International Journal of Computer Science and Information Security. 9(8). 194-202.

Karaboga, D. (2005). *An Idea based on Honey Bee Swarm for Numerical Optimization.* Kayseri, Turkey: Erciyes University. Retrieved from http://mf.erciyes.edu.tr/abc/pub/tr06_2005.pdf

Kennedy, J. and Eberhart, R. (1995). *Particle Swarm Optimization.* Paper presented at IEEE International Conference on Neural Networks. Retrieved from IEEE Xplore Digital Library. (Accession Number 5263228).

Kopetz, H. (2011). *Real-Time Systems: Design Principles for Distributed Embedded Applications* (2nd ed.). In J. Stankovic (Ed.), Internet of Things (pp 307-324). New York, NY: Springer.

Liu, L., Dai, Y., and Gao, J. (2014). *Ant Colony Optimization Algorithm for Continous Domains Based on Position Distribution Model of Ant Colony Foraging.* The Scientific Journal. Retrieved from ProQuest Thesis and Dissertation Database. (Accession Number 1547786263).

Madakam, S., Ramaswamy, R., & Tripathi, S. (2015). Internet of Things (IoT): A Literature Review. *Journal of Computer and Communications.* 3, 164-173.

Rijmenam, M. (2016). *Self-driving Cars Will Create 2 Petabytes of Data, What Are The Big Data Opportunities For The Car Industry?.* DATAFLOQ. Retrieved from: https://datafloq.com/read/self-driving-cars-create-2-petabytes-data-annually/172

Rini, D., Shamsuddin, S., and Yuhaniz, S. (2011). *Particle Swarm Optimization: Technique, System, and Challenges.* International Journal of Computer Applications. 14(1). Retrieved from http://www.ijcaonline.org/volume14/number1/pxc3872331.pdf

Roy, B., Banik, S., Dey, P., Sanyal, S. and Chaki, N. (2012). *Ant Colony Based Routing for Mobile Ad-Hoc Networks towards Improved Quality of Services.* Journal of Emerging Trends in Computing and Information Sciences. 3(1), 10-14.

Rutherford, A. (2012). *ANOVA and ANCOVA: A GLM Approach*. Hoboken, New Jersey: John Wiley & Sons, Inc.

Shah, S. and Yaqoob, I. (2016). *A Survey: Internet of Things (IOT) Technologies, Applications, and Challenges.* Paper presented at 2016 IEEE Smart Energy Grid Engineering (SEGE). Retrieved from IEEE Xplore Digital Library. (Accession Number 7589556).

Sharp, R. (2012). *Latency in cloud-based interactive streaming content.* Bell Labs Technical

    Journal. 17(2), 67-80.

Songfan, H., Muqing, W., Wenxing, L., and Dongyang, W. (2015). *Performance Comparison of*

    AODV and DSR in MANET Test-Bed Based on Internet of Things. Retrieved from IEEE

    Xplore Digital Library. (Accession Number 15731633)

Srivastava, S. and Singh, S. (2016). *A Survey on Latency Reduction Approaches for Performance*

    Optimization in Cloud Computing. Paper presented at Second International Conference

    on Computational Intelligence & Communication Technology (CICT). Retrieved from

    IEEE Xplore Digital Library. (Accession Number 16232760).

Strom, D. and Zwet J.F. (2015). *Truth and Lies about Latency in the Cloud.* [White Paper].

    Retrieved from http://www.interxion.com/globalassets/documents/whitepapers-and-

    pdfs/cloud/WP_TRUTHANDLIES_en_0715.pdf

Su, H., Wang, X., and Lin, Z. (2009). *Flocking of Multi-Agents with a Virtual Leader.* Paper

    presented at 46th IEEE Conference on Decision and Control. Retrieved from IEEE Xplore

    Digital Library. (Accession Number 9885165).

Sujithra, S. and Padmavathi, G. (2016). *Internet of Things - An Overview.* Paper presented at

    National Conference on Internet of Things. Retrieved from

    http://www.worldscientificnews.com/wp-content/uploads/2015/10/WSN-41-2016-1-

    3159.pdf

Umamaheswari and Radhamani, G. (2012). *Clustering schemes for Mobile Adhoc Networks: A*

    Review. Paper presented at 2012 International Conference on Computer Communication

    and Informatics (ICCCI). Retrieved from IEEE Xplore Digital Library. (Accession

    Number 12576520).

Varga, A., and OpenSim. (2016). OMNeT++ Installation Guide – Version 5.1.1. [White Paper].

Retrieved from OMNeT++ website: https://omnetpp.org/doc/omnetpp/InstallGuide.pdf

Verma, Y. (2016). *Secure System Simulation – Internet of Things* (Master's Thesis). Retrieved

from ProQuest Thesis and Dissertation Database. (Accession Number 10116148)

Voas, J. (2016). *Networks of Things.* National Institute of Standards and Technology Special

Publication 800-183. Retrieved from

http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-183.pdf

Wahab, M. Nefti-Meziani, S., and Atyabi, A. (2015). *A Comprehensive Review of Swarm*

Optimization Algorithms. PLoS ONE 10(5). Retrieved from

http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0122827

Wang, R. Zhan, Y. and Zhou, H. (2015). Application of Artificial Bee Colony in Model

Parameter Identification of Solar Cells. *Energies,* 8(8), 7563-7581.

doi:http://dx.doi.org/10.3390/en8087563

Ye, Q. and Zhuang, W. (2016). *Distributed and Adaptive Medium Access Control for Internet-of-*

Things-Enabled Mobile Networks. IEEE Internet of Things Journal. PP(99). Retrieved

from IEEE Xplore Digital Library. (Accession Number 2566659).

APPENDIX A: SIMULATION DATA

Validation Data: (Riverbed Modeler – 1, DSO Simulator – 2)

Group (Riverbed Modeler / DSO Simulator)	Bit Rate (Bits/Sec)
1	587.97
1	752.32
1	587.20
1	504.83
1	587.20
1	362.69
1	391.87
1	481.09
1	550.14
1	412.54
1	493.25

1	366.46
1	235.07
1	151.49
1	257.97
1	336.45
1	477.89
1	339.52
1	176.70
1	195.90
2	361.82
2	508.74
2	554.77
2	464.12
2	552.01
2	500.75
2	413.81

2	444.37
2	524.57
2	490.69
2	469.94
2	493.62
2	465.84
2	557.22
2	450.26
2	471.63
2	534.67
2	460.47
2	446.92
2	472.79

Data taken from DSO and Client-Server Models to test and compare DSO performance.

Model (DSO = 1; Client-Server = 2)	Sensors	Data Size	Computing Time
1	12	2500	2.556
1	12	2500	2.845
1	12	2500	2.631
1	12	2500	2.879
1	12	2500	2.623
1	12	2500	2.882
1	12	2500	3.191
1	12	2500	3.117
1	12	2500	3.003
1	12	2500	3.286
1	12	2500	3.174
1	12	2500	2.271
1	12	2500	3.289

1	12	2500	2.596
1	12	2500	2.524
1	12	2500	2.851
1	12	2500	2.616
1	12	2500	2.959
1	12	2500	4.018
1	12	2500	3.315
1	12	2500	3.459
1	12	2500	3.008
1	12	2500	2.751
1	12	2500	3.276
1	12	2500	2.973
1	12	2500	2.918
1	12	2500	2.877
1	12	2500	2.713
1	12	2500	2.827

1	12	2500	3.023
1	12	2500	2.901
1	12	2500	3.164
1	12	2500	1.569
1	12	2500	2.431
1	12	2500	3.022
1	12	2500	3.379
1	12	2500	2.703
1	12	2500	2.651
1	12	2500	2.265
1	12	2500	2.618
2	12	2500	29.341
2	12	2500	28.881
2	12	2500	23.809
2	12	2500	20.912
2	12	2500	28.896

2	12	2500	29.117
2	12	2500	12.901
2	12	2500	21.559
2	12	2500	26.623
2	12	2500	28.931
2	12	2500	15.655
2	12	2500	18.242
2	12	2500	31.613
2	12	2500	29.047
2	12	2500	18.174
2	12	2500	26.311
2	12	2500	29.056
2	12	2500	33.768
2	12	2500	22.771
2	12	2500	18.623
2	12	2500	21.119

2	12	2500	32.352
2	12	2500	31.513
2	12	2500	23.422
2	12	2500	31.499
2	12	2500	18.336
2	12	2500	27.252
2	12	2500	21.073
2	12	2500	29.204
2	12	2500	28.768
2	12	2500	18.361
2	12	2500	34.506
2	12	2500	17.899
2	12	2500	20.935
2	12	2500	34.527
2	12	2500	28.886
2	12	2500	29.171

2	12	2500	31.782
2	12	2500	26.599
2	12	2500	28.898

Data were taken from DSO simulation to test the effect of sensors on DSO mechanism.

Sensors	Data Size	Computing Time
4	2500	2.983
5	2500	2.549
6	2500	2.132
7	2500	3.248
8	2500	2.306
9	2500	1.605
10	2500	1.771
11	2500	2.432
12	2500	2.951
13	2500	2.965
14	2500	2.359

15	2500	1.641
16	2500	1.988
17	2500	2.963
18	2500	3.884
19	2500	3.022
20	2500	2.951
21	2500	2.025
22	2500	2.793
23	2500	2.214
24	2500	2.555
25	2500	2.481
26	2500	3.064
27	2500	2.953
28	2500	2.228
29	2500	1.951
30	2500	2.498

31	2500	1.913
32	2500	3.373
33	2500	3.107
34	2500	3.561
35	2500	3.209

Data were taken from DSO simulation to test the limit of sensors on simulation tool.

Sensors	Data Size (Bits)	Computing Time (Seconds)
5	5000	5.162
10	5000	4.908
15	5000	7.005
20	5000	4.688
25	5000	6.261
30	5000	5.139
35	5000	4.943
40	5000	4.386

45	5000	4.858
50	5000	5.271
55	5000	4.593
60	5000	4.181
65	5000	2.879
70	5000	5.148
75	5000	6.025
80	5000	5.976
85	5000	7.392
90	5000	6.016
95	5000	5.208
100	5000	6.361
105	5000	4.823
110	5000	5.448
115	5000	7.808
120	5000	6.442

125	5000	3.329
130	5000	4.876
135	5000	5.975
140	5000	5.147
145	5000	6.891
150	5000	7.792
155	5000	0.000

Data were taken from DSO simulation to test the effect of data size on simulation tool.

Sensors	Data Size (Bits)	Computing Time (Seconds)
12	600	0.836
12	800	1.165
12	1000	1.067
12	1200	1.268
12	1400	1.483
12	1600	2.015

12	1800	2.476
12	2000	2.419
12	2200	2.668
12	2400	2.796
12	2600	3.015
12	2800	3.058
12	3000	3.401
12	3200	3.604
12	3400	3.581
12	3600	3.829
12	3800	3.566
12	4000	4.182
12	4200	4.603
12	4400	4.961
12	4600	4.429
12	4800	5.292

12	5000	5.232
12	5200	4.707
12	5400	4.977
12	5600	5.814
12	5800	5.236
12	6000	6.067
12	6200	6.682
12	6400	6.427
12	6600	6.089
12	6800	6.243

Data taken from DSO simulation to test effect of both sensors and data size on simulation tool.

Sensors	Data Size (Bits)	Computing Time (Seconds)
4	600	1.968
5	800	0.966
6	1000	1.197

7	1200	1.344
8	1400	1.586
9	1600	1.831
10	1800	2.378
11	2000	3.453
12	2200	2.601
13	2400	3.005
14	2600	3.267
15	2800	3.276
16	3000	3.862
17	3200	4.098
18	3400	4.578
19	3600	5.251
20	3800	4.835
21	4000	5.578
22	4200	6.216

23	4400	3.615
24	4600	4.985
25	4800	5.613
26	5000	5.926
27	5200	6.548
28	5400	6.988
29	5600	6.362
30	5800	7.952
31	6000	6.849
32	6200	7.915
33	6400	7.065
34	6600	6.642
35	6800	7.738

APPENDIX B: SOURCE CODE OF DSO SIMULATOR

File: Simulator.cs

```csharp
using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Threading;

using System.Windows.Forms;

using System.Diagnostics;


namespace DSO_Simulation

{

    public partial class Simulator : Form

    {

        Graphics drawArea;

        Pen blackPen = new Pen(Color.Black);
```

```csharp
List<Node> nodes = new List<Node>();

Dictionary<string, Node> dictNode = new Dictionary<string, Node>();

DataTable dtGlobalTable = new DataTable();

Node aggregator = new Node();

int cloudLatency = 0;

double bitRatePerSecond = 0;

double nodeDistance = 0;

bool aggregatorReached = false;


BackgroundWorker backgroundworker = new BackgroundWorker();

BackgroundWorker backgroundworkerCloud = new BackgroundWorker();

BackgroundWorker backgroundworkerValidation = new BackgroundWorker();

delegate void SetTextCallback(string[] strArray);


static Stopwatch stopwatch;

static Stopwatch overallStopWatch;

static Stopwatch cloudStopWatch;

static Stopwatch validationStopWatch;


static readonly Object lockObj = new Object();


public Simulator()
{
```

```
InitializeComponent();

drawArea = drawingArea.CreateGraphics();

stopwatch = new Stopwatch();

overallStopWatch = new Stopwatch();

cloudStopWatch = new Stopwatch();

validationStopWatch = new Stopwatch();


drawArea.Clear(Color.White);


dtGlobalTable.Columns.Add("NodeName", typeof(String));

dtGlobalTable.Columns.Add("Node", typeof(Node));

dtGlobalTable.Columns.Add("NeighborNodes", typeof(List<Node>));

dtGlobalTable.Columns.Add("ConnectionVisited", typeof(bool));

dtGlobalTable.Columns.Add("ComputingVisited", typeof(bool));

dtGlobalTable.Columns.Add("AggregatorPathVisited", typeof(bool));


backgroundworker.DoWork += new DoWorkEventHandler(DSO_Computing);

backgroundworker.ProgressChanged += new
ProgressChangedEventHandler(Progress_Changed);

backgroundworker.RunWorkerCompleted += new
RunWorkerCompletedEventHandler(background_RunWorkerCompleted);

backgroundworker.WorkerReportsProgress = true;
```

```
        backgroundworkerCloud.DoWork += new
DoWorkEventHandler(MultiHopTransferToAggregator);

        backgroundworkerCloud.ProgressChanged += new
ProgressChangedEventHandler(Cloud_Progress_Changed);

        backgroundworkerCloud.RunWorkerCompleted += new
RunWorkerCompletedEventHandler(Cloud_background_RunWorkerCompleted);

        backgroundworkerCloud.WorkerReportsProgress = true;


        backgroundworkerValidation.DoWork += new
DoWorkEventHandler(MultiHopTransferForValidation);

        backgroundworkerValidation.ProgressChanged += new
ProgressChangedEventHandler(Validation_Progress_Changed);

        backgroundworkerValidation.RunWorkerCompleted += new
RunWorkerCompletedEventHandler(Validation_background_RunWorkerCompleted);

        backgroundworkerValidation.WorkerReportsProgress = true;
    }


    private void btnRun_Click(object sender, EventArgs e)
    {
      int iSensors = 0;
      Int32.TryParse(txtSensors.Text, out iSensors);


      if(iSensors <= 0)
```

```
{

   MessageBox.Show("Please enter a valid Number of Sensors");

   return;

}


SolidBrush blueBrush = new SolidBrush(Color.DarkBlue);

SolidBrush redBrush = new SolidBrush(Color.DarkRed);

Random random = new Random();


//-------------------------------------------------------------------------

//Create Sensor Nodes

//-------------------------------------------------------------------------

int x_max_width = (iSensors * 20) + 100;

if (x_max_width > 920)

{

   x_max_width = 920;

}

int y_max_length = (iSensors * 30);

if(y_max_length > 580)

{

   y_max_length = 580;

}

for (int i = 0; i < iSensors; i++)
```

```
        {

            Node node = new Node();

            node.Name = i.ToString();

            node.X_axis = random.Next(1, x_max_width);

            node.Y_axis = random.Next(1, y_max_length);


            drawArea.FillPolygon(blueBrush, DrawNode(node.X_axis, node.Y_axis));


            nodes.Add(node);

        }


        //------------------------------------------------------------------------------

        //Add Aggregator to the environment

        //------------------------------------------------------------------------------

        aggregator.Name = "A";

        aggregator.X_axis = random.Next(1, x_max_width);

        aggregator.Y_axis = random.Next(1, y_max_length);

        drawArea.FillPolygon(redBrush, DrawNode(aggregator.X_axis,

aggregator.Y_axis));

        nodes.Add(aggregator);


        //------------------------------------------------------------------------------
```

```
        //Finding neighboring nodes that are within 150 points range and fill the global
table

        //----------------------------------------------------------------------------------

        FillGlobalTable(150);


        DrawConnections("0", "");

      }


      private void FillGlobalTable(int range)

      {

        for (int i = 0; i < nodes.Count; i++)

        {

          List<Node> tempList = new List<Node>();

          IList<string> neighborNodeList = new List<string>();

          for (int j = 0; j < nodes.Count; j++)

          {

            if (i != j)

            {

              if (Math.Abs(nodes[i].X_axis - nodes[j].X_axis) <= range &&
Math.Abs(nodes[i].Y_axis - nodes[j].Y_axis) <= range)

              {

                tempList.Add(nodes[j]);

                neighborNodeList.Add(nodes[j].Name);
```

```
            }

         }

   }

   DataRow drNew = dtGlobalTable.NewRow();

   drNew["NodeName"] = nodes[i].Name;

   drNew["Node"] = nodes[i];

   drNew["NeighborNodes"] = tempList;

   drNew["ConnectionVisited"] = false;

   drNew["ComputingVisited"] = false;

   drNew["AggregatorPathVisited"] = false;


   dtGlobalTable.Rows.Add(drNew);


   string strNeighborNodeList = "";

   foreach (Node tnode in tempList)

   {

      strNeighborNodeList += tnode.Name + ",";

   }

   nodes[i].NeighborList = string.Join(",", neighborNodeList);

   dictNode.Add(nodes[i].Name, nodes[i]);


   dgvNeighbors.Rows.Add(new string[] { nodes[i].Name, strNeighborNodeList

});
```

```csharp
            }
        }


        private Point[] DrawNode(int middleX, int middleY)
        {
            //Size of Node is 20 points in X axis, 30 points in Y axis
            return new Point[] { new Point(middleX-10, middleY+15), new Point(middleX,
middleY-15), new Point(middleX+10, middleY+15) };
        }


        private void btnReset_Click(object sender, EventArgs e)
        {
            drawArea.Clear(Color.White);
            dtGlobalTable.Clear();
            nodes.Clear();
            dgvResults.Rows.Clear();
            dgvNeighbors.Rows.Clear();
            dictNode.Clear();

            txtDSOResult.Text = "";
            txtClientServerResult.Text = "";
            txtBitPerSec.Text = "";
            txtNodeDistance.Text = "";
```

```
        aggregatorReached = false;

}


private void DrawConnections(string currentNodeName, string previousNodeName)

{

    //----------------------------------------------------------

    //Check the global table for neighboring nodes

    //----------------------------------------------------------

    DataRow drRow = (from Row in dtGlobalTable.AsEnumerable()

            where Row.Field<string>("NodeName") == currentNodeName

            select Row).FirstOrDefault<DataRow>();


    List<Node> lstNeighborNodes = drRow.Field<List<Node>>("NeighborNodes");

    Node currentNode = drRow.Field<Node>("Node");


    //----------------------------------------------------------------------

    //Setting the flag to reflect the current node is visited for connection

    //----------------------------------------------------------------------

    drRow["ConnectionVisited"] = true;


    List<Node> neighborNodesToBeVisited = new List<Node>();

    //-------------------------------------------------
```

```csharp
//Find each neighboring node to draw a connection

//-----------------------------------------------

foreach (Node neighborNode in lstNeighborNodes)

{

    //------------------------------------------------------------------------

    //First find if the neighboring node already got connection established.

    //------------------------------------------------------------------------

    DataRow drNeighborRow = (from dRow in dtGlobalTable.AsEnumerable()

                where dRow.Field<string>("NodeName") ==

neighborNode.Name

                select dRow).FirstOrDefault<DataRow>();


    if(drNeighborRow.Field<bool>("ConnectionVisited") == false)

    {

        //----------------------------------------------------------------------

        //This node hasn't established connection yet, so new connection will be

        //created and ConnectionVisited property will be set to true.

        //----------------------------------------------------------------------

        drawArea.DrawLine(blackPen, currentNode.X_axis, currentNode.Y_axis,

neighborNode.X_axis, neighborNode.Y_axis);


        //-------------------------------------------------------------------------------
```

```
        //Qualified Neighbor nodes are added to the list so they can be recursively

visited

            //-------------------------------------------------------------------------------

            neighborNodesToBeVisited.Add(neighborNode);

        }

    }



        //--------------------------------------------------------------------

        //Now recursively visit available neighboring nodes to establish connection

        //--------------------------------------------------------------------

        foreach (Node neighborToBeVisited in neighborNodesToBeVisited)

        {

            DrawConnections(neighborToBeVisited.Name, currentNodeName);

        }

    }



    private void MultiHopTransferToAggregator(object sender, DoWorkEventArgs e)

    {

        string data = BuildData();

        if(data == "")

        {

            return;

        }
```

```
//-------------------------------------------

//Find the 0 node from global table

//-------------------------------------------

DataRow[] drRows = dtGlobalTable.Select("NodeName='0'");

if(drRows.Length >0)

{

   Node currNode = drRows[0].Field<Node>("Node");

   MultiHoptoAggregator("0", data, currNode, 0);

}




}


    private void MultiHoptoAggregator(string nodeName, string data, Node prevNode,

double sumTransferTime)

    {

      if (!aggregatorReached)

      {

        //-------------------------------------------

        //Find the node from global table

        //-------------------------------------------

        DataRow[] drRows = dtGlobalTable.Select("NodeName='" + nodeName + "'");
```

```
if (drRows.Length > 0)

{

    drRows[0]["AggregatorPathVisited"] = true;

    Node currNode = drRows[0].Field<Node>("Node");


    double bit_RatePerSecond = 0;

    double distance = GetDistance(currNode, prevNode);

    double transferTime = GetTransferTime(distance, data, out
bit_RatePerSecond);

    sumTransferTime += transferTime;


    //-----------------------------------------------------------

    //First check if the node is Aggregator. If so, begin sending

    //to cloud.

    //-----------------------------------------------------------

    if (nodeName == "A")

    {

        //-------------------------------------------------------------

        //Sleep for total transfer time to simulate the bit transfer time

        //-------------------------------------------------------------

        Thread.Sleep(Convert.ToInt32(Math.Round(sumTransferTime)));
```

```
//-------------------------------------------------------------
//Send to cloud for processing. Thread sleeps for latency time.
//-------------------------------------------------------------
Thread.Sleep(cloudLatency);


ProcessData(data);

aggregatorReached = true;

cloudStopWatch.Stop();


}
else
{
    //---------------------------------------------------------
    //Select all the neighbor nodes and recursively move if any
    //one of them not visited yet.
    //---------------------------------------------------------
    List<Node> neighborNodes =
drRows[0].Field<List<Node>>("NeighborNodes");


    foreach (Node n in neighborNodes)
    {
        //-------------------------------------------------------
        //Check if this node was already visited; if not, it will be
```

```csharp
                    //recursively sent.

                    //---------------------------------------------------------

                    bool visited = (from drQRow in dtGlobalTable.AsEnumerable()

                            where drQRow["NodeName"].ToString() == n.Name

                            select

drQRow.Field<bool>("AggregatorPathVisited")).FirstOrDefault<bool>();



                    if (visited == false)

                    {

                        MultiHoptoAggregator(n.Name, data, currNode, sumTransferTime);

                    }

                }

            }

        }

    }



        //---------------------------------------------------------

        //Finds the direct distance between nodes based on both of

        //their x and y axis.

        //---------------------------------------------------------

        private double GetDistance(Node currNode, Node prevNode)

        {
```

```
        double distance = 0;


    //----------------------------------------------------

    //Find the absolute x and y distance between the nodes

    //----------------------------------------------------

    int x_distance = Math.Abs(currNode.X_axis - prevNode.X_axis);

    int y_distance = Math.Abs(currNode.Y_axis - prevNode.Y_axis);


    distance = Math.Sqrt(Math.Pow(x_distance, 2) + Math.Pow(y_distance, 2));

    return distance;

}



    //---------------------------------------------------------

    //Finds the bit transfer time from one node to another.

    //---------------------------------------------------------

    private double GetTransferTime(double distance, string data, out double

bitRate_perSecond)

    {

        bitRate_perSecond = Find_BitRate(distance);

        double transferTime_millisecond = ((1000 / bitRate_perSecond) * data.Length);

        return transferTime_millisecond;

    }
```

```
//-------------------------------------------------------

//Finds the bit rate of the node.

//-------------------------------------------------------

private double Find_BitRate(double distance)

{

    //-----------------------------------------------------

    //Based on Riverbed Modeler's MANET simulation, following

    //equation was derived to calculate the transfer time.

    //-----------------------------------------------------

    double bitRate_perSecond = 1030.095 - (0.823 * distance);

    return bitRate_perSecond;

}
/// <summary>

/// Builds the data bases on the size given. Size is in bytes

/// and it is considered one byte is equal to one character in

/// the string.

/// </summary>

/// <param name="dataSize"></param>

/// <returns></returns>

private string BuildData()

{

    int dataSize = 0;

    int.TryParse(txtSizeofData.Text, out dataSize);
```

```csharp
if(dataSize == 0)

{

   MessageBox.Show("Enter a valid data size");

   return "";

}

string data = "";

for(int i = 0; i<dataSize; i++)

{

   data += "0";

}


return data;

}

private void btnDSO_Click(object sender, EventArgs e)

{

   stopwatch.Reset();

   stopwatch.Start();


   object[] parameters = new object[] { "0", ""};

   backgroundworker.RunWorkerAsync(parameters);


}
```

```csharp
private void DSO_Computing(object sender, DoWorkEventArgs e)
{
    string data = BuildData();


    stopwatch.Reset();

    stopwatch.Start();


    overallStopWatch.Reset();

    overallStopWatch.Start();

    //----------------------------------------------------------

    //Get the '0' node. Process starts with '0' node.

    //----------------------------------------------------------

    DataRow[] drNeighborRows = dtGlobalTable.Select("NodeName='0'");


    if(drNeighborRows.Length > 0)
    {

        drNeighborRows[0]["ComputingVisited"] = true;

        List<Node> lstNeighborNodes =
drNeighborRows[0].Field<List<Node>>("NeighborNodes");

        Node currNode = drNeighborRows[0].Field<Node>("Node");
```

```
//-------------------------------------------------------------
//Build distributed data with neighboring nodes
//-------------------------------------------------------------
string currentNodeData = "";
if (data.Length >= 50)
{
    Dictionary<Node, string> dso_information =
DistributeData(lstNeighborNodes, data, out currentNodeData);


    Parallel.ForEach(dso_information, (nodeDataPair) =>
Computing(nodeDataPair,currNode));
}
else
{
    currentNodeData = data;
}


//-------------------------------------------------------------
//Now compute the current node
//-------------------------------------------------------------
ProcessData(currentNodeData);
stopwatch.Stop();
```

```
            //---------------------------------------------

            //Get Neighboring Node list for debug information

            //---------------------------------------------

            Node currentNode;

            dictNode.TryGetValue("0", out currentNode);

            string[] strArray = { drNeighborRows[0].Field<Node>("Node").Name,

currentNodeData.Length.ToString(), stopwatch.Elapsed.ToString(), currentNode.NeighborList };

            this.SetText(strArray);

        }


    }


        public void Computing(KeyValuePair<Node,string> nodeDataPair, Node

previousNode)

        {

            stopwatch.Reset();

            stopwatch.Start();

            Node n = nodeDataPair.Key;

            string data = nodeDataPair.Value;

            //-------------------------------------------------------------------------------------

            //Find all the neighbors who are available for computing (based on

computingvisited flag

            //-------------------------------------------------------------------------------------
```

```
DataRow[] drRows = dtGlobalTable.Select("NodeName='" + n.Name + "'");

if(drRows.Length > 0)
{
    List<Node> lstNeighborNodes = new List<Node>();

    string currentNodeData = "";

    Dictionary<Node, string> dso_neighborInformation = new Dictionary<Node,
string>();

    lock (lockObj)
    {
        drRows[0]["ComputingVisited"] = true;
    }


    Node currNode = drRows[0].Field<Node>("Node");

    double bit_RatePerSecond = 0;

    double distance = GetDistance(currNode, previousNode);

    double transferTime = GetTransferTime(distance, data, out
bit_RatePerSecond);


        //-----------------------------------------------------------
        //Let it sleep the transfer time to simulate the actual transfer
        //time.
        //-----------------------------------------------------------
```

```
Thread.Sleep(Convert.ToInt16(Math.Round(transferTime)));


//---------------------------------------------------------------

//If the data to processed is less than 50 bytes, we don't have to

//distribute because it is more efficient to process in a single node

//than distributing to tiny pieces.

//---------------------------------------------------------------

if (data.Length >= 50)

{

   lock (lockObj)

   {

      lstNeighborNodes = drRows[0].Field<List<Node>>("NeighborNodes");


      dso_neighborInformation = DistributeData(lstNeighborNodes,
data.ToString(), out currentNodeData);

   }

      Parallel.ForEach(dso_neighborInformation, (neighborNodeDataPair) =>
Computing(neighborNodeDataPair, currNode));

}

else

{

   currentNodeData = data;

}
```

```csharp
                    //-------------------------------------------------

                    //No neighbor nodes are found. So compute the data

                    //-------------------------------------------------

                    ProcessData(currentNodeData);

                    stopwatch.Stop();


                    Node currentNode;

                    dictNode.TryGetValue(n.Name, out currentNode);


                    string[] strArray = { n.Name, currentNodeData.Length.ToString(),
stopwatch.Elapsed.ToString(), currentNode.NeighborList };

                    this.SetText(strArray);

                }

            }


        private Dictionary<Node, string> DistributeData(List<Node> lstNeighborNodes,
string data, out string currentNodeData)

            {

                //---------------------------------------------------------

                //Find which neighboring nodes are available for computing

                //and assign them for computing

                //---------------------------------------------------------

                List<Node> availableNodesForComputing = new List<Node>();
```

```
        foreach (Node neighborNode in lstNeighborNodes)

        {

            //--------------------------------------------------------

            //If the neighbor node is Aggregator, it will skip because

            //aggregators are not participating in computing.

            //--------------------------------------------------------

            if (neighborNode.Name != "A")

            {

                DataRow[] drAvailableNode = dtGlobalTable.Select("NodeName='" +
neighborNode.Name + "' and ComputingVisited=false");

                if (drAvailableNode.Length > 0 && drAvailableNode[0] != null)

                {

                    Node nd = drAvailableNode[0].Field<Node>("Node");

                    availableNodesForComputing.Add(nd);


                    //------------------------------------------------------------------------------------

                    //Flag will be set to visited so distribution will happen accordingly. If flag
is not set

                    //here other threads would be using these neighboring nodes for
distribution.

                    //------------------------------------------------------------------------------------
-
```

```
        drAvailableNode[0]["ComputingVisited"] = true;

    // }

  }

}

}


List<string> dataSegments = new List<string>();


int totalNodes = availableNodesForComputing.Count() + 1;

int DataSegmentLength = data.Length / totalNodes;


if ((data.Length % totalNodes) == 0)

{

  //-------------------------------------------------------

  //Data can be equally distributed. First segment is for

  //the current node to process and the remaining segments are

  //for neighbor nodes to process.

  //-------------------------------------------------------

  int startIndex = 0;

  currentNodeData = data.Substring(startIndex, DataSegmentLength);

  startIndex += DataSegmentLength;


  while((startIndex+DataSegmentLength) <= data.Length)
```

```
    {

        dataSegments.Add(data.Substring(startIndex, DataSegmentLength));

        startIndex += DataSegmentLength;

    }

}

else

{

    //-----------------------------------------------

    //Data segments can not be equally distributed

    //so we will put the first segment to have +1 and

    //rest will be equal segments

    //-----------------------------------------------

    int firstDataSegmentLength = DataSegmentLength + 1;

    int startIndex = firstDataSegmentLength;


    currentNodeData = data.Substring(0, firstDataSegmentLength);


    while ((startIndex + DataSegmentLength) <= data.Length)

    {

        dataSegments.Add(data.Substring(startIndex, DataSegmentLength));

        startIndex += DataSegmentLength;

    }

}
```

```csharp
        Dictionary<Node, string> neighborNodesData = new Dictionary<Node, string>();

        int counter = 0;

        foreach(Node node in availableNodesForComputing)

        {

            neighborNodesData.Add(node, dataSegments[counter++]);

        }


        return neighborNodesData;

    }


    /// <summary>

    /// Process the data incoming by changing the characters to 'x' and

    /// pauses for the size of the incoming data.

    /// </summary>

    /// <param name="data"></param>

    private void ProcessData(string data)

    {

        char[] dataArray = data.ToCharArray();

        for (int i = 0; i < dataArray.Length; i++)

        {

            dataArray[i] = '1';

        }
```

```csharp
    //---------------------------------------------------------

    //This simulation considers one millisecond to process a byte

    //and takes the data size of the time to pause to simulate

    //the processing.

    //---------------------------------------------------------

    Thread.Sleep(data.Length);

}


private void SetText(string[] textArray)

{

    if(this.btnReset.InvokeRequired)

    {

        SetTextCallback d = new SetTextCallback(SetText);

        this.Invoke(d, new object[] { textArray });

    }

    else


    {

        dgvResults.Rows.Add(textArray);

    }

}

private void Progress_Changed(object sender, ProgressChangedEventArgs e)
```

```
        {
            btnReset.Text = e.ProgressPercentage.ToString();


        }


    private void background_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)
        {
            overallStopWatch.Stop();


            Node aggNode = new Node();

            dictNode.TryGetValue("A", out aggNode);


            if(aggNode == null)
            {
                MessageBox.Show("No network found. Please build MANET.");

                return;
            }
            string[] strArray = { "A", "-", "-", aggNode.NeighborList };

            this.SetText(strArray);


            string[] strTotArray = { "Total", "-", overallStopWatch.Elapsed.ToString(), "-" };

            this.SetText(strTotArray);
```

```
            txtDSOResult.Text = overallStopWatch.Elapsed.ToString();

        }


        private void Cloud_Progress_Changed(object sender, ProgressChangedEventArgs e)

        {

            //btnReset.Text = e.ProgressPercentage.ToString();


        }


        private void Validation_Progress_Changed(object sender,

ProgressChangedEventArgs e)

        {

            //btnReset.Text = e.ProgressPercentage.ToString();


        }


        private void Cloud_background_RunWorkerCompleted(object sender,

RunWorkerCompletedEventArgs e)

        {

            cloudStopWatch.Stop();

            txtClientServerResult.Text = cloudStopWatch.Elapsed.ToString();

        }
```

```
private void Validation_background_RunWorkerCompleted(object sender,
RunWorkerCompletedEventArgs e)

    {

        validationStopWatch.Stop();

        txtClientServerResult.Text = validationStopWatch.Elapsed.ToString();

        txtBitPerSec.Text = bitRatePerSecond.ToString();

        txtNodeDistance.Text = nodeDistance.ToString();

    }


    private void btnClientServer_Click(object sender, EventArgs e)

    {

        int.TryParse(txtLatency.Text, out cloudLatency);

        if(cloudLatency == 0)

        {

            MessageBox.Show("Cloud Latency should be set to non zero");

        }

        cloudStopWatch.Reset();

        cloudStopWatch.Start();


        backgroundworkerCloud.RunWorkerAsync();

    }


    private void MultihopToAggregator(object sender, DoWorkEventArgs e)
```

```csharp
{
    string data = BuildData();


    cloudStopWatch.Reset();

    cloudStopWatch.Start();


    //----------------------------------------------------------

    //Get the '0' node. Process starts with '0' node.

    //----------------------------------------------------------

    DataRow[] drNeighborRows = dtGlobalTable.Select("NodeName='0'");


    if (drNeighborRows.Length > 0)
    {


        drNeighborRows[0]["ComputingVisited"] = true;

        List<Node> lstNeighborNodes =
drNeighborRows[0].Field<List<Node>>("NeighborNodes");

        Node currNode = drNeighborRows[0].Field<Node>("Node");


        //Dictionary<Node, string> dso_information =
DistributeData(lstNeighborNodes, data, out currentNodeData);
```

```
            Parallel.ForEach(lstNeighborNodes, (neighborNode) =>
Multihop(neighborNode, currNode, data));

        }


        }


        public void Multihop(Node currNode, Node previousNode, string data)
        {
            //-----------------------------------------------------------------------------------
            //Find all the neighbors who are available for computing (based on
computingvisited flag
            //-----------------------------------------------------------------------------------
            DataRow[] drRows = dtGlobalTable.Select("NodeName='" + currNode.Name + "'
and ComputingVisited=false");

            if (drRows.Length > 0)
            {
                List<Node> lstNeighborNodes = new List<Node>();


                lock (lockObj)
                {
                    drRows[0]["ComputingVisited"] = true;
                }
```

```
lstNeighborNodes = drRows[0].Field<List<Node>>("NeighborNodes");


double bit_RatePerSecond = 0;

double distance = GetDistance(currNode, previousNode);

double transferTime = GetTransferTime(distance, data, out

bit_RatePerSecond);


//-----------------------------------------------------------

//Let it sleep the transfer time to simulate the actual transfer

//time.

//-----------------------------------------------------------

Thread.Sleep(Convert.ToInt16(Math.Round(transferTime)));


if (currNode.Name != "A")

{

    Parallel.ForEach(lstNeighborNodes, (neighborNode) =>

Multihop(neighborNode, currNode, data));

}

else

{

    //-----------------------------------------

    //Wait for simulating cloud latency
```

```
        //-----------------------------------------

        Thread.Sleep(cloudLatency);


        //-----------------------------------------

        //Wait for processing data

        //-----------------------------------------

        Thread.Sleep(data.Length);


        cloudStopWatch.Stop();

    }


  }

}

private void btnValidation_Click(object sender, EventArgs e)

{

    SolidBrush blueBrush = new SolidBrush(Color.DarkBlue);

    Random random = new Random();

    for (int i = 0; i < 4; i++)

    {

        Node node = new Node();

        node.Name = i.ToString();

        node.X_axis = random.Next(1, 920);

        node.Y_axis = random.Next(1, 580);
```

```
        drawArea.FillPolygon(blueBrush, DrawNode(node.X_axis, node.Y_axis));


        nodes.Add(node);

    }


    FillGlobalTable(700);

    DrawConnections("0", "");

}

private void btnRunValidation_Click(object sender, EventArgs e)

{

    validationStopWatch.Reset();

    validationStopWatch.Start();


    backgroundworkerValidation.RunWorkerAsync();

}

private void MultiHopTransferForValidation(object sender, DoWorkEventArgs e)

{

    string data = BuildData();

    if (data == "")

    {

        return;

    }
```

```
    //-------------------------------------------

    //Find the 0 node from global table

    //-------------------------------------------

    DataRow[] drRows = dtGlobalTable.Select("NodeName='0'");

    if (drRows.Length > 0)

    {

        Node currNode = drRows[0].Field<Node>("Node");

        Validation_MultiHop("0", data, currNode);

    }



}


/// <summary>

/// This method is called when validation process is running.

/// </summary>

/// <param name="nodeName"></param>

/// <param name="data"></param>

/// <param name="prevNode"></param>

private void Validation_MultiHop(string nodeName, string data, Node prevNode)

{

    //-------------------------------------------

    //Find the node from global table
```

```csharp
//------------------------------------------
DataRow[] drRows = dtGlobalTable.Select("NodeName='" + nodeName + "'");


if (drRows.Length > 0)

{

    drRows[0]["AggregatorPathVisited"] = true;

    Node currNode = drRows[0].Field<Node>("Node");


    double bit_RatePerSecond = 0;

    double distance = GetDistance(currNode, prevNode);

    double transferTime = GetTransferTime(distance, data, out
bit_RatePerSecond);


        //-----------------------------------------------------------
        //Let it sleep the transfer time to simulate the actual transfer
        //time.
        //-----------------------------------------------------------
        Thread.Sleep(Convert.ToInt16(Math.Round(transferTime)));


        //-----------------------------------------------------------
        //First check if the node is Aggregator. If so, begin sending
        //to cloud.
        //-----------------------------------------------------------
```

```csharp
if (nodeName == "3")

{

    bitRatePerSecond = bit_RatePerSecond;

    nodeDistance = distance;

    ProcessData(data);

}

else

{

    //----------------------------------------------------------

    //Select all the neighbor nodes and recursively move if any

    //one of them not visited yet.

    //----------------------------------------------------------

    List<Node> neighborNodes =
drRows[0].Field<List<Node>>("NeighborNodes");


    foreach (Node n in neighborNodes)

    {

        //----------------------------------------------------------

        //Check if this node was already visited; if not, it will be

        //recursively sent.

        //----------------------------------------------------------

        bool visited = (from drQRow in dtGlobalTable.AsEnumerable()

                    where drQRow["NodeName"].ToString() == n.Name
```

```
                               select
drQRow.Field<bool>("AggregatorPathVisited")).FirstOrDefault<bool>();


                     if (visited == false)

                     {

                         Validation_MultiHop(n.Name, data, currNode);

                     }

                  }

               }

            }



         }

      }


      public class Node

      {

         public string Name = "";

         public int X_axis = 0;

         public int Y_axis = 0;

         public string NeighborList = "";

      }

   }
   //---------------------------------------------------------------------------------------
```

File: Simulator.Designer.cs

```csharp
namespace DSO_Simulation
{
    partial class Simulator
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed; otherwise,
false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
```

```
}
```

#region Windows Form Designer generated code

```
/// <summary>

/// Required method for Designer support - do not modify

/// the contents of this method with the code editor.

/// </summary>

private void InitializeComponent()

{

    this.lblSensors = new System.Windows.Forms.Label();

    this.txtSensors = new System.Windows.Forms.TextBox();

    this.txtSizeofData = new System.Windows.Forms.TextBox();

    this.lblSizeofData = new System.Windows.Forms.Label();

    this.btnRun = new System.Windows.Forms.Button();

    this.drawingArea = new System.Windows.Forms.PictureBox();

    this.btnReset = new System.Windows.Forms.Button();

    this.btnDSO = new System.Windows.Forms.Button();

    this.dgvResults = new System.Windows.Forms.DataGridView();

    this.Node = new System.Windows.Forms.DataGridViewTextBoxColumn();

    this.DataSize = new System.Windows.Forms.DataGridViewTextBoxColumn();

    this.Time = new System.Windows.Forms.DataGridViewTextBoxColumn();

    this.NeighborsList = new System.Windows.Forms.DataGridViewTextBoxColumn();
```

```
this.btnClientServer = new System.Windows.Forms.Button();

this.txtLatency = new System.Windows.Forms.TextBox();

this.lblLatency = new System.Windows.Forms.Label();

this.groupBox1 = new System.Windows.Forms.GroupBox();

this.lblNodeDistance = new System.Windows.Forms.Label();

this.txtNodeDistance = new System.Windows.Forms.TextBox();

this.lblBitPerSec = new System.Windows.Forms.Label();

this.txtBitPerSec = new System.Windows.Forms.TextBox();

this.lblClientServerResult = new System.Windows.Forms.Label();

this.lblDSOResult = new System.Windows.Forms.Label();

this.txtClientServerResult = new System.Windows.Forms.TextBox();

this.txtDSOResult = new System.Windows.Forms.TextBox();

this.dgvNeighbors = new System.Windows.Forms.DataGridView();

this.NodeName = new System.Windows.Forms.DataGridViewTextBoxColumn();

this.NeighborList = new System.Windows.Forms.DataGridViewTextBoxColumn();

this.btnValidation = new System.Windows.Forms.Button();

this.btnRunValidation = new System.Windows.Forms.Button();

((System.ComponentModel.ISupportInitialize)(this.drawingArea)).BeginInit();

((System.ComponentModel.ISupportInitialize)(this.dgvResults)).BeginInit();

this.groupBox1.SuspendLayout();

((System.ComponentModel.ISupportInitialize)(this.dgvNeighbors)).BeginInit();

this.SuspendLayout();

//
```

```
// lblSensors

//

this.lblSensors.AutoSize = true;

this.lblSensors.Location = new System.Drawing.Point(21, 21);

this.lblSensors.Name = "lblSensors";

this.lblSensors.Size = new System.Drawing.Size(80, 13);

this.lblSensors.TabIndex = 0;

this.lblSensors.Text = "No. of Sensors:";

//

// txtSensors

//

this.txtSensors.Location = new System.Drawing.Point(101, 19);

this.txtSensors.Name = "txtSensors";

this.txtSensors.Size = new System.Drawing.Size(32, 20);

this.txtSensors.TabIndex = 1;

//

// txtSizeofData

//

this.txtSizeofData.Location = new System.Drawing.Point(101, 46);

this.txtSizeofData.Name = "txtSizeofData";

this.txtSizeofData.Size = new System.Drawing.Size(32, 20);

this.txtSizeofData.TabIndex = 3;

//
```

```
// lblSizeofData

//

this.lblSizeofData.AutoSize = true;

this.lblSizeofData.Location = new System.Drawing.Point(1, 48);

this.lblSizeofData.Name = "lblSizeofData";

this.lblSizeofData.Size = new System.Drawing.Size(93, 13);

this.lblSizeofData.TabIndex = 2;

this.lblSizeofData.Text = "Size of Data (bits):";

//

// btnRun

//

this.btnRun.Location = new System.Drawing.Point(8, 158);

this.btnRun.Name = "btnRun";

this.btnRun.Size = new System.Drawing.Size(132, 30);

this.btnRun.TabIndex = 7;

this.btnRun.Text = "Build MANET";

this.btnRun.UseVisualStyleBackColor = true;

this.btnRun.Click += new System.EventHandler(this.btnRun_Click);

//

// drawingArea

//

this.drawingArea.Location = new System.Drawing.Point(163, 11);

this.drawingArea.Name = "drawingArea";
```

```
this.drawingArea.Size = new System.Drawing.Size(920, 580);

this.drawingArea.TabIndex = 8;

this.drawingArea.TabStop = false;

//

// btnReset

//

this.btnReset.Location = new System.Drawing.Point(8, 256);

this.btnReset.Name = "btnReset";

this.btnReset.Size = new System.Drawing.Size(132, 30);

this.btnReset.TabIndex = 9;

this.btnReset.Text = "Reset";

this.btnReset.UseVisualStyleBackColor = true;

this.btnReset.Click += new System.EventHandler(this.btnReset_Click);

//

// btnDSO

//

this.btnDSO.Location = new System.Drawing.Point(8, 190);

this.btnDSO.Name = "btnDSO";

this.btnDSO.Size = new System.Drawing.Size(132, 30);

this.btnDSO.TabIndex = 10;

this.btnDSO.Text = "DSO";

this.btnDSO.UseVisualStyleBackColor = true;

this.btnDSO.Click += new System.EventHandler(this.btnDSO_Click);
```

```
//

// dgvResults

//

this.dgvResults.ColumnHeadersHeightSizeMode =

System.Windows.Forms.DataGridViewColumnHeadersHeightSizeMode.AutoSize;

this.dgvResults.Columns.AddRange(new

System.Windows.Forms.DataGridViewColumn[] {

this.Node,

this.DataSize,

this.Time,

this.NeighborsList});

this.dgvResults.Location = new System.Drawing.Point(295, 596);

this.dgvResults.Name = "dgvResults";

this.dgvResults.Size = new System.Drawing.Size(448, 155);

this.dgvResults.TabIndex = 13;

//

// Node

//

this.Node.HeaderText = "Node";

this.Node.Name = "Node";

this.Node.ReadOnly = true;

//

// DataSize
```

```
//

this.DataSize.HeaderText = "Data Size (Bytes)";

this.DataSize.Name = "DataSize";

this.DataSize.ReadOnly = true;

//

// Time

//

this.Time.HeaderText = "Time";

this.Time.Name = "Time";

this.Time.ReadOnly = true;

//

// NeighborsList

//

this.NeighborsList.HeaderText = "Neighbors List";

this.NeighborsList.Name = "NeighborsList";

this.NeighborsList.ReadOnly = true;

//

// btnClientServer

//

this.btnClientServer.Location = new System.Drawing.Point(8, 223);

this.btnClientServer.Name = "btnClientServer";

this.btnClientServer.Size = new System.Drawing.Size(132, 30);

this.btnClientServer.TabIndex = 14;
```

```
this.btnClientServer.Text = "Client-Server";

this.btnClientServer.UseVisualStyleBackColor = true;

this.btnClientServer.Click += new System.EventHandler(this.btnClientServer_Click);

//

// txtLatency

//

this.txtLatency.Location = new System.Drawing.Point(101, 72);

this.txtLatency.Name = "txtLatency";

this.txtLatency.Size = new System.Drawing.Size(32, 20);

this.txtLatency.TabIndex = 15;

//

// lblLatency

//

this.lblLatency.AutoSize = true;

this.lblLatency.Location = new System.Drawing.Point(1, 73);

this.lblLatency.Name = "lblLatency";

this.lblLatency.Size = new System.Drawing.Size(100, 13);

this.lblLatency.TabIndex = 16;

this.lblLatency.Text = "Cloud Latency (ms):";

//

// groupBox1

//

this.groupBox1.Controls.Add(this.lblNodeDistance);
```

```
this.groupBox1.Controls.Add(this.txtNodeDistance);

this.groupBox1.Controls.Add(this.lblBitPerSec);

this.groupBox1.Controls.Add(this.txtBitPerSec);

this.groupBox1.Controls.Add(this.lblClientServerResult);

this.groupBox1.Controls.Add(this.lblDSOResult);

this.groupBox1.Controls.Add(this.txtClientServerResult);

this.groupBox1.Controls.Add(this.txtDSOResult);

this.groupBox1.Location = new System.Drawing.Point(749, 596);

this.groupBox1.Name = "groupBox1";

this.groupBox1.Size = new System.Drawing.Size(332, 133);

this.groupBox1.TabIndex = 17;

this.groupBox1.TabStop = false;

this.groupBox1.Text = "Results";

//

// lblNodeDistance

//

this.lblNodeDistance.AutoSize = true;

this.lblNodeDistance.Location = new System.Drawing.Point(32, 106);

this.lblNodeDistance.Name = "lblNodeDistance";

this.lblNodeDistance.Size = new System.Drawing.Size(130, 13);

this.lblNodeDistance.TabIndex = 7;

this.lblNodeDistance.Text = "Validation Node Distance:";

//
```

```
// txtNodeDistance
//
this.txtNodeDistance.Location = new System.Drawing.Point(168, 103);

this.txtNodeDistance.Name = "txtNodeDistance";

this.txtNodeDistance.Size = new System.Drawing.Size(100, 20);

this.txtNodeDistance.TabIndex = 6;
//
// lblBitPerSec
//
this.lblBitPerSec.AutoSize = true;

this.lblBitPerSec.Location = new System.Drawing.Point(22, 77);

this.lblBitPerSec.Name = "lblBitPerSec";

this.lblBitPerSec.Size = new System.Drawing.Size(141, 13);

this.lblBitPerSec.TabIndex = 5;

this.lblBitPerSec.Text = "Validation Bit Rate (bit/Sec):";
//
// txtBitPerSec
//
this.txtBitPerSec.Location = new System.Drawing.Point(168, 74);

this.txtBitPerSec.Name = "txtBitPerSec";

this.txtBitPerSec.Size = new System.Drawing.Size(100, 20);

this.txtBitPerSec.TabIndex = 4;
//
```

```
// lblClientServerResult

//

this.lblClientServerResult.AutoSize = true;

this.lblClientServerResult.Location = new System.Drawing.Point(13, 48);

this.lblClientServerResult.Name = "lblClientServerResult";

this.lblClientServerResult.Size = new System.Drawing.Size(149, 13);

this.lblClientServerResult.TabIndex = 3;

this.lblClientServerResult.Text = "Client-Server Computing Time:";

//

// lblDSOResult

//

this.lblDSOResult.AutoSize = true;

this.lblDSOResult.Location = new System.Drawing.Point(50, 18);

this.lblDSOResult.Name = "lblDSOResult";

this.lblDSOResult.Size = new System.Drawing.Size(112, 13);

this.lblDSOResult.TabIndex = 2;

this.lblDSOResult.Text = "DSO Computing Time:";

//

// txtClientServerResult

//

this.txtClientServerResult.Location = new System.Drawing.Point(168, 45);

this.txtClientServerResult.Name = "txtClientServerResult";

this.txtClientServerResult.Size = new System.Drawing.Size(100, 20);
```

```
this.txtClientServerResult.TabIndex = 1;

//

// txtDSOResult

//

this.txtDSOResult.Location = new System.Drawing.Point(168, 15);

this.txtDSOResult.Name = "txtDSOResult";

this.txtDSOResult.Size = new System.Drawing.Size(100, 20);

this.txtDSOResult.TabIndex = 0;

//

// dgvNeighbors

//

this.dgvNeighbors.ColumnHeadersHeightSizeMode =

System.Windows.Forms.DataGridViewColumnHeadersHeightSizeMode.AutoSize;

this.dgvNeighbors.Columns.AddRange(new

System.Windows.Forms.DataGridViewColumn[] {

this.NodeName,

this.NeighborList});

this.dgvNeighbors.Location = new System.Drawing.Point(12, 596);

this.dgvNeighbors.Name = "dgvNeighbors";

this.dgvNeighbors.Size = new System.Drawing.Size(249, 155);

this.dgvNeighbors.TabIndex = 18;

//

// NodeName
```

```
//

this.NodeName.HeaderText = "Node";

this.NodeName.Name = "NodeName";

this.NodeName.ReadOnly = true;

//

// NeighborList

//

this.NeighborList.HeaderText = "Neighbors";

this.NeighborList.Name = "NeighborList";

this.NeighborList.ReadOnly = true;

//

// btnValidation

//

this.btnValidation.Location = new System.Drawing.Point(8, 289);

this.btnValidation.Name = "btnValidation";

this.btnValidation.Size = new System.Drawing.Size(132, 30);

this.btnValidation.TabIndex = 19;

this.btnValidation.Text = "Build Validation";

this.btnValidation.UseVisualStyleBackColor = true;

this.btnValidation.Click += new System.EventHandler(this.btnValidation_Click);

//

// btnRunValidation

//
```

```
        this.btnRunValidation.Location = new System.Drawing.Point(8, 322);

        this.btnRunValidation.Name = "btnRunValidation";

        this.btnRunValidation.Size = new System.Drawing.Size(132, 30);

        this.btnRunValidation.TabIndex = 20;

        this.btnRunValidation.Text = "Run Validation";

        this.btnRunValidation.UseVisualStyleBackColor = true;

        this.btnRunValidation.Click += new

System.EventHandler(this.btnRunValidation_Click);

        //

        // Simulator

        //

        this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);

        this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;

        this.ClientSize = new System.Drawing.Size(1093, 741);

        this.Controls.Add(this.btnRunValidation);

        this.Controls.Add(this.btnValidation);

        this.Controls.Add(this.dgvNeighbors);

        this.Controls.Add(this.groupBox1);

        this.Controls.Add(this.lblLatency);

        this.Controls.Add(this.txtLatency);

        this.Controls.Add(this.btnClientServer);

        this.Controls.Add(this.dgvResults);

        this.Controls.Add(this.btnDSO);
```

```csharp
            this.Controls.Add(this.btnReset);

            this.Controls.Add(this.drawingArea);

            this.Controls.Add(this.btnRun);

            this.Controls.Add(this.txtSizeofData);

            this.Controls.Add(this.lblSizeofData);

            this.Controls.Add(this.txtSensors);

            this.Controls.Add(this.lblSensors);

            this.Name = "Simulator";

            this.Text = "Distributed Shared Optimization Simulator";

            ((System.ComponentModel.ISupportInitialize)(this.drawingArea)).EndInit();

            ((System.ComponentModel.ISupportInitialize)(this.dgvResults)).EndInit();

            this.groupBox1.ResumeLayout(false);

            this.groupBox1.PerformLayout();

            ((System.ComponentModel.ISupportInitialize)(this.dgvNeighbors)).EndInit();

            this.ResumeLayout(false);

            this.PerformLayout();

        }
        #endregion

        private System.Windows.Forms.Label lblSensors;

        private System.Windows.Forms.TextBox txtSensors;

        private System.Windows.Forms.TextBox txtSizeofData;

        private System.Windows.Forms.Label lblSizeofData;

        private System.Windows.Forms.Button btnRun;
```

```
private System.Windows.Forms.PictureBox drawingArea;

private System.Windows.Forms.Button btnReset;

private System.Windows.Forms.Button btnDSO;

private System.Windows.Forms.DataGridView dgvResults;

private System.Windows.Forms.DataGridViewTextBoxColumn Node;

private System.Windows.Forms.DataGridViewTextBoxColumn DataSize;

private System.Windows.Forms.DataGridViewTextBoxColumn Time;

private System.Windows.Forms.Button btnClientServer;

private System.Windows.Forms.TextBox txtLatency;

private System.Windows.Forms.Label lblLatency;

private System.Windows.Forms.GroupBox groupBox1;

private System.Windows.Forms.Label lblClientServerResult;

private System.Windows.Forms.Label lblDSOResult;

private System.Windows.Forms.TextBox txtClientServerResult;

private System.Windows.Forms.TextBox txtDSOResult;

private System.Windows.Forms.DataGridViewTextBoxColumn NeighborsList;

private System.Windows.Forms.DataGridView dgvNeighbors;

private System.Windows.Forms.DataGridViewTextBoxColumn NodeName;

private System.Windows.Forms.DataGridViewTextBoxColumn NeighborList;

private System.Windows.Forms.Button btnValidation;

private System.Windows.Forms.Button btnRunValidation;

private System.Windows.Forms.Label lblBitPerSec;

private System.Windows.Forms.TextBox txtBitPerSec;
```

```
        private System.Windows.Forms.Label lblNodeDistance;

        private System.Windows.Forms.TextBox txtNodeDistance;

    }

}
```