THESIS

TOPOLOGY INFERENCE OF SMART FABRIC GRIDS - A VIRTUAL COORDINATE BASED APPROACH

Submitted by

Gayatri Arun Pendharkar

Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2020

Master's Committee:

Advisor: Anura P. Jayasumana

Anthony A. Maciejewski
Yashwant K. Malaiya

ABSTRACT

TOPOLOGY INFERENCE OF SMART FABRIC GRIDS - A VIRTUAL COORDINATE BASED APPROACH

Driven by increasing potency and decreasing cost/size of the electronic devices capable of sensing, actuating, processing and wirelessly communicating, the Internet of Things (IoT) is expanding into manufacturing plants, complex structures, and harsh environments with the potential to impact the way we live and work. Subnets of simple devices ranging from smart RFIDs to tiny sensors/actuators deployed in massive numbers forming complex 2-D surfaces, manifolds and complex 3-D physical spaces and fabrics will be a key constituent of this infrastructure. Smart Fabrics (SFs) are emerging with embedded IoT devices that have the ability to do things that traditional fabrics cannot, including sensing, storing, communicating, transforming data, and harvesting and conducting energy. These SFs are expected to have a wide range of applications in the near future in health monitoring, space stations, commercial building rooftops and more.

With this innovative Smart Fabric technology at hand, there is a need to create algorithms for programming the smart nodes to facilitate communication, monitoring, and data routing within the fabric. Automatically detecting the location, shape, and other such physical characteristics will be essential but without resorting to localization techniques such as Global Positioning System (GPS), the size and cost of which may not be acceptable for many large-scale applications. Measuring the physical distances and obtaining geographical coordinates becomes infeasible for many IoT networks, particularly those deployed in harsh and complex environments. In SFs, the proximity between the nodes makes it impossible to deploy technology like GPS or Received Signal Strength Indicator (RSSI) for distance estimation. This thesis devises a Virtual Coordinate (VC) based method to identify the node positions

and infer the shape of SFs with embedded grids of IoT devices.

In various applications, we expect the nodes to communicate through randomly shaped fabrics in the presence of oddly-shaped holes. The geometry of node placement, the shape of the fabric, and dimensionality affect the identification, shape determination, and routing algorithms. The objective of this research is to infer the shape of fabric, holes, and other non-operational parts of the fabric with different grid placements. With the ability to construct the topology, efficient data routing can be achieved, damaged regions of fabric could be identified, and in general, the shape could be inferred for SFs with a wide range of sizes. Clothing and health monitoring being two essential segments of living, SFs that combines both would be a success in the textile market. SFs can be synthesized in space stations as compact sensing devices, assist in patient health monitoring, and also bring a spark to the showbiz.

Identifying the position of different nodes/devices within SF grids is essential for applications and networking functions. We study and devise strategic methods for localization of SFs with rectangular grid placement of nodes using the VC approach, a viable alternative to geographical coordinates. In our system, VCs are computed using the hop distances to the anchors. For a full grid (no missing nodes), each grid node has predictable unique VCs. However, a SF grid may have holes/voids/obstacles that cause perturbations and distortion in VC pattern and may even result in non-unique VCs. Our shape inference method adaptively selects anchors from already localized nodes to compute VCs with the least amount of perturbation. We evaluate the proposed algorithm to simulate SF grids with varied sizes (i.e. number of nodes) and the number of voids. For each scenario, e.g. a SF grid with length X breadth dimensions - 19X19, 10% missing nodes, and 3 voids, we generate 60 samples of the grid with random possible placements and sizes of voids. Then, the localization algorithm is executed on these grids for all different scenarios. The final results measure the percentages of localized nodes as well as the total number of elected anchors required for the localization.

We also investigate SF grids with triangular node placement and localization methods for

the same. Additionally, parallelization techniques are implemented using an Message Parsing Interface (MPI) mechanism to run the simulations for rectangular and triangular grid SFs with efficient use of time and resources. To summarize, an algorithm was presented for the detection of voids in smart fabrics with embedded sensor nodes. It identifies the minimum set of node perturbations to be consistent with VCs and adaptively selects anchors to reduce uncertainty.

ACKNOWLEDGEMENTS

As I approach the successful completion of my Master's Thesis, I experience a lot of mixed emotions. I fall short of words when I say that this journey would not have been possible without the constant support and guidance of my advisor, Dr. Anura Jayasumana. He has been the most important person in shaping my work and my personality. He helped me see my abilities in times when I lost faith and always had confidence in the paths that I pursued through this process of personal change and development. I am deeply grateful to have found an opportunity to work with an amazing mentor, insightful researcher, and a person with a beautiful soul.

I sincerely thank my committee members, Dr. Anthony Maciejewski for his valuable teachings and insights in the field of Robotics and Dr. Yashwant Malaiya for his teachings in Fault-Tolerant Computing. They helped me learn and explore various domains outside my research topic. I would also like to thank Dr. Jade Morton for introducing me to the innovative field of Global Navigation Satellite Systems and Dr. Ann Hess for helping me understand the importance and applicability of statistical analysis in my research. My deepest gratitude to my colleagues; Gunjan Mahindre who has been a mentor and a great friend, Savini Samarasinghe, Shashika R. Muramudalige, Sridhar Ramasamy, and Aly Boud. Thank you so much for being patient with my doubts, questions and giving repeated reviews and feedback on my work. All these people have contributed to the foundations of my career and future.

A lot of friends in my life have made this journey so much easier, joyful, and worth the time, my RamFam, my home away from home. I will always cherish all the vacations, birthdays, parties, and so many other times that I spent with them and look forward to more. Difficult times make it hard to look at the positive side of things and see the light at the end of the tunnel. I am so thankful to have Yashad Samant by my side during these times providing love, support, and help in seeing my worth. Last but not the least, my family

is the one who made me the person I am today. They gave me the strength to keep going and fighting. My grandpa, Baburao Pendharkar always had a smile on his face irrespective of him missing me dearly which motivated me to complete my work righteously. My father, Arun taught me that nothing comes easy and there is no limit to what you can achieve with hard work. My mother, Jayashree set an amazing example for me to look up to. She has relentlessly worked all her life; juggling the family and her career to give us a wonderful upbringing. My little sister, Rutuja has been my greatest critique. She has been the most understanding, non-judgemental, and caring person whom I could rely on to look after my family while I worked here care-free.

DEDICATION

To my Grandparents, Baburao and Vimal; my parents, Arun and Jayashree; and my sister, Rutuja

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The textile industry has expanded and altered beyond recognition due to industrialization and innovative manufacturing techniques. In addition to that, the increasing applications for Wireless Sensor Networks (WSNs) have led to constant miniaturization and cost reduction of the electrical sensors. Smart Fabrics (SFs) are a notable breakthrough in the textile industry with their capabilities to perform tasks that traditional fabrics cannot. SFs are embedded with small digital components like sensors, batteries or similar electronic components called Smart Fabric Nodes. These SF systems provide an added value to the customer due to the ability of embedded nodes to sense, communicate, transform, grow, compute and store data. These advances in SF technology is a seamless blend of two approaches – "technology-push", in which the technological innovations are leading to new systems and products, and the "application-pull", due to the rising demand of users for solutions. Technology has now enabled us to carry around our own personal Body Area Network (BAN) [3]. SFs are assorted into a pool of applications in the field of sports, entertainment, media, medical, and military.

According to the Global Health Observatory Data on Current health expenditure (CHE) as a percentage of gross domestic product (GDP) (%) for the year of 2015, $7.2 trillion or 10% of the global GDP was spent on healthcare in 2015 [4]. The U.S. spent about 16.8 percent of its GDP on healthcare in 2015. As per the reports by the Centers for Medicare & Medicaid Services (CMS), U.S. healthcare spending marked an increase in 2017 accounting to 17.9 percent of the GDP [5]. The WHO reports also mention the increase in the global aging population as a severe issue. It is said that the amount of people over the age of 60 years is set to double by 2050 leading to a surge in healthcare expenditures. Accounting to these

statistics and the current medical needs, it is projected that the healthcare expenditures will reach almost 20% of the GDP in the next few years, threatening the welfare of the economy [6]. These anticipated changes have generated a huge demand for affordable and proactive healthcare services. Numerous Smart Wearable Systems (SWS) have been developed for ubiquitous health monitoring, ambient assisted living or even motion capture. Due to their ability of remote monitoring, the expense of personal medical visits can be highly minimized.

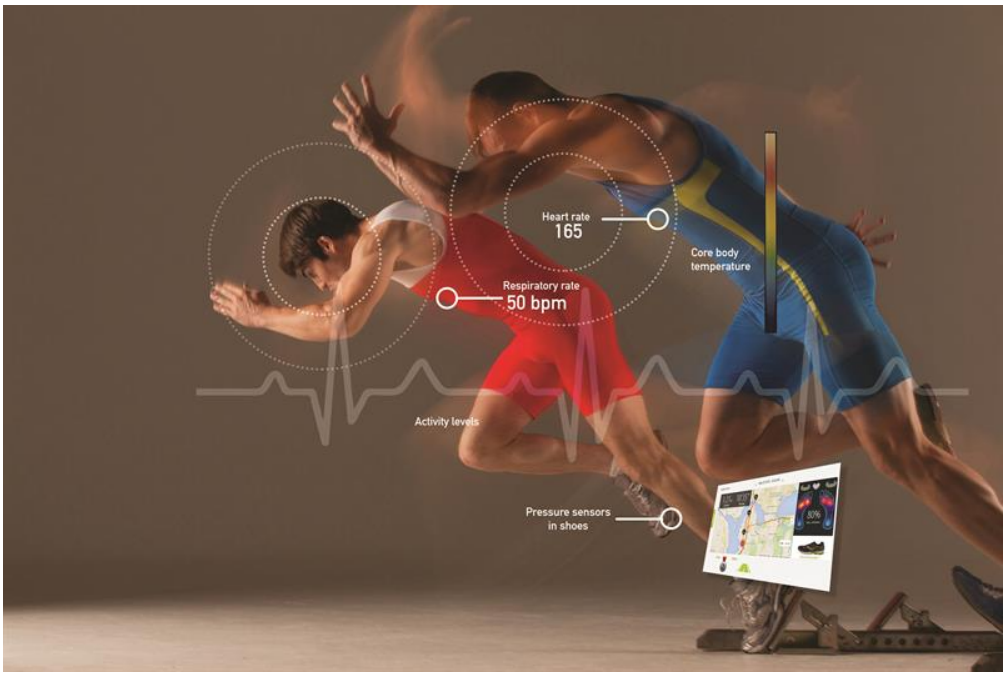


Figure 1.1: Smart Fabric in sportswear [1]

Athletes, health professionals, and people who exercise on a regular basis would highly appreciate SF products. Figure 1.1 shows an illustration of smart fabric wearables for athletes. It would be vital even on a preliminary basis to monitor heart rate, calorie count, respiratory rates, core body temperatures, etc. Marathon runners or athletes participating in tournaments such as the Olympics would highly benefit from this technology. For tournaments of this level, it would be crucial to obtain data at high accuracy to determine the participant's performance and also to monitor their health. It is conceivable that future smart suits can convey information from one user to the other, e.g., allowing a trainer to detect the pressure or pain felt by an athlete.

Individuals in professions such as soldiers, submarines, astronauts, and miners are susceptible to dangers, fatigue and sensitive environments all the time. Additionally, they have to attend vigorous training and keep up with a heavy schedule. With the amount of mobility and vulnerability in the occupation, it could be unsafe and infeasible to have medical personnel come in for health check-ups. This demands an application to remotely monitor the health of the individuals in these sensitive areas. A vest can be designed using SF to monitor the vitals of the patients and provide them with feedback to help the individuals maintain optimal health status.

This thesis anticipates a future in which SF would exist with a large number of embedded devices. In such fabrics, it would be desirable to determine the shapes of the fabric automatically and for the network of devices to be self-configurable. We investigate an approach centered on Virtual Coordinate System (VCS) which is based on connectivity rather than physical distance metric. This is a major advantage as it reduces the cost and complexity.

1.1 Virtual Coordinate System for Smart Fabrics

Sensor nodes in SF need to be localized using scalable and robust algorithms and protocols for computational purposes. Localization and routing are among the essential functions for SF network operations. Node localization in a sensor network alludes to identifying the positional coordinates of network nodes. In complex and condensed networks, location information by itself cannot facilitate routing. On the other hand, obtaining location information in the form of physical coordinates is expensive and unreliable at best, or highly infeasible. Thus, we use Virtual Coordinates (VCs) which are economical to compute and less susceptible to parametric variations and interference, and in many scenarios, provide equal or better routing performance compared to physical coordinates.

The VC system essentially defines node data points in the given space identified for a specified number of base nodes called anchors. As per the Figure 1.2, node n is a node in the

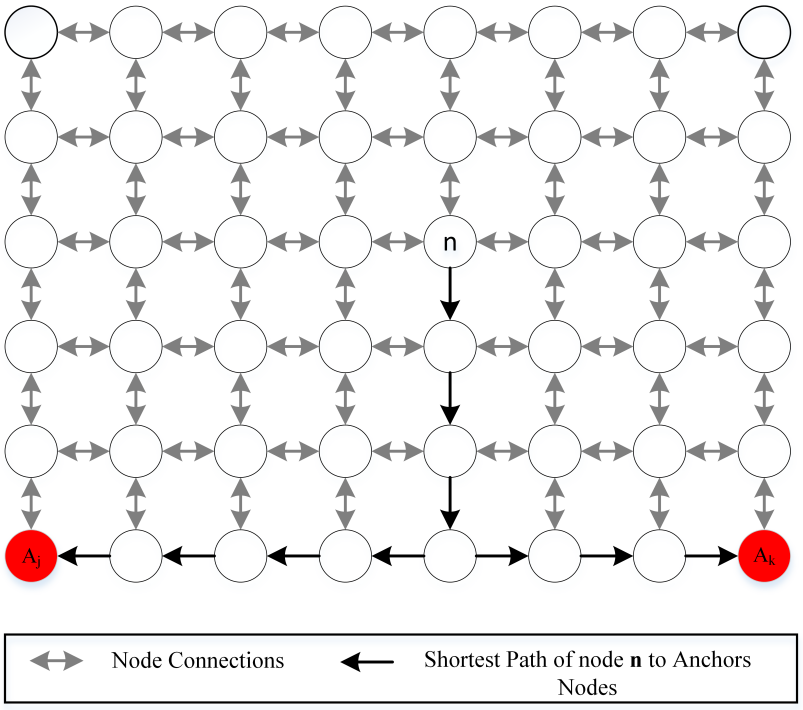


Figure 1.2: Virtual Coordinate System (VCS) with two anchors A_j and A_k for a rectangular grid

network, A_j and A_k are the anchors. The VCS for this network is a hashmap with structure,

$$V_{n_1} : (h_{n_1 A_j}, h_{n_1 A_k}), V_{n_2} : (h_{n_2 A_j}, h_{n_2 A_k}), ..., V_{n_N} : (h_{n_N A_j}, h_{n_N A_k}) \tag{1.1}$$

where,

$N \leftarrow$ Total number of network nodes

$n, A_j, A_k \in N$

These VC vectors for smart fabrics are computed using node connectivity (hops) and anchor information. Thus, for node n in 1.2, the VCs are,

$$(h_{n_{A_j}}, h_{nA_k}) = (7, 6) \tag{1.2}$$

In this thesis, we use the concept of VCs to characterize the nodes in a given SF. Our goal is to use these VCs to identify embedded voids in the SF. These voids can be specifically introduced or caused by failure of nodes.

1.2 Outline

The rest of the thesis is organized as follows. Chapter 2 entails the literature review in the field of Smart Fabrics. Chapter 3 explains the problem statement and motivation for the thesis. Chapter 4 describes the Grid Simulator designed for the purpose of detecting voids for this thesis. Chapter 5 narrates the algorithm for Adaptive Localization. Chapter 6 discusses the proposed localization model for Triangular Grid Smart Fabrics. Chapter 7 addresses the parallelization module used speedup the computations. Chapter 8 concludes the thesis.

CHAPTER 2

LITERATURE REVIEW

This section comprises background research on Smart Fabrics; the recent advances and prospective research on them. We also review Virtual Coordinate Systems and the existing models that use them. Lastly, we present the literature on anchor election and placement. It discusses algorithms to achieve efficient localization. The list of abbreviations used in the description is summarized in Table 2.1.

Table 2.1: List of Abbreviations

SF	Smart Fabric
WSN	Wireless Sensor Network
IoT	Internet of Things
VCS	Virtual Coordinate Systems
MPI	Message Parsing Interface
VC	Virtual Coordinate
GR	Greedy Routing
BFS	Breadth First Search
RSSI	Received Signal Strength Indicator
ENS	Extreme Node Search

2.1 Smart Fabrics

Miniaturized wireless sensors with the capability to sense, compute, store and forward the data in the fabric can be placed strategically over the human body to monitor the vitals and have a huge demand in the medical field. The renowned wearable textile project by

the Information and Communication (ICT) program of the EC, namely MyHeart [7] unites inter-disciplinary research institutes, academia, and medical centers in Europe to fabricate solutions for cardiovascular diseases. This project was an early effort to empower the citizens to take control of their own health through the use of smart wearable systems. Such systems would potentially allow preventing at-risk diseases by early-diagnosis.

Another venture funded by the EU, namely WEALTHY [8] involves SF technology, advanced signal processing techniques and modern telecommunication systems. Conducting and piezo-resistive materials are used to fabricate the sensors and the connecting links between them. This prototype senses, pre-processes, transmits, processes and provides data management. It manages to achieve simultaneous recording of multiple biomedical signals like ECG, EMG, respiration rate, and body movements.

Gesture capture or recognition via SFs can be used in academia, sporting and physiotherapy [9] to determine useful information about the users' body movement. Smart clothing in association with Cloud and Big Data can be used for monitoring health through mobility. Applications such as medical emergency response, emotion care, disease diagnosis, and real-time tactile interaction have introduced that work with big data clouds [10]. These applications, for instance, can record electrocardiograph (ECG) signals collected through the SF to monitor moods and emotions of the subject. A prototype smart shirt described in [11] using SF for ubiquitous health and activity monitoring. It measures ECG and acceleration signals for continuous and real-time health monitoring and has conductive fabrics to receive the body signal as electrodes. The measured physiological ECG and activity data are transmitted in an ad-hoc network via IEEE 802.15.4 communication standard through compatible miniature devices to a base-station or a server PC for remote monitoring. The Lab of tomorrow designed a Smart Vest for remote health monitoring of users with less intervention and discomfort in their daily movements. As per the experimental results, the vest was able to extract accurate data better than other systems that can be uneasy to wear [11].

Several localization techniques have been proposed for WSNs to compute the physical

coordinates of a node. Insight about the physical coordinate of the node helps to reconstruct the shape of the grid and be aware of the exact location on the Fabric from where the data is sensed or collected. The traditional methods use GPS and Receiver Signal Strength Indicator (RSSI) for localization purposes [12] [13] [14]. However, GPS becomes obsolete in these Fabric Networks due to limitations in cost per unit and energy budget demanded by GPS devices. Additionally, due to the dense placement of nodes on Fabric, the GPS resolution may not be sufficient for localization. Alternatively, Time-of-Arrival (TOA) or RSSI are used to estimate distances to other nodes, and thereby obtain node positions. However, these techniques fail to provide accuracy and are susceptible to phenomena such as noise, fading, multipath and interference, and errors in localization tend to accumulate. These techniques could potentially fail to work in large-scale networks outside laboratory settings, and of course in harsh and complex environments.

The advancements in the sensor network and textile industry fields have fueled the research to find definitive solutions to the newly introduced challenges and issues. In healthcare, several applications and prototypes are proposed to monitor body issues like diabetes [15], cardiac arrest early detection [16], vital signs monitoring using 3G networks [17], monitoring multiple biomedical signs of the body [18] [19]. In addition to health monitoring, smart fabric technology can be used for smart fitness and training for athletes, emotion monitoring [10] and for remotely monitoring the soldiers' locations and physical activity [11].

2.2 Virtual Coordinate Systems

To subjugate problems due to physical coordinates in IoT, developments are in place that can facilitate operations such as routing and self-organization without any physical location information. In addition, these coordinates can also provide a passage to localization [20]. A VCS structure defined in 1.1 has all nodes in the sensor network with a coordinate vector of dimension equal to the number of anchors which may or may not be different from the SF grid space dimension. VCS elects the anchors and its VCs based on parameters such

as connectivity, packet loss, topology and more. VCS usually are associated with Euclidean frameworks where node connectivity is preserved (hops) but not the actual physical distances. Thus, a measure of 1 hop does not necessarily correspond to 1 unit.

Several techniques have emerged that use VCS for efficient localization and routing. Graph embedding is a technique that uses VCS wherein network nodes hold node connectivity information that is embedded inside them [21]. In this case, each node carries a map or a sub-map of the network topology with connectivity information which can be used for routing purposes while capturing the voids or patches in the grid. The most commonly used VC assignment techniques elect a set of anchor nodes for constructing the coordinate framework. These anchors are network nodes that are elected randomly or by a defined process or algorithm as shown in Figure 1.2. The number of anchors determine the dimension of the vector coordinate. Greedy Forwarding (GF) or some other technique is used in routing algorithms using the defined VCs. These VCs are also used for distance evaluation as well as for node identification [22].

VCs could provide utility in the form of network parameters in the Internet and overlay networks [23]. These VCs are computed from network properties such as the latency, packet loss or any other network measurement parameters. It is vital in network operations to preserve the network topology like in overlay networks it is needed to optimally trace the neighboring nodes and communication paths keeping in account the proximity, network delays, and round-trip time (RTT) [23]. However, capturing such information in real-time would cause measurement traffic in the network and result in a large overhead. To overcome these issues, Network Coordinate Systems (NCS) have been proposed. This technique couples network measurement parameters to the parameters of interest for the VCS. E.g. Maximum Likelihood Topology Maps [24] proposes a packet reception probability function that helps capture the graph topology. Topology preserving maps [25] too retain the graph topology, yet are also homeomorphic to physical layout [2].

A.-M. Kermarrec et al. [26] propose an algorithm that helps the network nodes to self-structure thus identifying its position and collaboratively impose a geometric structure to the network. Shah and Sardana [27] have invented a parameter computation and search approach using VCS for IoT. This method uses VCS to compute and obtain network statistics like delays, latency, etc. using a decentralization technique. Li et al. [28] present a fascinating real-time raveling path tracking algorithm for smart vehicles that have encoders installed on opposite sides of the wheels that computes the distance that a vehicle rolled over. Thus, the VCS for this system is fixed on the ground. The results comprise the vehicle path contributed by the position and heading angle of the vehicle. The techniques give efficient results even in the presence of obstacles, fog, rain, snow, etc.

Leone and Samarsinghe [29] propose an algorithm to use Greedy Routing (GR) on a Virtual Raw Anchor Coordinate (VRAC) System. The VRAC coordinate is calculated using roughly three raw node distances to be used as coordinates. Given that a saturated graph or network exists, greedy routing provides guaranteed packet delivery using VRAC system [2]. Jayasumana et al. [30] proposed techniques for the extraction of topology maps of the network graphs using anchor-based VCs. This paper uses the low-rank matrix completion theory in which the topology network maps are extracted for 2-D or 3-D networks by using the partial VC information. However, it is observed that geographical information alone does not suffice to overcome obstacles or voids in-network as they disrupt the routing algorithms. Sheu et al. [31] have presented a distributed algorithm called Hexagonal Virtual Coordinate (HVC) to construct a VCS. Using this HVC, the source node is able to find an auxiliary routing path to the defined destination. This algorithm overcomes the issue of obstacles as it provides with aptly placed anchors or landmarks spread across the network irrespective of any voids.

The listed VCS examples use diverse parameters and schemes to construct a coordinate framework. VCS has many advantages over traditional coordinate schemes like high routing capability with the ability to provide consistent performance regardless of voids and minimal localization errors. Issues such as identical coordinates and local minima can occur due to

lost directionality [20] in VCS. The problem of the identical coordinate election for multiple voids may happen due to an insufficient number of anchors and the local minima in these systems are virtual voids in the network.

The survey presented in the Appendix C "Virtual Coordinate Systems and Coordinate-Based Operations for IoT" gives a brief overview of many such VCS, their operations, and advantages and dis-advantages [2].

2.3 Anchor Selection and Anchor Placement

As discussed in Section 2.2 most of the VCs are elected based on a set of anchor nodes in the network. The coordinate dimension depends on the number anchors e.g. with M anchors, a node would have an M-tuple coordinate vector. The number of anchors and their placements plays a vital role in optimal localization and efficient implementation of routing algorithms using these coordinates. Many of the protocols have been evaluated with random anchor placement while others have specific declarations about the selection of anchors. Some algorithms propose placing the anchors on the network border given that we have knowledge about the border nodes [32]. Other techniques depend on electing anchors nodes with hop distances as far as possible from one another which would potentially make them fall on the borders. Additionally, the single anchor-based VCS [33] uses Depth First Search (DFS) to compute the node position coordinates wherein the anchor is placed at the center in this scheme.

2.3.1 Random Anchor Placement

This is the most common and straightforward procedure that elects random nodes from the network as the anchors. It works in a distributed manner such as each node carries a certain probability to become an anchor. Most of the time, the anchor nodes are chosen to be furthest apart from each other. The further the anchor nodes are located, it increases the coverage area for the coordinate computation for the algorithm. Another important aspect is to select the optimum number of anchors. There is no ideal number defined for the same

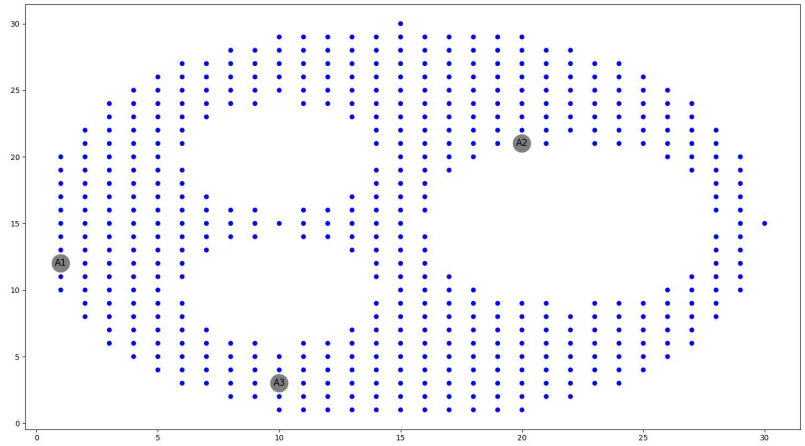


Figure 2.1: Random election of anchors in a network with 496 nodes

and it highly depends on the number of network nodes, network topology, the density of nodes in the network, etc. To avoid identical coordinate computation problems and achieve good performance, it is advised to select a reasonably high number of anchors rather than trying to minimize the number of anchors. However, as Figure 2.1 shows, even with random anchors chosen several hops away, it does not guarantee full network coverage.

2.3.2 Single Mobile based anchor

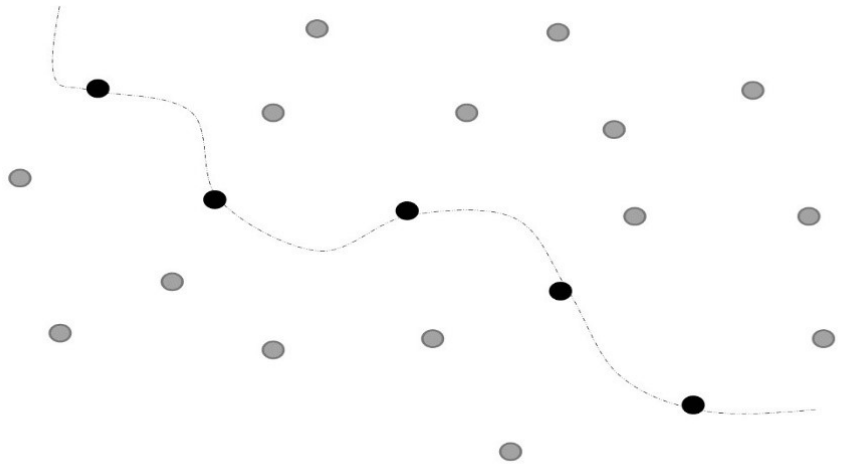


Figure 2.2: Single Mobile based Anchor

For the single mobile-based anchor, the physical coordinates for each node are derived from hop distance from an anchor node. This anchor node is a mobile robot that traverses around the network to assist with the coordinate assignment. The robot device is GPS-enabled and thus, the location is known. Using the location information for the robot, it is feasible to find out VCs for network nodes concerning the robot. During the traversal, the robot transmits its position coordinates to the neighboring sensor nodes that lie within its communication range. This distance between the robot transmitter and the sensor node n is determined using the RSSI. The algorithm computes Barycentric coordinates for the network nodes and a distributed routing algorithm is used [33]. Using this technique, unique VCs can be assigned for all the nodes in the network successfully. However, distance measurement error due to RSSI can lead to inaccuracy in results. Figure 2.2 shows the example of a single mobile anchor robot navigating through the sensor field.

2.3.3 Extreme Node Search (ENS)

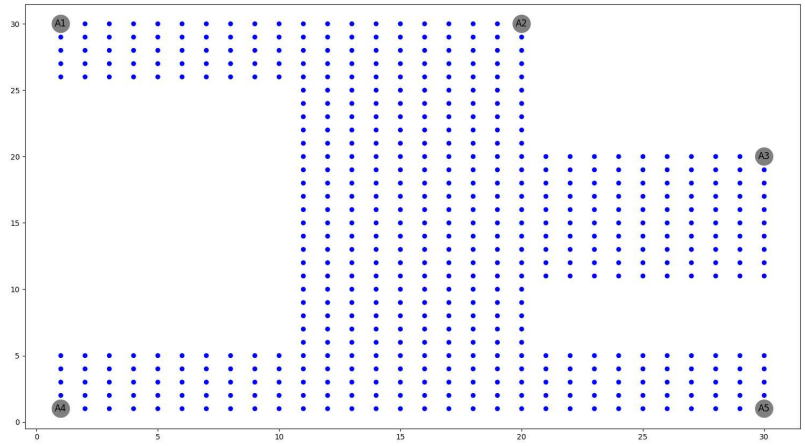


Figure 2.3: Anchor Placement using Extreme Node Search (ENS) algorithm [2]

This is a simple yet effective anchor election and placement scheme. Anchor election in this technique corresponds to choosing extreme nodes of the network like corner and border

nodes. Unlike the above-described algorithm, it does not rely on prior knowledge about node being on a border [20]. ENS follows three major steps for anchor election,

Step 1: Two random nodes are elected from the network to serve as initial anchors. The network is then flooded with beacons that originate from these two nodes hence providing every node a two-tuple VC. Using these coordinates, each node computes its DVC using the DVCS algorithm [22].

Step 2: In this step, each node identifies whether it is a local minimum/maximum in terms of the DVC within its h-hop neighborhood [22]. This is performed in parallel and involves communication between nodes in the h-hop neighborhood. The value of h is typically very small (range of 2–5), and the value typically determines the total number of self-identified extreme nodes.

Step 3: Any node that identifies itself as local minima or maxima in its h-hop neighborhood becomes a new anchor in the network. Thus, VC generation now begins with this newly elected set of anchors. As the VCs are pre-computed for the two random anchors elected originally, they could be considered as a part of the anchor set without any additional cost and overhead though they may not be extreme nodes. Figure 2.3 shows anchors in the network identified through ENS scheme for an odd-shaped network.

CHAPTER 3

PROBLEM STATEMENT AND CONTRIBUTION

As stated in Chapter 2, the demand for Smart Fabrics is increasing rapidly while they are also getting "smarter". As of today, the fashion industry sees the most applications of these textiles, along with medical and sportswear companies that have discerned the potential of this technology and have dived into it. Applications of SFs will extend far beyond such industries to areas such as construction, automobiles and military.t SFs provide an innovative field with high prospects of research and development to create sustainable and efficient products for mankind. They have empowered us to modernize the health appliances by integrating intelligent hardware and the Internet into the appliances.

Soldiers in combat demand for intelligent weapons for fighting and safeguard devices for survival. As of today, the safeguard clothing like the bullet-proof vests have several shortcomings. They can be uncomfortable to wear, intervene in natural body movements and weigh down the person wearing it due to the heavy texture. In the future, vests can be designed to accommodate a layer of SF with the sensors nodes attached below the bullet-proof surface. The actual vest would be designed from a transformable textured material that can mold its strength. This can be achieved by the alteration of the electron structure and is a breakthrough in material sciences [34]. Additionally, the SF sensors in the layer below would have a very high and fast sensing capability. Whenever a hard object such as a bullet strikes the outer surface of the Fabric, the sensors would send across an electrical transmission to the bullet-proof surface underneath requesting it to temper.

These ideas produce a great deal of motivation for experimentation and analysis in various fields and industries such as medical, fashion, military, smart wear, material sciences, etc.

3.1 Problem Statement

Automatic Identification of the shape of active areas of smart fabrics is important for many applications related to SFs. They include routing, faulty node location, and reconfiguration. The goal of this research is to detect the shapes of holes in a smart fabric with a regular grid (lattice) structure. Achieving this goal requires maximizing the localization for the sensor nodes embedded in the smart fabric. To be scalable for large fabrics, the nodes have to be simple and cost effective. Furthermore, we anticipate that the shape detection will need to be carried out on-demand via remote access. The nodes may be densely deployed in certain applications. These considerations preclude the use of devices such as GPS and manual intervention. Automatic algorithms are needed that does not rely on physical distance estimation, which is difficult or unfeasible in many scenarios.

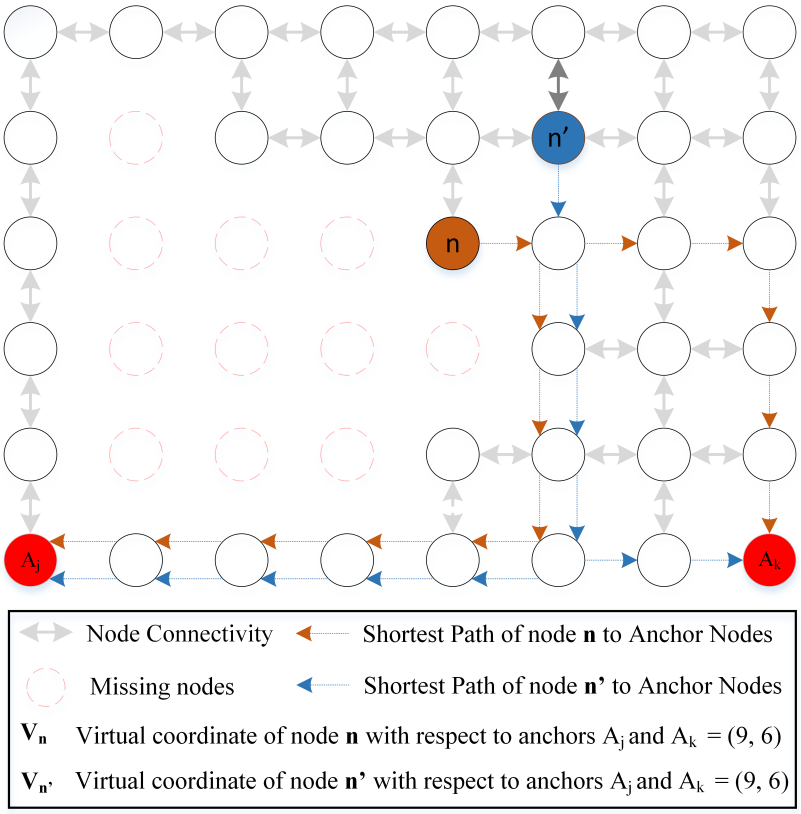


Figure 3.1: Virtual Coordinate System with grid patch or void

VCSs as discussed in Chapter 2 have resulted in several techniques for localization or node

identification. The most commonly discussed VCSs use anchor-based techniques. However, the major issue is such computations arise while dealing with grids that have voids or patches. In such grids, identical VCs and local maxima and minima issues are likely to occur. As Figure 3.1 shows, both nodes n and n' have identical VCs due to the void created by missing nodes in the grid. Usually, it is advisable to elect more anchors than less to avoid such problems. With more anchors, we gain more information about each node's placement in the grid.

This criterion of electing the right number and optimal placement for anchors is considered to be an NP-hard problem [35]. According to Wolfram Mathworld, a problem is NP-hard if an algorithm for solving it can be translated into one for solving any NP-problem (nondeterministic polynomial time) problem. NP-hard therefore means "at least as hard as any NP-problem," although it might, in fact, be harder [36]. The most common NP-hard problem is the Traveling Salesman in which the salesman has n cities to visit and there is a cost to travel a unit distance. The problem statement needs to find an itinerary for the salesman such as he only visits each city exactly once with minimum expenditure which would require checking all the combinations of cities depending on the path cost and number of places.

Similarly, the optimal anchor election and placement depends on several factors such as the size of SF, Shape of SF, Grid Placement, voids/patches/obstacles in SF, etc. Considering all of these parameters, it can be a NP-Hard problem to acquire an optimal set of anchors to achieve the best possible localization [35] [37]. NP-complete is the intersection of NP-hard problem and NP problem. It is the class of decision problems in NP to which all other problems in NP can be reduced to in polynomial time by a deterministic Turing machine.

3.2 Contribution

An algorithm for the localization of nodes and detecting the shapes of voids in a rectangular grid deployment of nodes in a SF with patches or voids is presented. Proposed method

uses anchor-based VCs and an adaptive and iterative anchor selection technique to gradually eliminate the uncertainty of node positions. We extend the approach to localization method for triangular grid fabrics and discuss an MPI technique for parallelization.

An adaptive anchor selection and placement algorithm is proposed that attempts to capture the voids in the grid and achieve maximum localization coverage for grid nodes. A set of linear equations is derived that identifies the VCs of every node. Using these VCs, We find the solution space that contains the location information for the node and then identify the solution that minimizes the perturbation from a full-grid of nodes. Additionally, we achieve further optimization based on the adjacency matrix using a neighbor detection algorithm.

The localization of the Smart Fabric nodes helps recognize the shape of the fabric or the voids. We initiate the algorithm with two known anchors and try to localize the grid nodes by adaptively adding additional anchors. A new anchor is chosen based on the node weight from the pool of nodes localized in the earlier step. Now, using these anchors, we try to localize the remaining grid nodes that could not be localized in the previous step. We repeat this localization technique until,

- All grid nodes are localized
- All nodes that can be used as anchors are exhausted.
- All grid nodes have been exhausted.
- Number of adaptive anchor addition steps reach $1/3^{rd}$ of the number of nodes in grid.

This algorithm is tested for different SF grid sizes with varying percentages of missing nodes and random void sizes and shapes. The grid simulator generates sample SF grids with random shaped voids for testing the adaptive localization method.

CHAPTER 4

SMART FABRIC SIMULATOR

4.1 Introduction

A Smart Fabric simulator is designed which generates a grid of sensor nodes satisfying a given set of specifications. The main focus of this work is on the generation and evaluation of SF grids with retained border nodes. When a SF is constructed, the placement of nodes can be varied with connectivity established amongst them via conductive threads. Here, we work on simulating grids with rectangular placement of nodes.

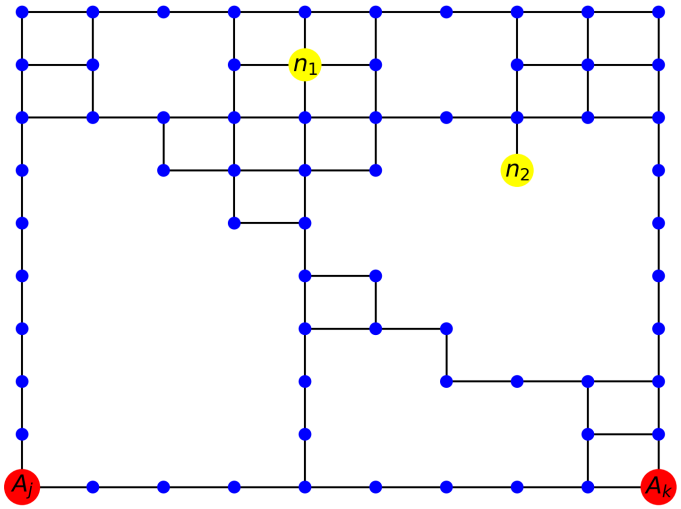


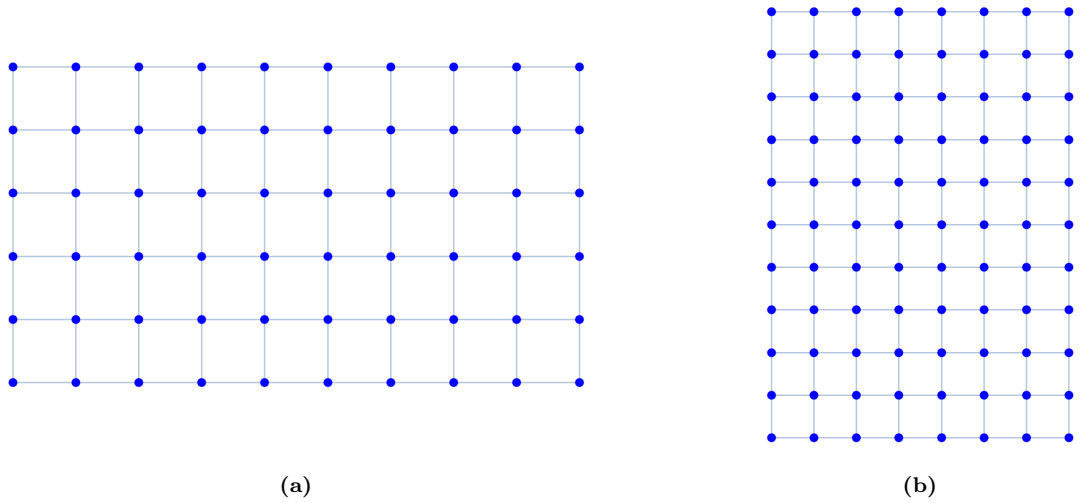
Figure 4.1: Rectangular-shaped Smart Fabric Grid

Figure 4.1 shows an example SF sensor grid with rectangular grid placement. In this placement, each of the sensor nodes is connected to at least one neighbor node in $+x, +y, -x$, or $-y$ direction. As shown in the Figure 4.1, node n_1 has 4 neighbors and n_2 has only 1 neighbor due to nature of the voids/patches in the network. Thus, an internal node will have a maximum of four neighbors, a border node will have a maximum of three neighbors and a corner node will have a maximum of two neighbors. In this and the following chapters, we

work with SF grids that posses an intact border, i.e., no missing nodes on the border and have varied percentages of internal missing sensor nodes with random shaped voids as shown in Figure 4.1. In the following sections, we discuss the algorithm to simulate these grids.

4.2 Simulator Design

As discussed, the SF Grids are designed with a rectangular-grid placement of nodes. These algorithms generate a grid based on basic inputs such as Length of the grid (L), Breadth of the network (B) and additional special inputs like percentage of missing nodes (P_M) and the number of voids (V). Figures in 4.2 show two Smart Fabric Grids of different sizes generated using the algorithm. The connection between two nodes is the sensor link or the conductive threads which carry information.



(a) **(b)**

Figure 4.2: Sample Smart Fabric Grids with dimensions (a) (9X5) and (b) (7X10)

However, Figure 4.2 show "full" SF Grids, i.e., with no missing nodes. Missing nodes in the grid cause a void which leads to a loss in connectivity information and deviation of virtual coordinates relative to a full grid. We design an algorithm that provides an approach to create a disintegrated network in accordance with the user input for the percentage of missing nodes (P_M) and the number of voids (V).

4.2.1 Grid Simulator

The grid simulator is implemented using specially designed modules, i.e., utility and helper modules. It follows a systematic approach verifying each step to design a grid true to the given specification.

Algorithm 4.2.1 Grid Simulator

Inputs
$L \leftarrow$ Grid Length
$B \leftarrow$ Grid Breadth
$P_M \leftarrow$ Percent missing nodes in Grid
$V \leftarrow$ Number of Voids in Grid

Outputs
$G_{AM} \leftarrow$ *Adjacency Matrix of Simulated Grid*

Variables
$I_N \leftarrow$ Inner nodes
$B_N \leftarrow$ Border nodes
$Dc_N \leftarrow$ Disconnected nodes
$A_M \leftarrow$ Adjacency Matrix for full grid

procedure SIMULATE A GRID WITH GIVEN *Inputs*
    Simulate Full Grid using L & B
    Recompute P_M for I_N
    Compute B_N and A_M
    while true **do**
        if G_{AM} is valid **then**
            break
        else
            Re-initialize B_N
        end if
        $V_D \leftarrow$ node distribution/void
        GenVoid using V_D
        Extract the G_{AM}
        Find Dc_N
    end while
    Return $\leftarrow G_{AM}$
end procedure

The algorithm 4.2.1 gives a brief overview of the grid simulator for generating SF grids. In the following sections, we will discuss in detail all the utility and helper modules used by

the simulator. The algorithm generates a full 2D coordinate set for the grid with the given length, breadth and other additional parameters. For example, the Figure 4.2 show the SF grid plots constructed using the physical coordinates.

$$T_N = (L + 1) * (B + 1) \tag{4.1}$$

$$I_N = T_N - (2 * (L + 1) + (2 * (B + 1) - 4)) \tag{4.2}$$

Here T_N is the total number of nodes in the grid. and I_N are the inner nodes in the SF ,i.e., not considering border nodes. From these values, the total number of nodes that must be deleted, De_N to create the void(s) are calculated i.e. P_M% of the I_N. Thus, for a 19X19 grid, if we wish to create 1 void with 10% missing nodes, the total number of nodes to be deleted would be 10% of 324 i.e. ≈ 32. In order to identify the nodes to be deleted, the algorithm first computes the A_M for the full grid.

After initializing and computing the necessary values, the algorithm repetitively runs to generate the grid with given number of voids V with the percent of missing nodes equal to P_M. The major steps involved here are listed as follows,

- Create a random node distribution per void using V, P_M and total number of nodes to be deleted.
- Generate a grid with voids and return the corresponding adjacency matrix.
- Find any disconnected nodes from the generated Adjacency matrix and delete them.

In the last step, the Grid Simulator performs a validation of the generated grid using its adjacency matrix to verify that the generated grid is true to the input specifications. The validation passes if equations 4.3 and 4.4 hold true.

$$G_{AM}.length = B_N + (I_N - De_N) \tag{4.3}$$

$$Dc_N = \varnothing \tag{4.4}$$

If validation fails, the variables are re-initialized and a new grid is simulated. The process repeats itself until a precise grid is generated that passes the validation test. As per the equations 4.3 and 4.4, the generated grid is valid if the length of the adjacency matrix for the grid is equal to the sum of all border nodes, and the difference between the inner nodes and deleted nodes. Also, the adjacency matrix for a valid grid must not have any disconnected nodes as they must have been deleted if present.

4.2.2 Utility Modules

The Grid Simulator uses utility modules for intermediate computational steps. In order to generate a grid with a void, the simulator initially generates a full grid with 2D coordinates using given L and B using the module. Then, another helper module is used to create the adjacency matrix and fetch the border nodes for the grid created by the earlier step. Additionally, the helper module for manual void creation deletes the nodes returned by the simulator to create a void. Further, there are modules that help with the computation of adjacency matrix and graph modules which provide assistance in finding the disconnected nodes in the simulated grid.

4.2.2.1 Adjacency Matrix

The adjacency matrix of a SF grid is the connection matrix for the data points (sensor nodes) placed adjacent to each other or the nodes that have connectivity established between them. It is a matrix with rows and columns with labels as the node IDs. The item value in the matrix is either 1 or 0 depending on whether the nodes are adjacent or not.

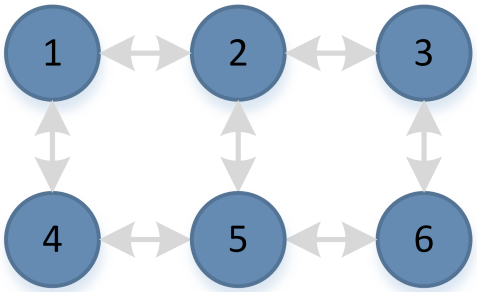


Figure 4.3: Adjacency Matrix Generation

For example, the SF grid in Figure 4.3 would have an adjacency matrix as follows,

$$AdjMat = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \tag{4.5}$$

Thus, a value of 1 conveys connectivity and 0 states no direct connection between the nodes. A module is presented that computes an adjacency matrix for the grid using given node positions or coordinates.

4.2.2.2 Graph Modules

The simulator uses a graph generator module which creates a graph of the grid using the python inbuilt library ***networkx*** [38]. This module facilitates the usage of graph algorithms. It also helps in easy plotting of nodes and edges or connections between them. Then a shortest path computation module uses the ***single_source_dijkstra*** method from the ***networkx*** library to discover the shortest path between nodes. These modules in combination provide information to derive disconnected nodes. If no path exists to the node, it is disconnected from the SF grid.

4.2.3 Helper Modules

Given the percentage of missing nodes (P_M) and the number of voids (V), a partition helper module creates a random distribution for nodes to be deleted. This module takes inputs De_N and V and returns a list with a random distribution of De_N nodes in V buckets. Following this, the void generator module iterates through the list creating one void at a time in the SF grid. The algorithm 4.2.2 states in brief the steps for void(s) creation.

Algorithm 4.2.2 Generate Void

```
Inputs
T_N ← Total number of nodes in Grid
B_N ← Border nodes
P_N ← Partition of nodes for V void(s)

Variables
De_N ← Deleted nodes
S_N ← Start Node for void

procedure GENERATE A VOID FOR ALL PARTITIONS
    Set De_N to empty
    for p in P_N do
        while true do
            Find S_N & validate
            if S_N is valid then
                break
            end if
        end while
        De_N+ = Find De_N using bfsVoid
        for d in De_N do
            B_N+ = Add new borders using voidBorders
        end for
    end for
    Create void using utility module
end procedure
```

For the void generation, the algorithm 4.2.2 uses the P_N information. For a given number of voids V if De_N number of nodes must be deleted to create a void, partition variable P_N

is defined as,

$$P_N = [p_1, p_2, .., p_V] \tag{4.6}$$

where $V \leftarrow$ Number of voids and,

$$\sum_{i=1}^{V} p_i = De_N \tag{4.7}$$

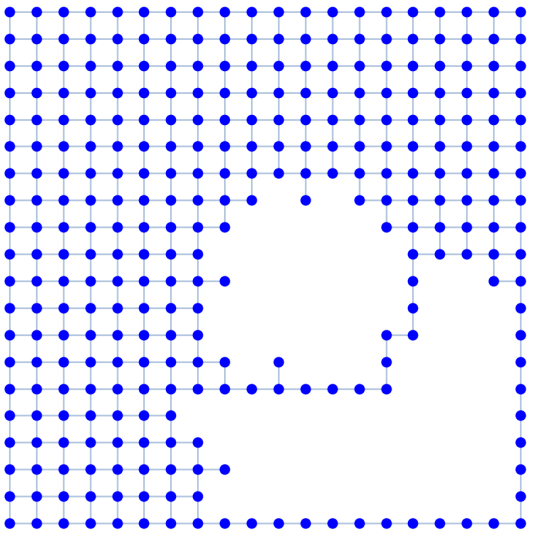


Figure 4.4: Smart Fabric Grid with Voids that have a shared border

The equations 4.6 and 4.7 imply that the summation of the partitions in P_N is equal to the total number of nodes that need to be deleted. The void generator loops through the partitions to create voids. Initially, it attempts to find a start node S_N to create a void such that the S_N is not a B_N and is not in any previously deleted nodes De_N. As soon as an eligible S_N is found, the algorithm uses the **Breadth First Search (BFS)** module for graph traversal to efficiently find out the nodes to be deleted to create a void in the given iteration of p in P_N. Once those nodes are derived, the border nodes for the newly created void are computed. These derived nodes are added to the original set of B_N. Hence, the border of the grid is updated with the internal border for voids as well as the outer grid border. It is important to update the borders to avoid unification of voids while creating

them. For example, Figure 4.4 shows a scenario for a 19X19 Smart Fabric grid with 2 voids due to 30% missing nodes. Here, as we keep track of updated border nodes as the void is being created, we can see distinct voids that share a border. This algorithm recurs for all partitions and ultimately provides coordinates for a grid with V voids due to P_M missing nodes.

We use the BFS Graph traversal algorithm that ensures each node is visited exactly once in the defined order. In a SF grid, to determine the nodes to be deleted, it is crucial to traverse through the graph marking a node as it is visited and making a decision about removing or keeping it. Also, the order in which the vertices are visited are important.

Algorithm 4.2.3 Breadth First Search (BFS) Graph traversal

Inputs
$S_N \leftarrow$ Start node to create void
$P \leftarrow$ Number of nodes to be deleted

Output
$De_N \leftarrow$ Nodes to be deleted

Variables
visited $\leftarrow$ Visited Nodes
queue $\leftarrow$ Queue of nodes to be checked

procedure TRAVERSE IN BFS FASHION
    Set *visited*, *queue* **to** $[S_N]$
    while *queue* exists **do**
        n = pop the first element from *queue*
        neighbors $\leftarrow$ Find random nodes to expand void
        for *ne* **in** *neighbors* **do**
            if *visited*.length $\geq P$ **then**
                Return *visited*
            end if
            if *ne* **not in** *visited* **then**
                Add *ne* to *visited* & *queue*
            end if
        end for
    end while
    Return *visited*
end procedure

In the BFS traversing algorithm, we initiate the traversal from the start node, S_N and keep traversing layer by layer, exploring the neighbour nodes of the source. Neighbors are the immediately connected nodes that can be reached in 1-hop. Once all the neighbors are explored, we move towards the next-level neighbour nodes (i.e. more than 1-hop distances).

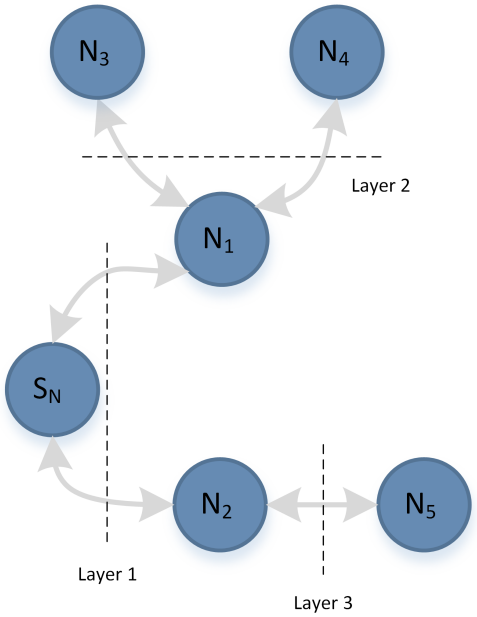


Figure 4.5: Breadth First Search Graph Traversal

As the Figure 4.5 shows, we have a SF grid or a graph with random nodes and edges. According to the algorithm 4.2.3, we select a start node S_N. We then explore the nodes within 1-hop for level 1 traversal. Graph traversal can be a cyclic and cause you to visit same node again. To avoid that, explored nodes are recorded as *visited*. So, neighbors of the source/start node are N_1 and N_2 which are stored in *queue* while being traversed in a defined order i.e. N_1 before N_2. Following the same, the child nodes of N_1, viz. N_3 and N_4 are traversed followed by child of N_2, viz. N_3. This allows us to explore the graph and choose a set of nodes to be deleted to create a void. Now, as the algorithm 4.2.3 states, there is an additional step where we compute the neighbors.

Since the motive is to create a random void, we do not wish the void to expand in an orderly fashion deleting all the neighbors every layer. Hence, we keep a track of *visited*

nodes and only choose the *neighbors* such that it is a subset or equal to all the neighbors belonging to the node. Additionally, any neighbors that are border nodes, already deleted or visited are excluded.

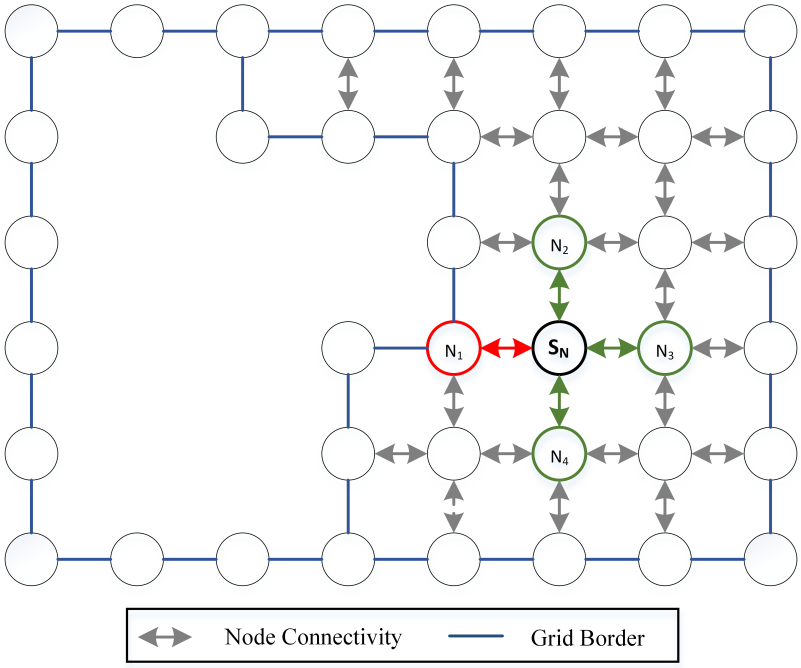


Figure 4.6: Expansion & Creation of new voids

Figure 4.6 shows a grid with a void already being created and also shows the border. Now, to create a new void from a start node, S_N, for BFS expansion, we can only choose neighbors N_{2-4} since N_1 is already a part of border nodes and so on for neighbors of these neighbors. However, to keep the approach random, we only chose random number of neighbors from the set of available neighbors i.e. in this scenario, we can choose 1, 2 or all 3 out of 3 available neighbors.

4.3 Results

A SF Grid simulator that creates random sized voids for given percentage of missing nodes, length and breadth dimensions has been presented. SF grids generated from these simulators are used as data-sets for the Adaptive Anchor placement algorithm for localization that is explained in the Chapter 5. Figures 4.7 and 4.8 show heterogeneously sized SF

grids with voids induced by specified percentages of missing nodes as generated by the Grid Simulator.

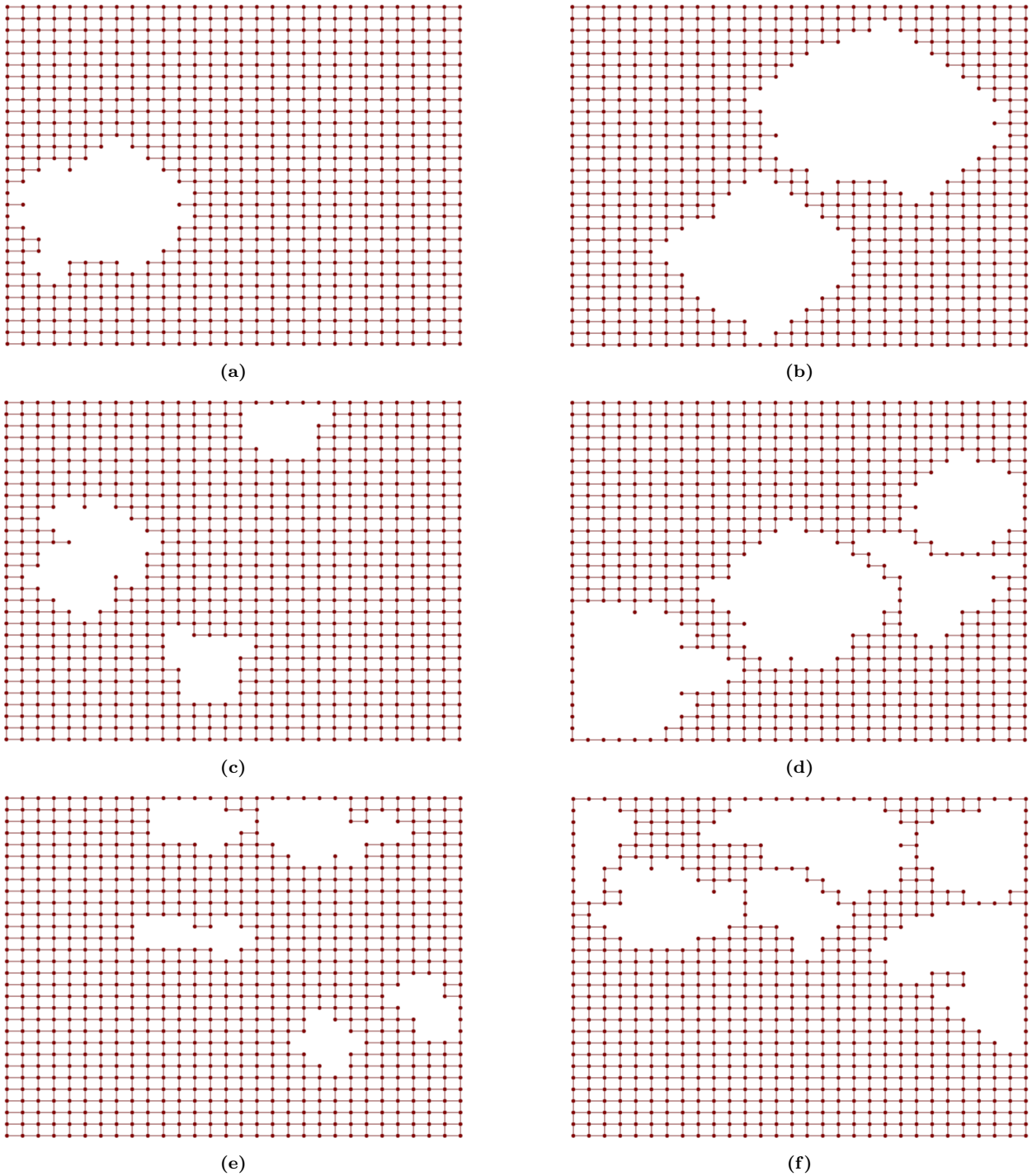


Figure 4.7: 29X29 Smart Fabric Grids with (a) 10% missing nodes with 1 void, (b) 30% missing nodes with 2 voids, (c) 10% missing nodes with 3 voids, (d) 30% missing nodes with 4 voids, (e) 10% missing nodes with 5 voids, (f) 30% missing nodes with 6 voids

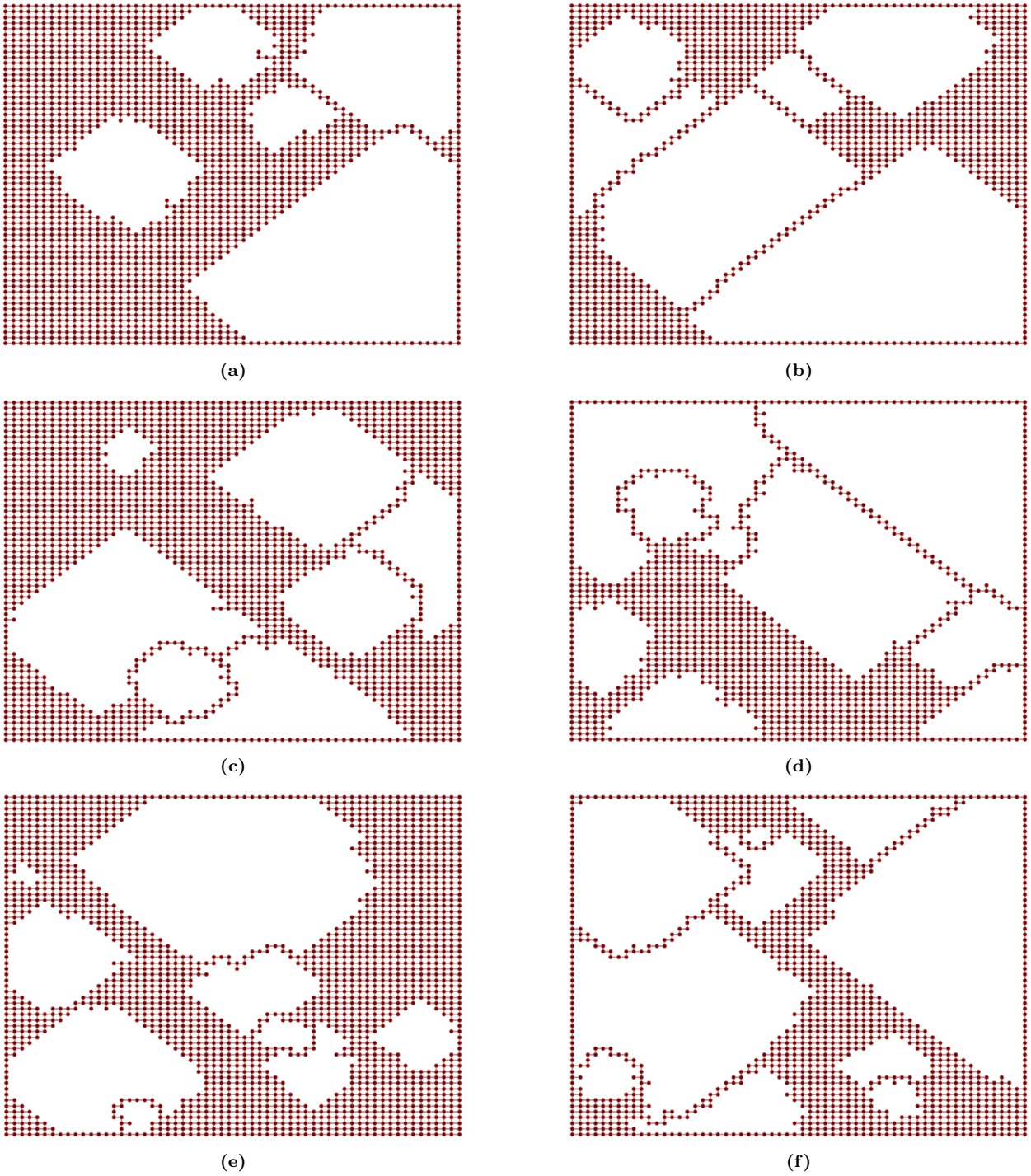


Figure 4.8: 59X59 Smart Fabric Grids with (a) 50% missing nodes with 5 voids, (b) 70% missing nodes with 6 voids, (c) 50% missing nodes with 7 voids, (d) 70% missing nodes with 8 voids, (e) 50% missing nodes with 9 voids, (f) 70% missing nodes with 10 voids

CHAPTER 5

ADAPTIVE LOCALIZATION IN SMART FABRICS WITH RECTANGULAR GRID

Localization is an important aspect of smart fabric sensor grids. It facilitates identification, estimation of spatial distribution of sensed data, evaluation of spatial correlation of data, estimation of grid density and coverage, and data routing. However, there are challenges for achieving accurate localization using traditional techniques such as GPS due to limitations on node battery, installation costs, and computational power. Additionally, for SF grids, it is even more difficult to have GPS embedded sensor nodes due to high density of the nodes that results in inadequate resolution of GPS due to node spacing, and unavailability of GPS in many indoor environments of interest. To overcome these issues, we use a Virtual Coordinate System which does not rely on any physical measurements to determine nodes' positions in SF sensor grids. Several VCSs have been proposed over time each with its advantages and disadvantages as discussed in Chapter 2. In this chapter, we propose an **adaptive localization** which identifies the locations of nodes in SFs where the rectangular grid formation is disrupted. e.g., due to holes in the fabric either intentionally created or caused by defects. The technique rely on an iterative technique, which identifies the node locations with some initial anchors and then adaptively places anchors to determine remaining node locations.

5.1 Location vs. Virtual Coordinates

Figure 5.1 shows a SF Grid with a length L and breadth B that comprises $N = 48$ nodes. A designated number of these nodes is elected as anchor nodes A (A_j, A_k). Our initial anchor placement consists of the two anchors that are placed at the extreme corners

along the length of the fabric. These nodes (also referred to as landmark nodes) are used for VCs computation. Every node has a 2-tuple virtual coordinate associated with two anchors. In case of a full rectangular grid, we can find the associated 2D position coordinates "(x, y)" for each node i.e. achieve localization of all the nodes. These coordinates can facilitate inference of topology of the SF grid. Given the grid topology in Figure 5.1, every node lies on the intersection of a grid made up of row and column.

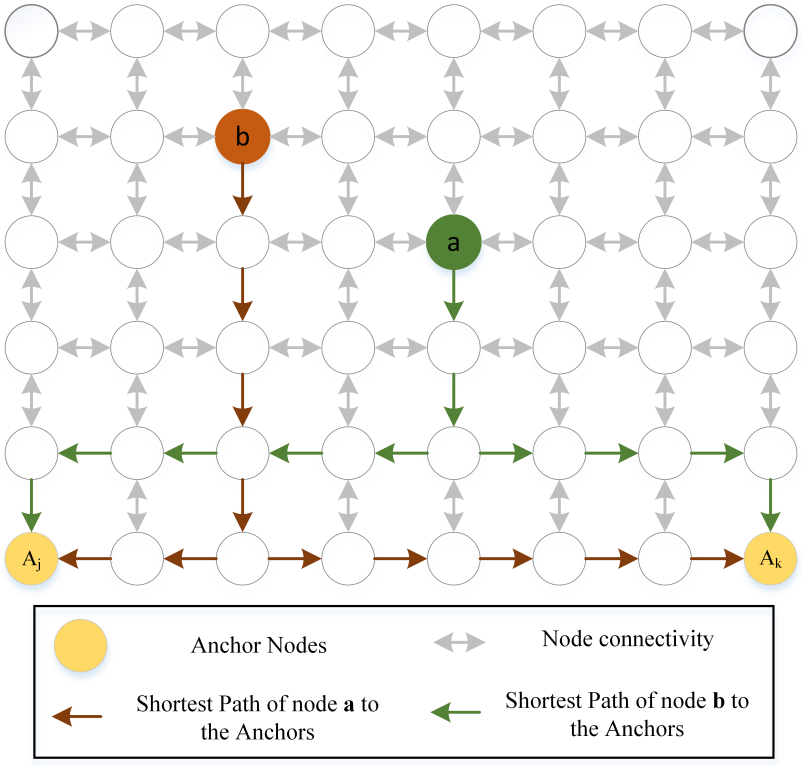


Figure 5.1: Addressing in a Complete Network

The anchor A_j is the base node of the topology is considered to be positioned at (0, 0). Consider a node a in the smart fabric network in Figure 5.1. The VCs for this node with reference to the anchors is $(h_aA_j,\ h_aA_k) = (7, 6)$. Additionally, the node a has physical coordinates $(x, y) = (4, 3)$. These physical coordinate positions can be determined uniquely as long as the shortest path exists to the anchors. Similarly, node y in the network has a VC of $(bA_j,\ bA_k) = (6,\ 9)$ and has a 2D physical coordinate of (x', y') = (2, 4). Using these physical coordinates, it is easy to compute the linear distance between the nodes x and

y. Additionally, this coordinate data also facilitates routing algorithms with information to route the packets.

For a given node in a SF grid, the row number (row_{n_i}) and column number(col_{n_i}) can be evaluated from the corresponding VCs using the following equations,

$$row_{n_i} = \frac{h_{n_iA_j} + h_{n_iA_k} - h_{A_jA_k}}{2} \tag{5.1}$$

$$col_{n_i} = |\frac{h_{A_jA_k}}{2}| - |\frac{h_{n_iA_k} - h_{n_iA_j}}{2}| \tag{5.2}$$

These equations provide us with the 2D physical coordinate for node a as (4, 3) and node b as (2, 4). From our earlier observations, these values are correct and the nodes lie on these locations in the topology of Figure 5.1. In more complex networks, or with some other anchor placement, it is possible for different nodes to have identical VCs. However, to be able to uniquely identify a node, each node must have an unique set of VCs. This anchor selection plays a huge role in identifying unique coordinates for all the nodes of the SF grid and therefore localization of the nodes. These placement of anchors substantially helps to localize the nodes in the grid [39].

Now, consider the Figure 5.2, a SF of the same length L and breadth B with two anchors located at the same positions as shown in Figure 5.1. However, in this case there are a few nodes that are missing in the grid creating a void. In practice, voids could be of any random shape and there could be more than one void in the grid. Due to this, the VCs of the nodes concerning the anchors are affected. In Figure 5.2, the VCs of the node a' are ($h_{a'}A_j$, $h_{a'}A_k$) = (9, 6). According to the equations 5.1 and 5.2, the equivalent 2D physical coordinate for the node a' evaluates to (x, y) = (4, 5). However, as we can see, this is incorrect and the coordinates conflict with the 2D physical coordinates of the node a''.

Localization is computationally expensive and many algorithms cause overheads. Hence, it is a complex issue to have a grid with void that is completely localized. When the nature of voids becomes irregular, the shortest distance hop count of nodes gets affected. To overcome

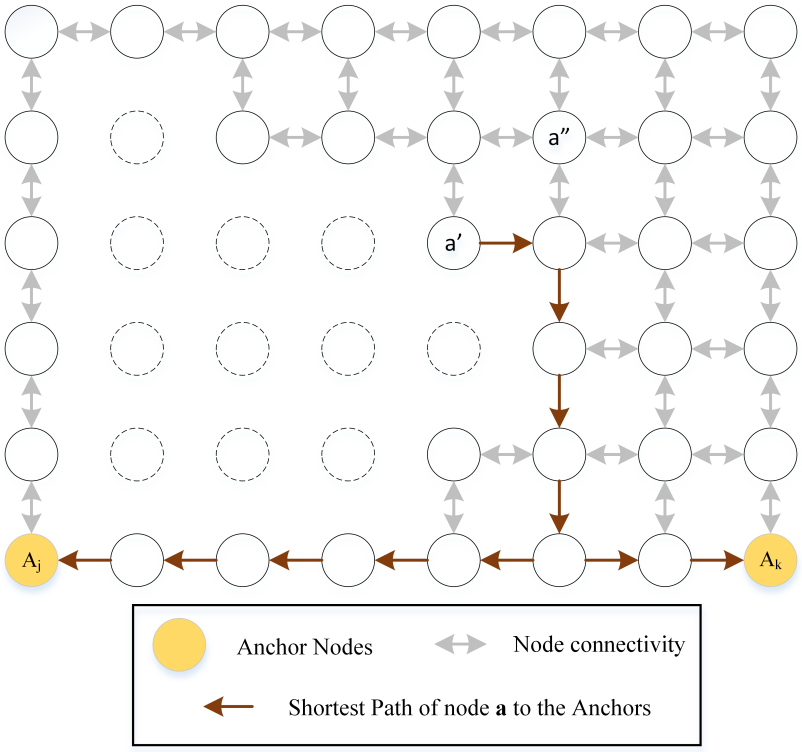


Figure 5.2: Addressing in a network with missing nodes

these limitations, a new localization scheme is proposed in the following sections.

5.2 Adaptive Localization Algorithm

The VCs for a node in a SF grid with void may be different from that in the full grid due to missing nodes. Anchor placement and the voids affect the shortest path between the node and anchor(s). Consider Figure 5.3. Here we have anchors A_j and A_k placed as shown. This grid has the same length, breadth and node a' at the same position as in Figures 5.1 and 5.2. Now, the VCs for node a' would be $(A_j, A_k) = (9, 6)$ due to the missing nodes in the grid. In the case when there are no missing nodes in the grid, the VCs would have been (7, 6). This shows that a message from anchor A_j requires an extra distance of $9 - 7 = 2$ hops to travel to node a'. This additional distance traveled is defined as delta (δ). Any additional distance traveled from the anchors would be a δ addition in the actual VC for the node. Also, node a'' has the same VC as node s' causing an overlap or conflict in achieving unique physical coordinates. As voids are developed in the grid, the algorithm fails to converge leading to

incorrect localization. The VCs for the nodes can fluctuate depending on the size, shape, and location of these voids.

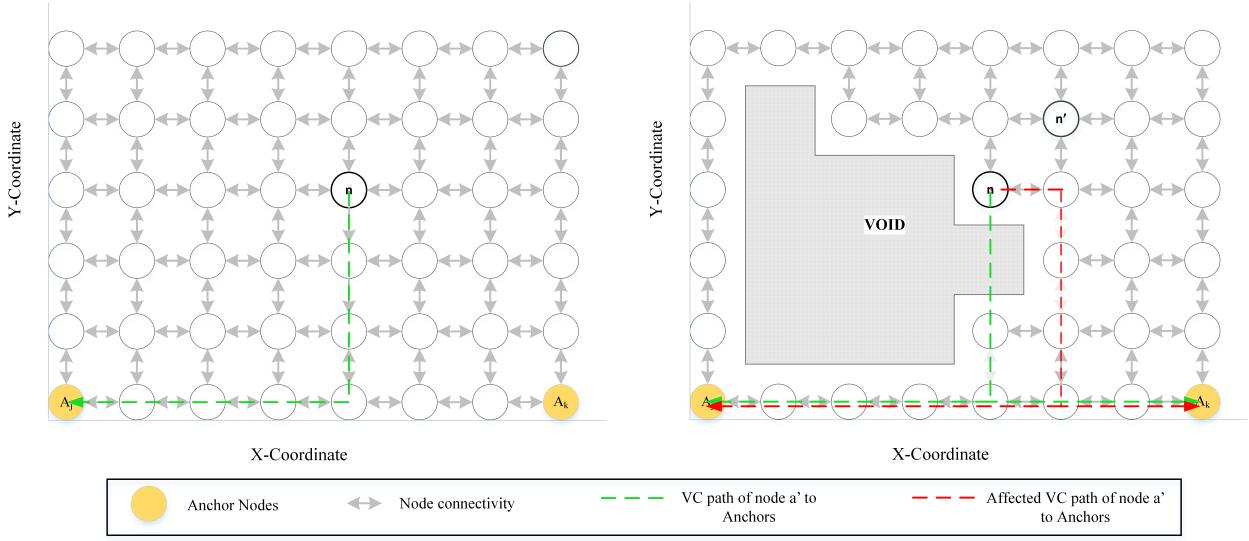


Figure 5.3: Voids affecting VCS path

The following sections explain the delta Minimization and neighbor verification techniques used by the Adaptive Localization algorithm.

5.2.1 Delta Minimization

This section describes in detail the delta (δ) minimization technique used in Adaptive Localization Algorithm. Considering the SF grids from the Figure 5.3. Following equations state the VC computation for the grid nodes. For a grid node n,

$$V_j(h_{nA_j}) = x + y + \delta_j \tag{5.3}$$

$$V_k(h_{nA_k}) = L - x + y + \delta_k \tag{5.4}$$

where,

- V_j and V_k are the hop distances from node n to anchors A_j and A_k respectively. To keep the explanation simple, we use V_j and V_k to denote V_j (h_{nA_j}) and V_k (h_{nA_k}) respectively.

- x, y are the horizontal and vertical distances traveled by the node respectively. ($0 \leq x \leq L$) and ($0 \leq y \leq B$).
- δ_j and δ_k are the additional distances to be traveled by the node to anchors A_j, and A_k respectively. Note that, δ_j, $\delta_k \geq 0$.

Once the VCs for the grid nodes are computed, we have x, y, and δ parameters as the unknowns in the equations 5.3, 5.4. Additionally, there are multiple shortest paths that can be taken to reach the anchors from the base node n. When a node has the shortest path to an anchor that is unaffected by any external parameters, x, and y are computed to be consistent with that of the full grid and all the δ parameters would be '0' as there would be no additional hops to be traveled to reach the anchors. However, in a scenario when a node cannot reach an anchor with the shortest path due to a void, δ has a non-zero value.

Consider Figure 5.3 with a complete grid with length $L = 7$ and breadth $B = 4$ with anchors A_j and A_k where node n has a VCs $(V_j, V_k) = (7, 6)$. Since we are working with a complete grid with no voids, we can guarantee that the VCs are according to the shortest path. We use Dijkstra's algorithm to compute the hop distance. Analyzing with respect to 5.3 and 5.4 we can see that,

- $7 = 4 + 3 + \delta_1$
- $6 = 7 - 4 + 3 + \delta_2$

Solving these equations, we get δ_j, $\delta_k = 0$. This shows that for a SF grid with no voids, the shortest path is determined by x and y units. Now, consider Figure 5.3 grid with same L, B, and anchors. The difference here is that there are a few missing nodes in the grid now creating a void. Due to this void, the VCs of node n are affected. VCs can be deduced to $(V_j, V_k) = (9, 6)$. Here, the VC V_j is affected. In this case, we know that the minimized shortest path values of x and y are 4 and 3 respectively. Analyzing with respect to 5.3 and 5.4 we can see that,

- $9 = 4 + 3 + \delta_j$
- $6 = 7 - 4 + 3 + \delta_k$

Solving these equations, we get $\delta_j = 2$, $\delta_k = 0$. Thus, for node n to travel to anchor A_j, it needs to cover an additional distance of 2 hops. Thus, δ is an additional distance the node has to travel to reach the anchor due to the void created by missing nodes in the SF grid. We devise a technique to minimize the delta value. Thus, the distance δ could be a combination of hops traveled in x plus the distance traveled in y by the node towards the anchor. Hence, for all the δ parameters (δ_j and δ_k), we can resolve them as,

$$\delta = \delta_j + \delta_k \tag{5.5}$$

For a given SF grid with elected anchors and void(s), with the given adjacency matrix, there are several combinations of x, y, and δs that would provide a shortest path. However, only one value in that set would provide the correct shortest path. With the above speculations, we can find all the possible combinations of x, y, and δs given that any of the δs is non-zero. Let's take a look at equations 5.3 and 5.4. In a case when we have the correct shortest path, the δs will be 0 or minimized value of δ. As stated earlier, δ_1, $\delta_2 >= 0$. Rearranging the equations 5.3 and 5.4 and adding constraints for x and y we get,

$$x \geq 0 \tag{5.6}$$

$$y \geq 0 \tag{5.7}$$

$$x + y \leq V_j \tag{5.8}$$

$$-x + y \leq V_k - L \tag{5.9}$$

Here, x and y are the hop distances from the base node and are all positive values. V_j, V_k and L are constants and known. Hence, we can plot these equations as four lines and find out the intersection points around the area of interest. Consider Figure 5.4 that shows all possible scenarios in which we have the area of interest or the solution space. We find the

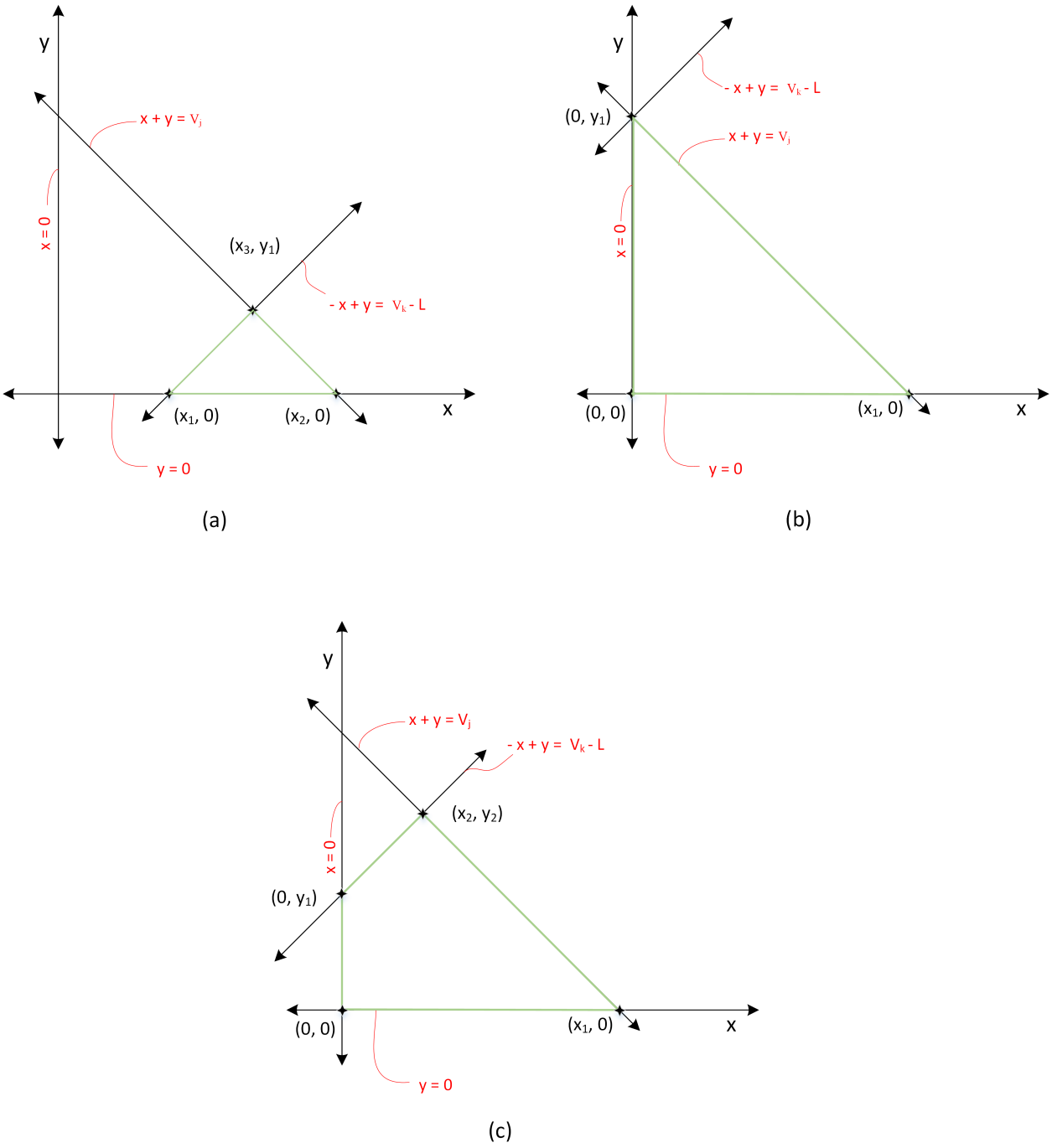


Figure 5.4: Bound calculation (a) Case I (b) Case II (c) Case III

intersection points of these lines given that they abide by equations 5.6, 5.7, 5.8, and 5.9. We have examples of three scenarios where we get a bound for x and y values that help us in minimizing δs.

Hence, from Figure 5.4 (a), we can see that $x_1 \leq x \leq x_2$ and $0 \leq y \leq y_1$. Similarly, in

5.4 (b), $0 \leq x \leq x_1$ and $0 \leq y \leq y_1$. In 5.4 (c), $0 \leq x \leq x_1$ and $0 \leq y \leq y_2$. Once these bounds are computed, using the equations 5.3 and 5.4 we can achieve minimization for all the δs such that $0 \leq \delta_n \leq max(x - coor, y - coor)$. We further narrow down the solution set to the best possible combination of (x, y) that achieves the required minimization.

Once we have the optimized deltas, most of the nodes that have the shortest path to the anchors will be localized with unique (x, y) values. Depending upon the void in the SF grid, we would still have a few nodes that would not be localized due to multiple possible values or identical nodes that share the same subset of values with it. These unassigned nodes may have several possible δ and (x, y) values even after minimization. These conditions would create conflicts in assigning a unique (x, y) coordinate from the derived coordinates.

After the delta minimization, whenever a node is assigned a (x, y) coordinate, the data is filtered to remove the corresponding (x, y) value as a possibility from the data sets of other nodes. This helps us optimize the individual data-sets to find the correct (x, y) coordinate per node gradually. Each time, post-filtering, any node(s) might be optimized to it's unique (x, y) coordinate. The algorithm checks at every iteration if a node elects its own (x, y) through the elimination from filtering and identical node identification. If in an iteration no node is updated, we break the loop. The result is all the nodes that we can be localized using two anchors. Since using two anchors there is a limit to the number of nodes that can be localized, we have adaptive anchor addition to achieve further localization. Here, we update the solution space of node coordinates using adaptive anchor addition. Each of the newly added anchors is from the previously localized nodes.

5.2.2 Neighbor Verification

Once the nodes are localized using delta minimization method, we use the neighbor verification algorithm to rectify any incorrectly determined node coordinates. We compare the neighbors of the node determined by the new localization coordinates vs the neighbors determined by the initial adjacency matrix. The algorithm depicts the neighbor verification

process.

Consider figure 5.5 with a node n as the node of interest. From the connectivity matrix information, we can derive the neighbor IDs for a node. The ***deltaOptimization*** algorithm gives us a subset of possible (x, y) coordinates for node n. E.g. The subset of possible values for node n are $(x', y'), (x", y") = (2, 2), (3, 3)$. Hence, we have two possible positions i.e. node with ID 14 or 20 of which only one given position is correct.

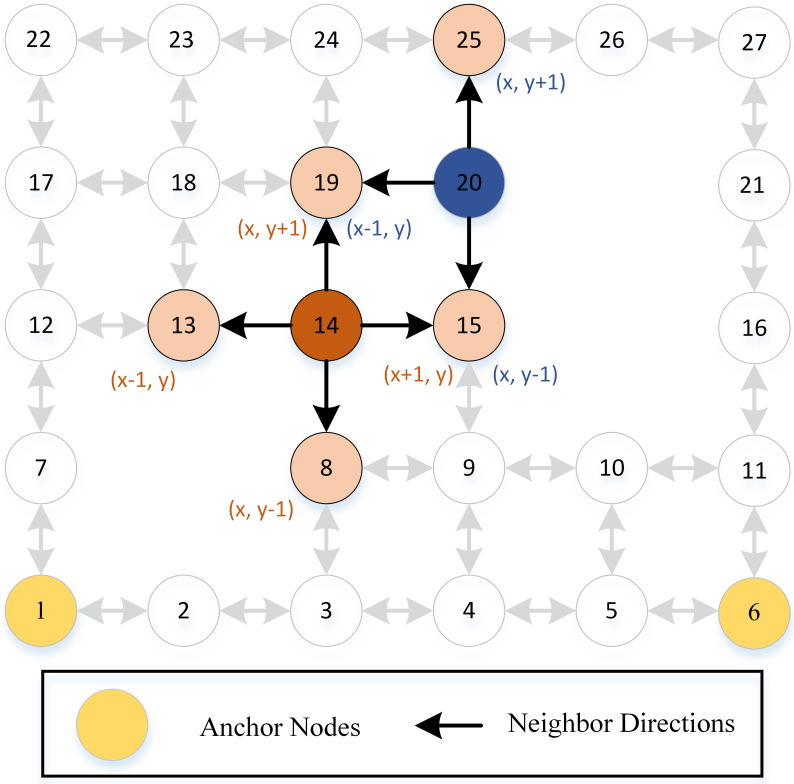


Figure 5.5: Identification of Neighbors for a node

Now, from the subset for node n, by analysis it can be said that, depending on the position of node n it would either have neighbors as nodes 13, 19, 15, and 8 for coordinates $(2, 2)$ i.e. node ID 14 or 19, 25, and 15 for position at $(3, 3)$ i.e. node ID 20. Thus, the neighbors for node 14 or node 20 would have positions $(x - 1, y)$, $(x, y + 1)$, $(x + 1, y)$, and $(x, y - 1)$ respectively where $(x, y) = (x', y') or (x'', y'')$ and number of neighbors depends on the void placement. The coordinates help us compute the neighbor coordinates for the subset. From the adjacency matrix, we can derive the neighbor node IDs for the given node. Comparing

Algorithm 5.2.1 Neighbor Verification

OLD: Find Old Neighbors using Adjacency Matrix

Inputs

$n \leftarrow$ Node ID in the network

$G_{AM} \leftarrow$ Adjacency Matrix of the Network

Outputs

$O\_Neighbors \leftarrow$ *indexes: (-y, -x, +x, +y)*

procedure FIND $O\_Neighbors$ WITH DIRECTIONS
  for ne, con in G_{AM} **do**
    if $0 < con \leq 1$ **then**
      $c = n - ne$
      if $c > 1$ **then**
        $O\_Neighbors[0] = ne$
      else if $c = 1$ **then**
        $O\_Neighbors[1] = ne$
      else if $c = -1$ **then**
        $O\_Neighbors[2] = ne$
      else if $c < -1$ **then**
        $O\_Neighbors[3] = ne$
      end if
    end if
  end for
end procedure

NEW: Find New Neighbors using Localized Coordinates

Inputs

$(x, y) \leftarrow$ Computed node coordinates

$L_N \leftarrow$ Localized grid nodes

Outputs

$N\_Neighbors \leftarrow$ *indexes: (-y, -x, +x, +y)*

procedure FIND $N\_Neighbors$ WITH DIRECTIONS
  $neighbors = (x, y-1), (x-1, y), (x+1, y), (x, y+1)$
  for ne in $neighbors$ **do**
    if ne in L_N **then**
      Append ne to $N\_Neighbors$
    end if
  end for
end procedure

these values would help us eradicate the incorrect subset of nodes and achieve localization.

If any node is localized in this process, filtering is initiated that removes the localized nodes' $(x, y$ coordinate from the subsets of other nodes that haven't been assigned a physical coordinate yet. There is a limitation here that not all the neighbors of the node would be assigned which would result in ambiguity and a node having multiple possible (x, y) values. However, the filtering in collaboration with the neighbor identification process helps localize some unassigned nodes with a unique coordinate and assists the rest of the nodes to optimize the subset of (x, y) values substantially.

5.2.3 Localization using Anchor addition & Coordinate Optimization

This section presents the algorithm for the adaptive anchor placement method for localization. The algorithm attempts to compute the solution space for the node physical coordinates; initially using two fixed anchors and adaptively adding more anchors to progressively improve localization. Algorithm 5.2.3 invokes the Grid Simulator algorithm 4.2.1 to simulate a SF Grid with given dimensions. It uses the adjacency matrix provided by the simulator algorithm to compute VCs. The algorithm initiates by executing the ***computeVCs*** method that computes the VCs for the grid using anchors A with coordinates $(0, 0)$ and $(0, L)$ across the L of grid. Additionally, the weights for all grid nodes are computed using ***computeWeights*** and are recorded. The weight computation for nodes is carried out for the borders of the grid; outer and inner border if a void is present. In the first step, only extreme border nodes with $W_n = 0$ are identified using the information that only border nodes have less than 4 neighbors.

In the first loop of adaptive addition, two anchors A are initialized. Then, the algorithm performs ***deltaOptimization*** to compute localized nodes L_n for that step. Then, these localized nodes are refined by ***refineLocalized*** to find any additional localized nodes and L_n is updated. After this step, ***neighborVerification*** method performs for each L_n a neighbor verification based on G_{AM} and it's existing coordinates. This provides with all localized

Algorithm 5.2.2 Localization by adaptive anchor placement

Simulating a Smart Fabric Grid

Inputs
$L \leftarrow$ Length of Grid
$B \leftarrow$ Breadth of Grid
$P_M \leftarrow$ Percent missing nodes in Grid
$V \leftarrow$ Number of Voids

procedure GRID SIMULATOR
$G_{AM} \leftarrow$ Adjacency Matrix of Simulated Grid
end procedure

Grid Localization: Adaptive approach

Inputs
$G_{AM} \leftarrow$ Adjacency Matrix
$A \leftarrow$ Set of Anchors
$NA \leftarrow$ Set of nodes not be elected as Anchors

Outputs
$S \leftarrow$ Adaptive Steps
$A \leftarrow$ Updated Anchor Set
$L_n \leftarrow$ Localized Nodes
$I_n \leftarrow$ Identical Nodes

Variables
$L_n \leftarrow$ Localized Nodes
$W_n \leftarrow$ Weight of Nodes
$VC_n \leftarrow$ VCs of the grid w.r.t A

procedure ADAPTIVE ANCHOR ADDITION FOR LOCALIZATION
$VC_n =$ ***computeVCs***
$W_n =$ ***computeWeights***

while true **do**
S $\leftarrow S + 1$
if S $= 1$ **then**
$A = [(0, 0), (0, L)]$
else
if $A.length > 2$ **then**
$NA \leftarrow$ ***adaptiveCheck***
end if
if $L_n = G_{AM}.length \;||\; L_n = A + NA \;||\; iter = GAM.length/3$ **then**
break
end if

```
            W_n ← updateWeights
            A ← adaptiveAnchorAdd
            VC_n ← updateVCs
        end if
        for n ← 1 to G_AM.length do
            L_n ← deltaOptimization
        end for
        L_n ← refineLocalized
        L_n, I_n ← neighborVerification
    end while
end procedure
```

nodes in the first step using set A with 2 anchors. This procedure follows a validation check wherein the algorithm stops execution if,

- All grid nodes are localized
- All possible nodes have been tested as anchors
- Number of iterations exceed $1/3^{rd}$ of the grid size

The last condition only keeps a check on the computational complexity and the value can be moderated. If this validation check fails, we proceed to update the weights W_n for all the grid nodes (newly localized) based on elected anchors. Figure 5.6 shows a SF grid with a void and weight values for nodes. As we proceed, the grid nodes are assigned weights based on their proximity to the already elected anchors (2 or more). Hence, as we can see, nodes form a weight spectrum around the elected anchors. Then, the algorithm follows to elect a new anchor ***adaptiveAnchorAdd*** from already localized nodes such that it is not been discarded already NA. Then, the VC VC_n is updated based on new anchors and the algorithm resumes to further computations as described. After we receive updated L_n from 3 anchors, a ***adaptiveCheck*** is performed which essentially checks if adding a new anchor improves the localization. If there is an improvement, the set A and the weights W_n are updated with new anchor addition. Else, the anchor is discarded kept in memory to not be used for future steps. This again follows a validation check and the steps repeat until we find a set of anchors to localize the maximum number of nodes.

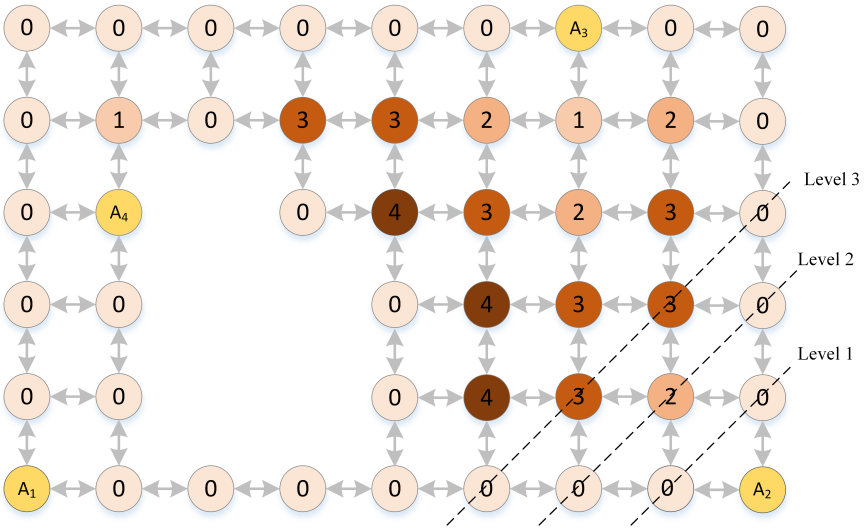


Figure 5.6: Weight computation based on elected Anchors

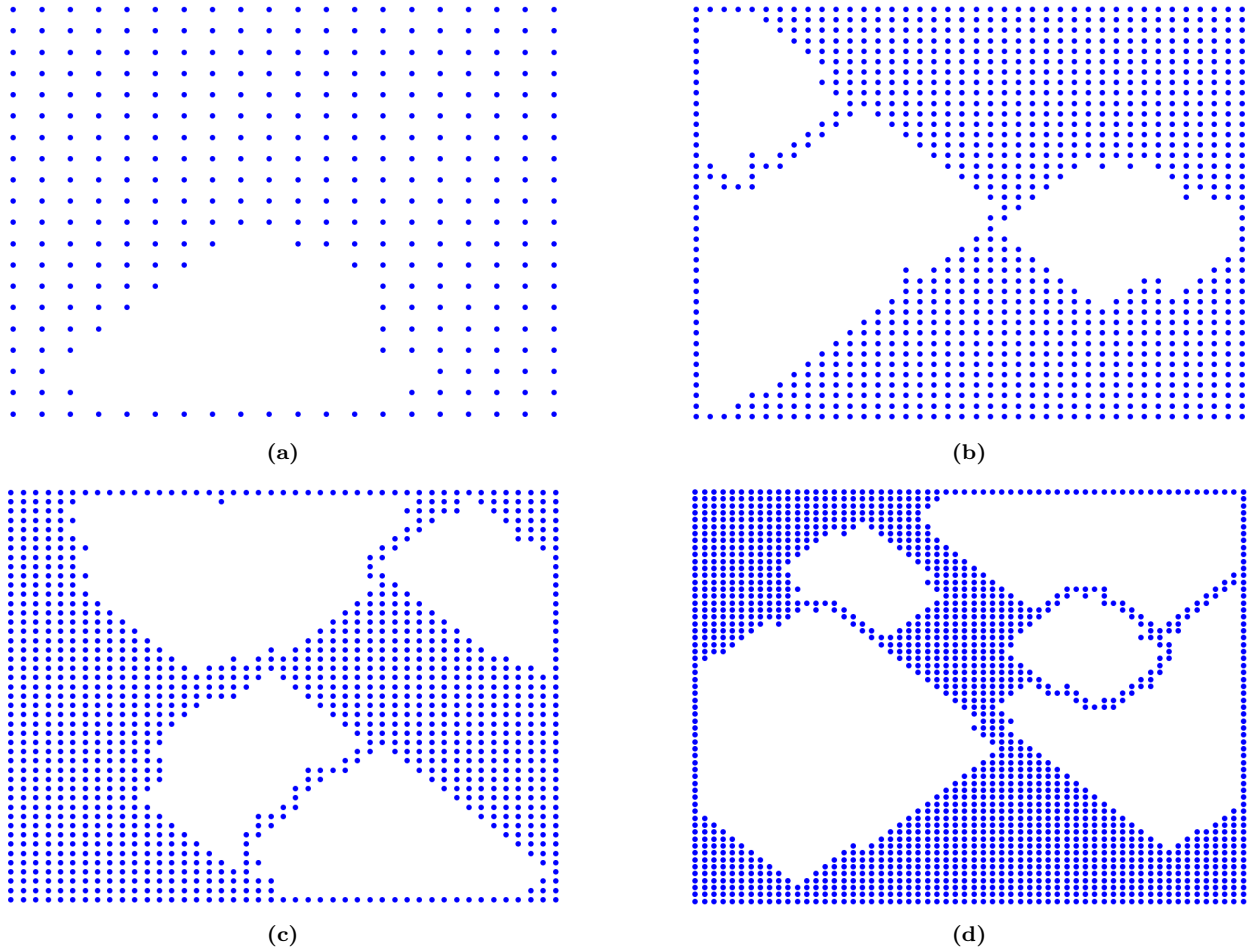
Two of the most important methods described earlier, ***deltaOptimization*** and ***neighborVerification*** will be discussed in detail in the following sections.

5.3 Results & Analysis

This section entails the analysis and result plots for the localization algorithm. As stated earlier, the algorithm adaptively selects anchors that are used to localize nodes. The following results include the plots for analysis, comparison with traditional localization, and localization values for SF grids with varied sizes, percentages of missing nodes, and voids.

Figure 5.7 shows three SF grids with dimensions (LXB) as (a) $19X19$ with 1 void and 10% missing nodes (b) $29X29$ with 3 voids and 30% missing nodes (c) $44X44$ with 4 voids and 40% missing nodes (d) $59X59$ with 5 voids and 60% missing nodes. These are SFs with varied grid sizes, void sizes, and the number of voids. Figure 5.7 (e) plots the number of adaptive anchor placement steps vs. the % localized nodes per adaptive step.

With every adaptive step, we elect a new anchor from one of the localized nodes depending on its weight value. If the anchor addition improves the localization, we retain the anchor or else discard it. Even if the number of anchors elected to achieve efficient localization is higher, as we are choosing new anchors from the nodes that were localized in earlier steps. Thus, we do not require prior knowledge of physical coordinates for any of the newly elected

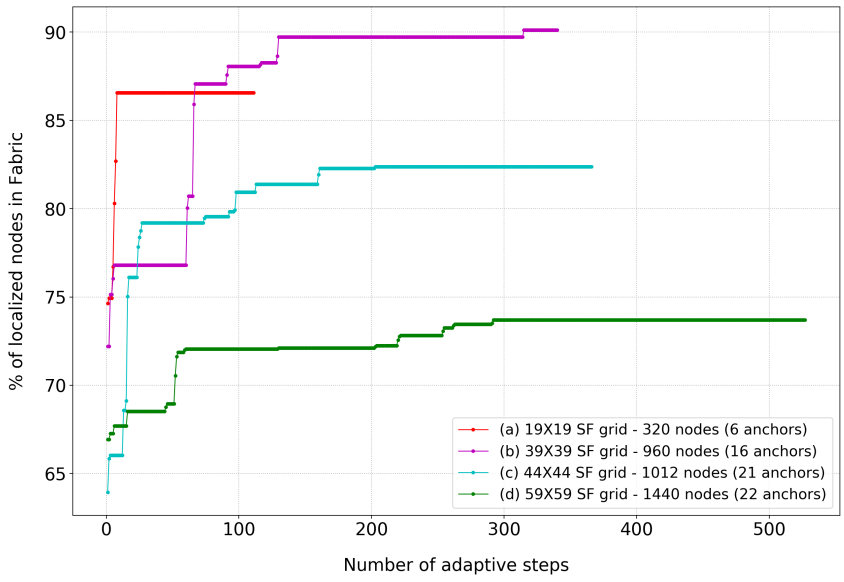


(a)

(b)

(c)

(d)



(e)

Figure 5.7: (a) Smart Fabric Grid with 320 nodes and 1 void, (b) Smart Fabric Grid with 960 nodes and 3 voids, (c) Smart Fabric Grid with 1012 nodes and 4 voids, (d) Smart Fabric Grid with 1440 nodes and 5 voids, (e) Number of adaptive anchor addition steps vs % of localized nodes for (a), (b), (c), (d)

anchors.

As every new anchor has been elected from the localized set of nodes, we have a closed-loop system that takes in feedback and elects an anchor based on earlier stated conditions. It then uses the old anchors in combination with the new ones to achieve maximum localization. In the Figure 5.7 (e), we see that $19X19$ size SF grid achieves 74.63% localization with initial 2 anchors. With the subsequent addition of anchors, the localization progresses reaching 86.57% until we exhaust the number of steps for computation i.e. ≈ 38 additional localized nodes. Similarly, for the SF grid in 5.7 (b) with size $39X39$, there is a localization improvement from 72.21% to 90.12% which is an efficiency of 17.91% i.e. ≈ 172 additional nodes being localized compared to step 1. The $44X44$ SF grid in 5.7 (e) improves localization by 18.44% i.e. ≈ 187 nodes additionally localized. Lastly, in Figure 5.7 (d) we have a $59X59$ grid in which the adaptive anchor addition enhances the localization from $\approx 66\%$ to 74%. Finally, the plot in Figure 5.7 (e) shows the improvements in localization with adaptive anchor addition steps.

Following figures, Figure 5.8, Figure 5.9, Figure 5.10, and Figure 5.11 show the analysis plots for % Localized Nodes and Anchors for SFs grids. Each plot contains a data-set with 60 samples for each data point. Each plot has two subplots - Plot 1: % of Localized Nodes vs. Number of voids in Smart Fabric and Plot 2: Number of elected anchors vs. Number of voids in Smart Fabric. Plot 1 shows the Confidence Interval (C.I.) for the data points as explained in Appendix A. For all the plots we can see that there is a gradual decline in the percentages of localized nodes with an increase in the number of voids. However, it is to be noted that despite the increasing size, the algorithm manages to localize approximately more than or equal to 65% nodes for all the SF grid sizes.

For all varied grid sizes for 10% missing nodes, and up to 5 voids, $\approx 95\%$ nodes are localized. For 20%, we have a localization percentage value of greater than 85%. In fact, we can see that the localization value rises with an increase in grid size. In SF grids with less number of nodes, with increasing percentages of missing nodes, it becomes challenging

to introduce voids as we lose a lot of information which essentially affects the ability of the adaptive localization algorithm. Further, for 30%, 40%, and 50% missing nodes as well we can see that the algorithm localization performance improves. Additionally, in Figure 5.12, we evaluate SF grids with higher percentages of missing nodes. In this scenario, we do see a steep decline in the percentages of localized nodes for a higher percentage of missing nodes (70%) but the algorithm manages to localize up to 50% of the nodes. This shows us the scalability and efficiency of the algorithm.

The second plot in each Figure shows the mean of number of anchors adaptively required to achieve the given localization. We expect the number of elected anchors to increase with number of voids, increase in percentages of missing nodes, and increasing grid size. However, for grids with small sizes ($19X19$), there is not a substantial increase in the number of elected anchors as we achieve the maximum localization possible in the given number of anchors. However, as the grid size increases, we see a gradual rise in the number of elected anchors (e.g. in SF grid with size $39X39$) as expected which helps us in achieving maximum localization.

These plots show that SF grids with voids due to certain percentages of missing nodes can be localized efficiently by using the adaptive anchor election algorithm. Detailed plotting data for all the figures is presented in the Appendix A for all grid sizes. It also lists the Confidence Interval (C.I.) for percentages of localized nodes and the elected anchors to illustrate the sample distribution.

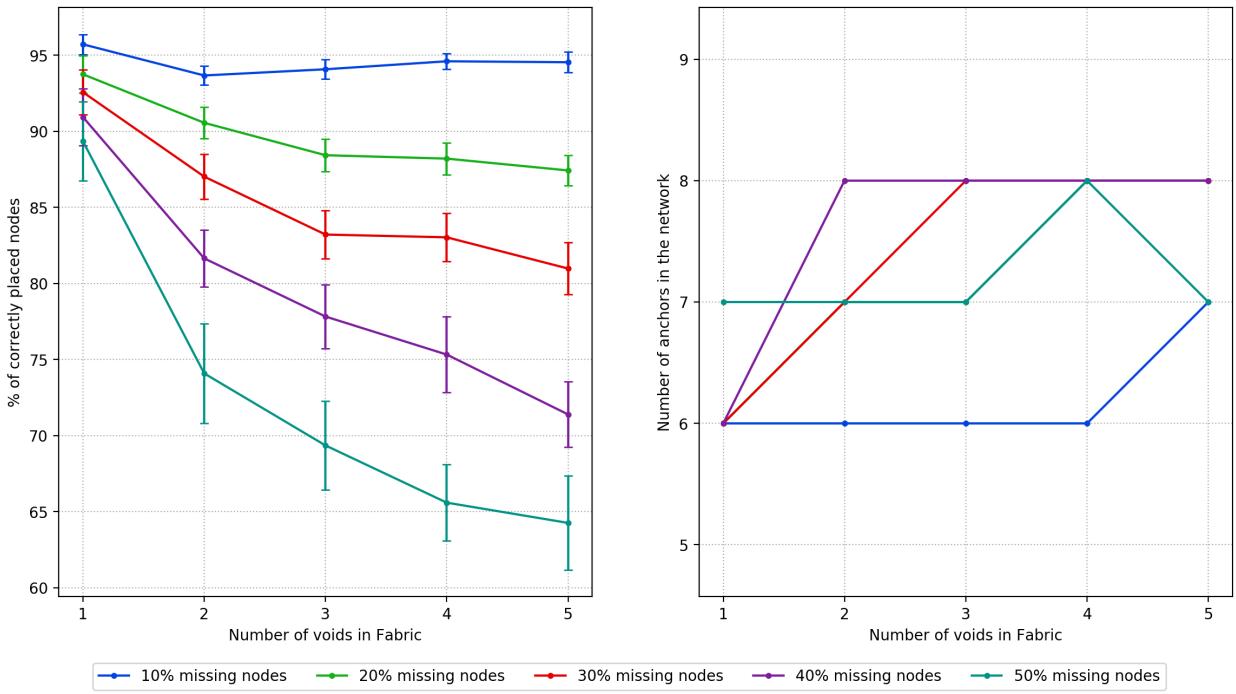


Figure 5.8: Localization and Anchor data for 19X19 SF Grid for 60 samples

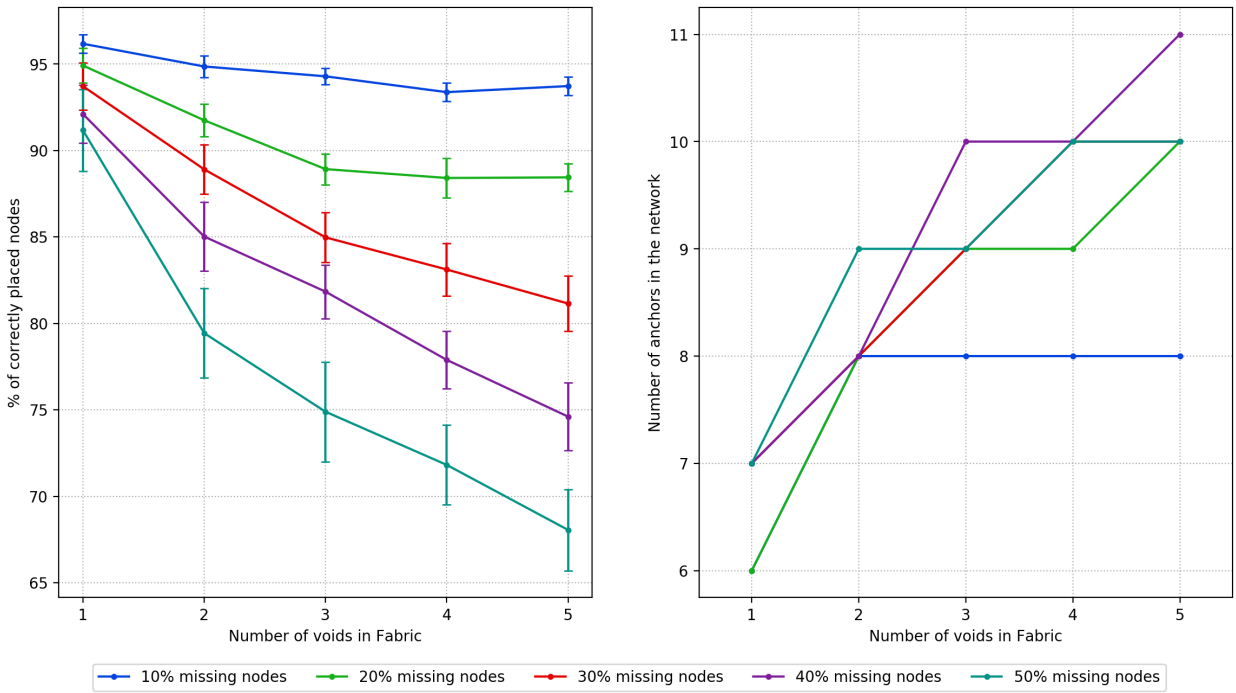


Figure 5.9: Localization and Anchor data for 24X24 SF Grid for 60 samples

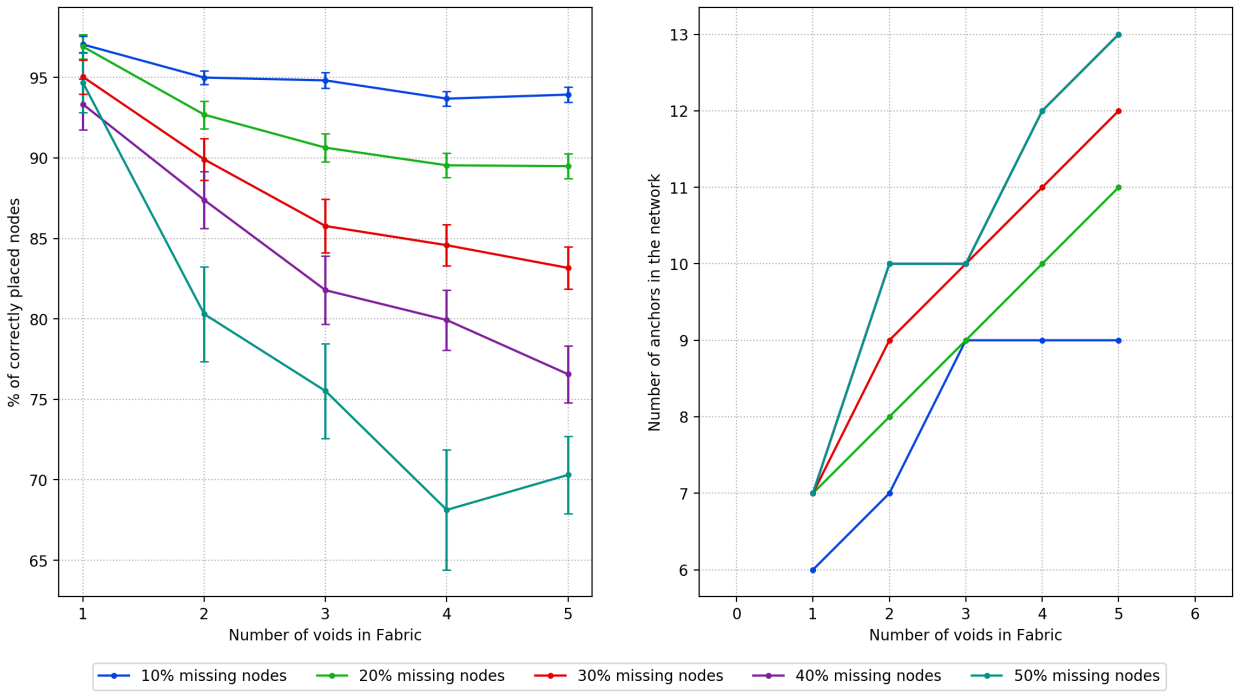


Figure 5.10: Localization and Anchor data for 29X29 SF Grid for 60 samples

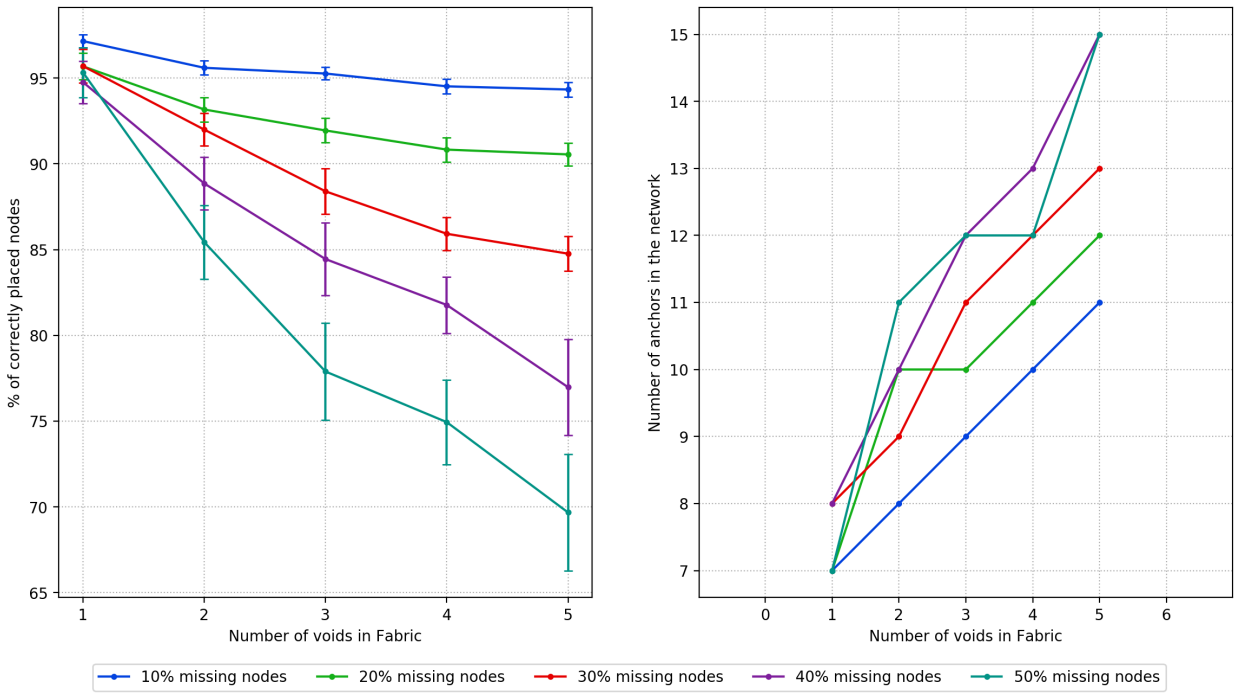


Figure 5.11: Localization and Anchor data for 34X34 SF Grid for 60 samples

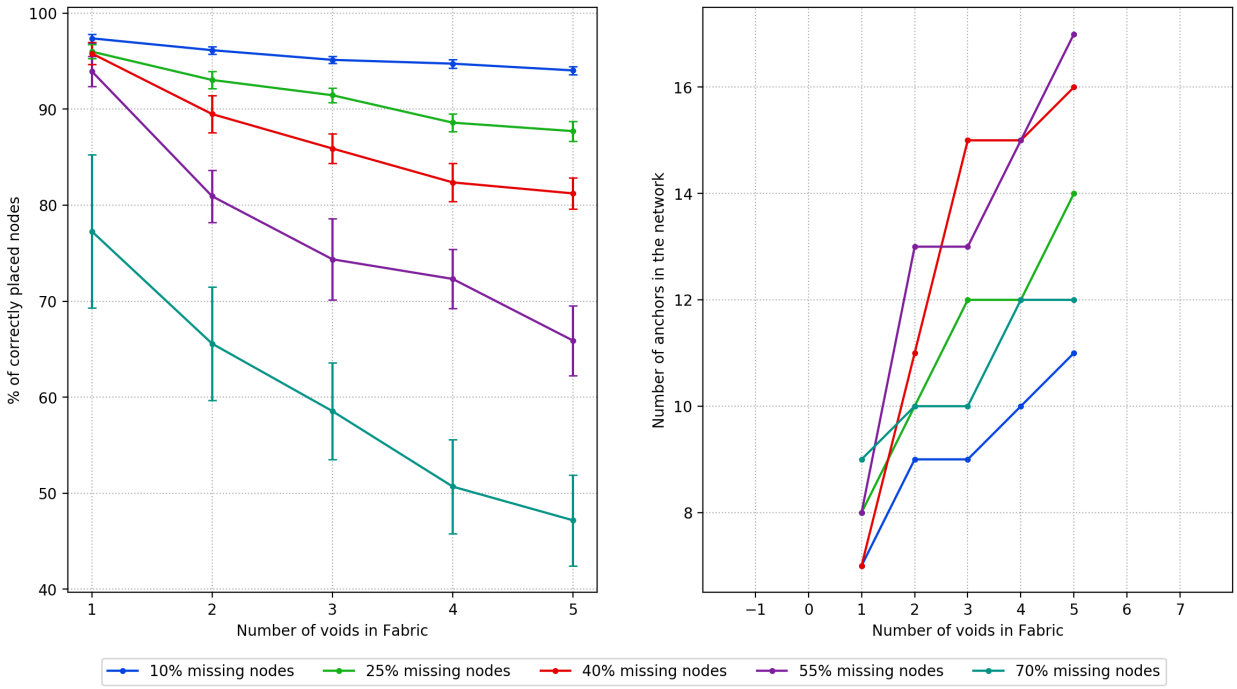


Figure 5.12: Localization and Anchor data for 39X39 SF Grid for 60 samples

CHAPTER 6

SMART FABRICS WITH TRIANGULAR GRIDS

6.1 Introduction

Chapters 4 and 5 address Rectangular grid Fabrics i.e. SFs with node placement in rectangular fashion. However, node placement in SFs could be in any geometric pattern depending on the application requirement. In this section, we address SF grids with node placement in a triangular fashion. We generate the triangular grids with defined shapes and use localization equations defined in the following sections to compute the position coordinates for the SF grid given the adjacency matrices for those patterns.

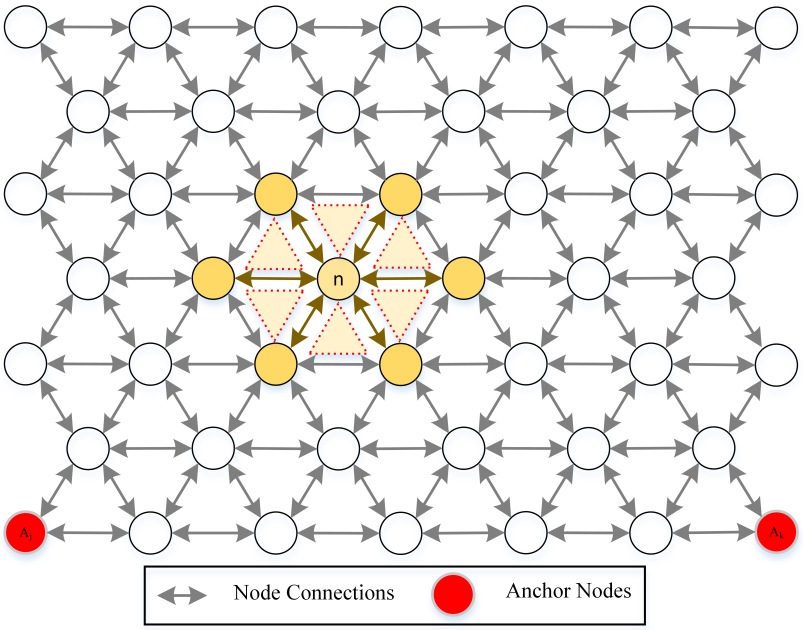


Figure 6.1: Triangular Smart Fabric Grid placement

Figure 6.1 shows a sample SF grid with triangular node placement. Each node except the border nodes has six neighbors connected in a triangular fashion. A sample node n is shown with its neighbors. It can be seen that there is triangular connectivity between the

nodes. All the 6 neighbors of a node are located at a 1-hop distance from the node. Hence, we can say that the nodes are connected in an equilateral-triangle geometric pattern.

Table 6.1: Notations used for Chapter 6

NOTATION	DESCRIPTION
N	Smart Fabric Grid Nodes
n_i	Node n_i in grid $\{ n_i \mid i \in N \}$
A_i	Elected Anchors $\{ A_i \mid i \in (j,k,l) \}$
$h_{n_i A_j}$	Minimum hop distance between node n_i and Anchor A_j
$h_{A_j A_k}$	Minimum hop distance between anchors A_j and A_k
D_l	Level of Region 2 nodes w.r.t Equilateral Region border $\{ D_l \mid l \in \mathbf{Z} : l \neq 0 \}$
Z_l	Level of Region 3 nodes w.r.t Equilateral Region border $\{ Z_l \mid l \in \mathbf{Z} : l > 0 \}$
θ_f	Angle factor for Smart Fabric $\{ \theta_f \mid \theta_f \in \mathrm{A} : \mathrm{A} = [1, -1] \}$

6.2 Triangular Smart Fabric Grids

SF grids with sensor nodes placed in the triangular pattern can be divided into 4 sections as defined by [39]. According to Shah, the anchors in the grid must not be placed along the zig-zag border to avoid identical VCs as shown in 6.1. Figure 6.2 shows the various regions for triangular grids as defined by Shah [39]. The regions 1 and 2 define a region where each node has a unique VC and regions 3 and 4 have nodes with identical VC due to their hop-distances from the anchors. These regions are determined based on the nodes' distances from the anchors. In addition to the regions, we introduce level parameters which help in localization for SF grids for the shapes mentioned in the following sections. Table 6.1 describes the notations used in this chapter.

We introduce a parameter, D_l which determines the levels of Region 2 nodes w.r.t to the Equilateral (Region 1) border. Additionally, D_l for a node can be positive or negative depending on the distance of the node from the corresponding anchors. In the scenario in Figure 6.2, nodes n_2, n_3, n_4, n_5 have level determined by D_l as, $+1$, $+2$, -1, -3.

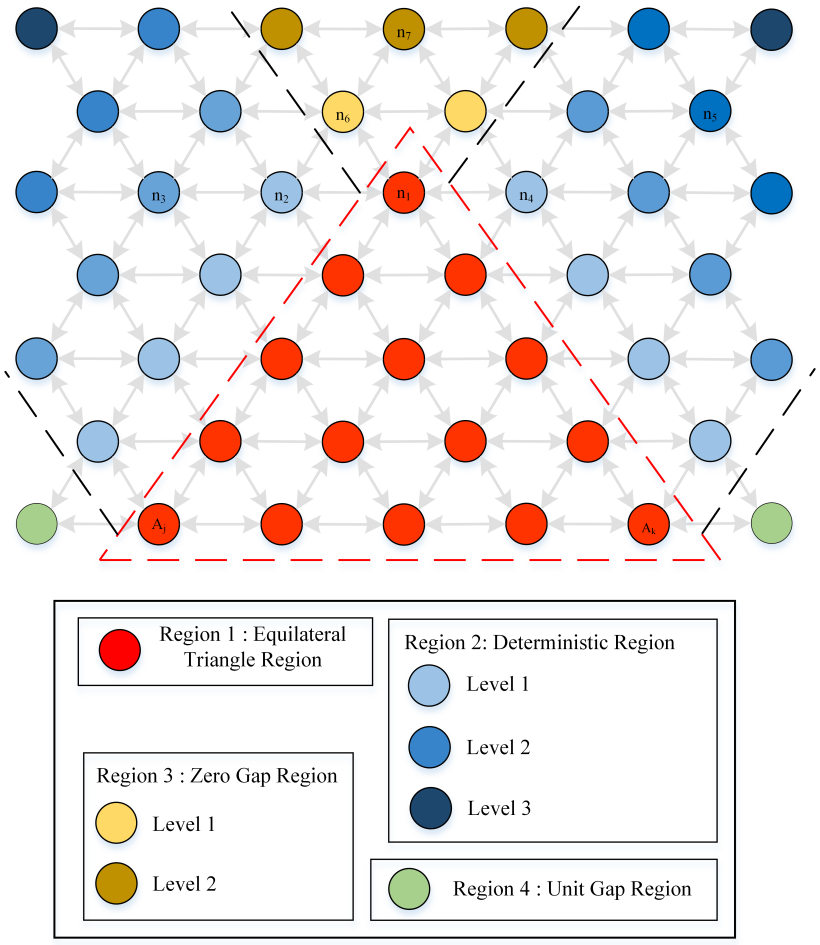


Figure 6.2: Triangular Smart Fabric Grid sectioning

In the following section, we define the localization equations for triangular SF grids of specified shapes given the adjacency matrix of the grid. Here, we define additional parameters to localize the nodes in a 2D format.

6.3 Localization for Triangular Smart Fabric Grids

SFs with varied shapes can be useful in applications wherein the area of interest is the respective shape. Depending on the coverage area, localization techniques stated below can be used for reconstructing the grid and computing the position of the sensor nodes. As discussed in Section 6.2, the sensor nodes have restrictions depending on the region they're located in to achieve unique VCs. Varied shapes with triangular grid node placement have been discussed. For these grids, given a certain number of elected anchors, all the grid nodes

in the grid can be localized. The localization coordinates in these grids are depicted as ***row*** and ***col*** coordinates and they are analogous to $\boldsymbol{y}$ and $\boldsymbol{x}$ coordinates in a 2D Cartesian system respectively.

6.3.1 Polygon shaped Smart Fabric Grids

In this section, we will discuss SF Grids with triangular, trapezoidal and Rectangular shapes. Given an adjacency matrix for a SF Grid with any polygon as shapes as described in the following sections with triangular node placement and specified number of anchors with designated positions, we can localize the sensor nodes with the following defined equations. We will be discussing - equilateral triangle, trapezoidal, and rectangular of SF Grids.

6.3.1.1 Equilateral Triangle & Trapezoid Shaped SF

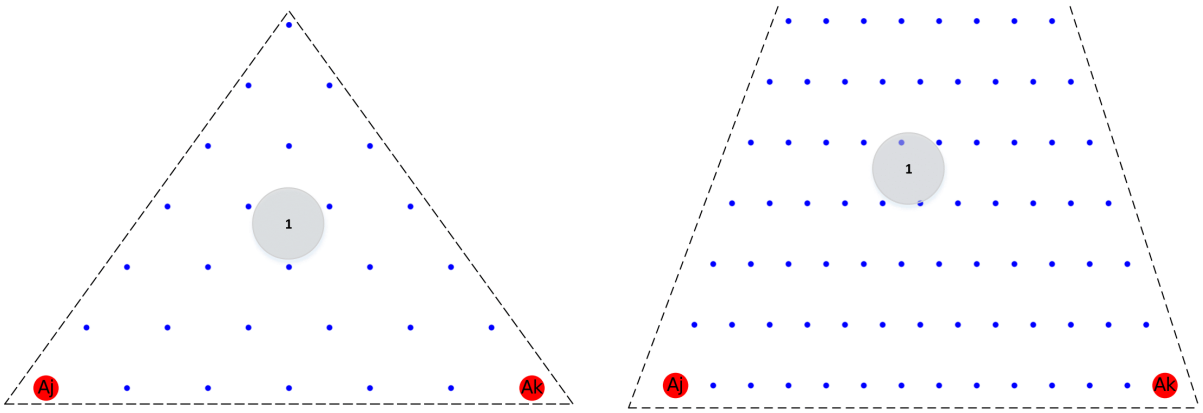


Figure 6.3: (a) Equilateral Triangle SF Grid (6 units) and (b) Trapezoidal SF Grid (13 X 7 X 6 units)

Here, we discuss the localization technique for Equilateral and Trapezoidal shaped Smart Fabric Grids. Figure 6.3 shows two Smart Fabric Grids; (a) Equilateral triangle and (b) Trapezoid with triangular node placement. As per the explanation in Section 6.2, the node VCs highly depend on the region that it is located in and these VCs of the nodes are used for the localization. Hence, the localization technique depends on the region in which the node is located. In the SF grids shown in Figure 6.3, all the nodes of the grid lie in **Region**

1 - Equilateral Triangle region even in case of the Smart Fabric in 6.3 (b). Hence, the same localization equations can be applied to all the nodes in the region 1.

Theorem I: Given a Smart Fabric with triangular node placement, if the shape of the fabric is an **Equilateral Triangle**, 2D localized coordinates for grid nodes can be computed using anchors A_j and A_k placed along any one of the triangle sides at the corners by using the equations 6.1 and 6.2.

Theorem II: Given a Smart Fabric with triangular node placement, if the shape of the fabric is a **Trapezoid** such that, the length of the angular side of the Trapezoid is less than or equal to the largest side of the Trapezoid, 2D localized coordinates for grid nodes can be computed using anchors A_j and A_k placed along the largest parallel side by using the equations 6.1 and 6.2.

$$row_{n_i} = \left(h_{n_iA_j} + h_{n_iA_k} - h_{A_jA_k}\right) \times \frac{\sqrt{3}}{2} \tag{6.1}$$

$$col_{n_i} = \left(\frac{h_{A_jA_k}}{2}\right) - \left(\frac{h_{n_iA_k} - h_{n_iA_j}}{2}\right) \tag{6.2}$$

Note that the physical coordinate as per 2D Cartesian system is $(x, y) = (col_{n_i}, row_{n_i})$. Hence, using the equations 6.1 and 6.2, we can localize all the nodes for the SF grids in Figures 6.3.

6.3.1.2 Rectangle Shaped SF

In this section, we discuss localization techniques for rectangular shaped SFs in detail to prove how this shape provides an efficient way to localize grid nodes using minimal anchors with pre-defined placement. Figure 6.4 shows the Rectangular shaped SF grid.

For a rectangular shaped fabric, as the Figure shows, we can divide the grid nodes in two regions viz. Region 1: Equilateral triangle region and Region 2: Deterministic region. In this placement, all the SF grid nodes have a unique combination of VCs which can be used

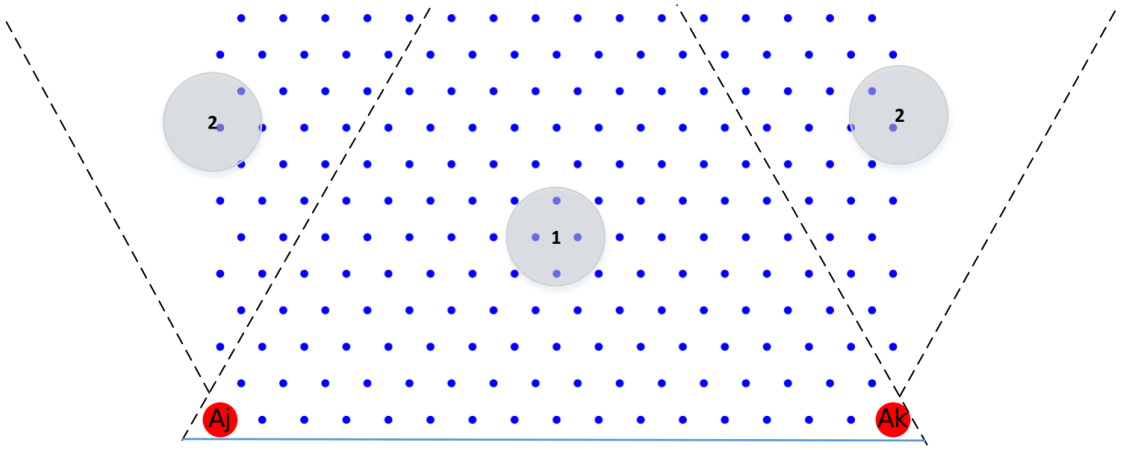


Figure 6.4: Rectangular SF Grid (16 X 10 units)

in combination with an additional parameter D_l; the level of a node in the Deterministic Region w.r.t to Equilateral Region border (as explained in figure 6.2) for localization.

Theorem III: Given a Smart Fabric with triangular node placement, if the shape of the fabric is a **Rectangle**, 2D localized coordinates for grid nodes can be computed using anchors A_j and A_k placed along the Length of the fabric by using the following equations.

Equilateral Region Nodes

All the nodes for the Equilateral Region of a Rectangular shaped SF can be localized using the same equations 6.1 and 6.2 stated for the Section 6.3.1.1.

Deterministic Region Nodes

$$row_{n_i} = \left(h_{n_iA_j} + h_{n_iA_k} - h_{A_jA_k} - |D_l|\right) \times \frac{\sqrt{3}}{2} \tag{6.3}$$

$$col_{n_i} = \left(\frac{h_{A_jA_k}}{2}\right) - \left(\frac{h_{n_iA_k} - h_{n_iA_j}}{2}\right) - \left(\frac{D_l}{2}\right) \tag{6.4}$$

Here, the level of the node in the Deterministic Region is dependent on the hop distance of the node from the anchors A_j and A_k, computed using the following equations 6.5 and 6.6.

if $\left(h_{n_iA_j} < h_{n_iA_k}\right)$

$$D_l = 1 * \left(h_{n_iA_k} - h_{A_jA_k}\right) \tag{6.5}$$

elif $\left(h_{n_iA_k} < h_{n_iA_j}\right)$ **:**

$$D_l = -1 * \left(h_{n_iA_j} - h_{A_jA_k}\right) \tag{6.6}$$

Note that none of the nodes of the deterministic region have the same VC towards both anchors. i.e. $h_{n_iA_j} \neq h_{n_iA_k}$ for all the nodes in the Deterministic Region. Thus, levels of the nodes are determined as shown in Figure 6.2.

6.3.2 Angular Strip shaped Smart Fabric Grids

In addition to the SF grids discussed in 6.3.1.1, we have derived localization techniques for SF grids in the form of strips. In this section, we will be working with "angular" strips as shown in Figure 6.5.

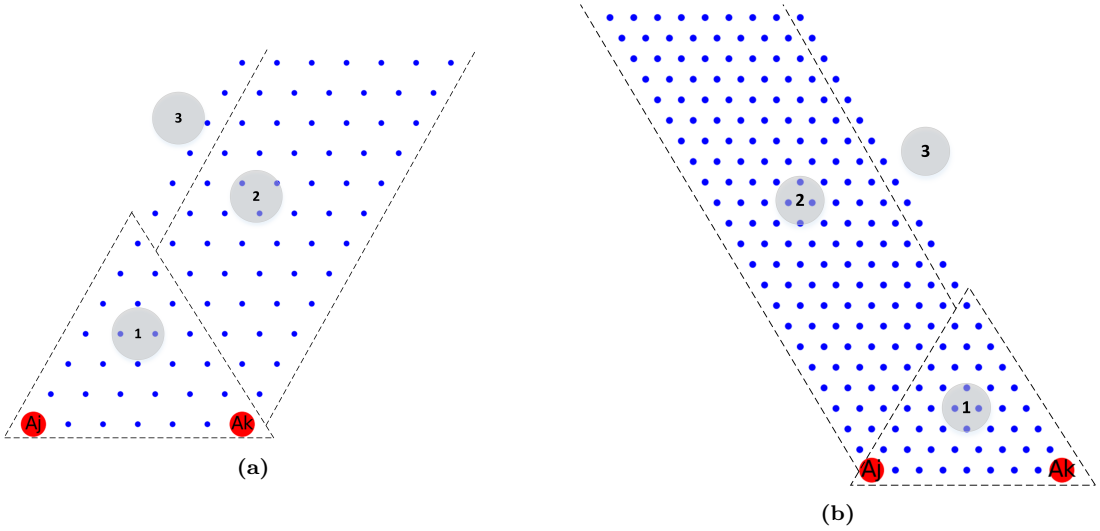


Figure 6.5: (a) Acute-angle Strip of Smart Fabric (12 X 6 units) and (b) Obtuse-angle strip of Smart Fabric (22 X 8 units)

In Figure 6.5 (a) we have a strip with acute angle orientation w.r.t the horizontal base with a width of 6 units and length of 12 units and (b) has an obtuse angle oriented strip with 22 unit length and 8 unit width. The grid nodes lie in three different regions in this

case. The nodes in Region 3: Zero-Gap Region all have the same hop distance from both the anchors A_j and A_k. Hence, we propose an additional parameter 6.9 to compute the level of nodes in the Zero-Gap region. Additionally, the angled orientation of the strip accounts for the angle factor θ_f of the SF strip. If the strip has acute orientation, $\theta_f = (-)1$ else, $\theta_f = (+)1$.

Theorem IV: Given a Smart Fabric with triangular node placement, if the shape of the fabric is a strip with Length $\geq$ Width and **Acute (60°)** or **Obtuse (120°)** angle orientation, 2D localized coordinates for grid nodes can be computed using anchors A_j and A_k placed along the width and at the base of the fabric by using the following equations.

Equilateral Region Nodes

All the nodes for the Equilateral Region of a Strip shaped SF can be localized using the same equations 6.1 and 6.2 stated for the Section 6.3.1.1.

Deterministic Region Nodes

All the nodes for the Deterministic Region shaped SF can be localized using the same equations 6.3 and 6.4 stated for the Section 6.3.1.2.

Zero-Gap Region Nodes

$$row_{n_i} = \left(h_{n_iA_j} + h_{n_iA_k} - h_{A_jA_k} - Z_l\right) \times \frac{\sqrt{3}}{2} \tag{6.7}$$

$$col_{n_i} = \left(\frac{h_{A_jA_k}}{2}\right) - \theta_f \times \left(\frac{Z_l}{2}\right) \tag{6.8}$$

Here, the level of the node in the Zero-Gap Region is dependent on the hop distance of the node from the anchors A_j or A_k and distance between the anchors themselves i.e. $h_{A_jA_k}$. The level Z_l is computed using the following equation 6.9.

$$Z_l = \left(h_{n_iA_j} \vee h_{n_iA_k}\right) - h_{A_jA_k} \tag{6.9}$$

and,

$$\theta_f = (+)1 \vee (-1) \tag{6.10}$$

Thus, using two anchors as stated, we can localize all the grid nodes in a strip with infinite "length" given that the anchors are placed at the base of the strip and along it's width.

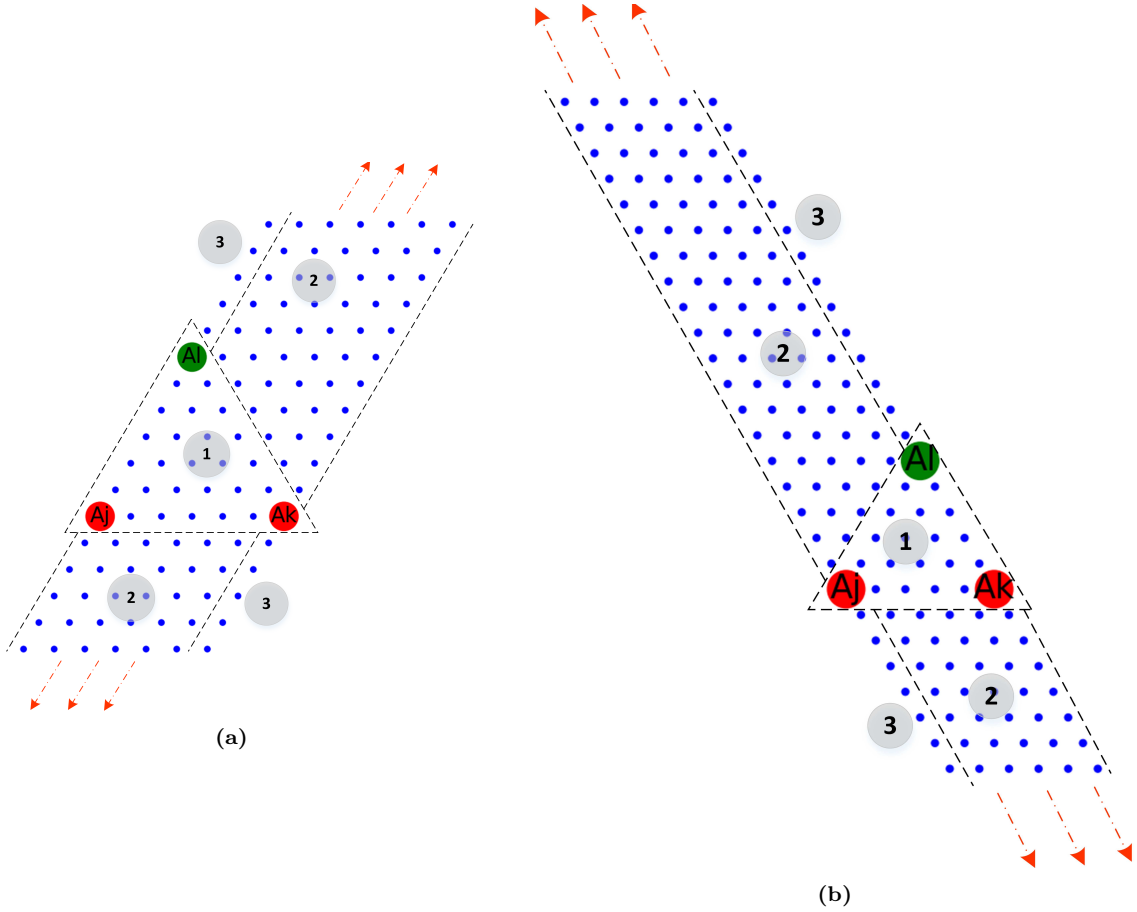


Figure 6.6: (a) Infinite Acute-angle Strip of Smart Fabric (16 X 6 units) and (b) Infinite Obtuse-angle strip of Smart Fabric (26 X 5 units)

In addition to placing anchors at the base of the strip, we propose additional placement to achieve efficient localization. Figure 6.6 shows similar pattern SF strips as Figure 6.5. Here, since the anchors are placed along the width of the strip and NOT at the base, we have a mirroring condition. We split the grid nodes in Regions 1, 2, and 3 as shown and perform localization using the following equations.

To proceed with the localization, we compute a third anchor using the anchors A_j and A_k. The third anchor A_l is computed as,

$$A_l = \{\, n_i \mid n_i \in \mathrm{N} : h_{n_iAj} = h_{n_iAk} = h_{A_jAk} \,\} \tag{6.11}$$

Now, given the structure of the SF in discussion from Figures 6.6, the equation 6.11 gives us two eligible nodes as anchor A_l due to the mirroring condition. We can choose any one of these as the Anchor A_l. Now, we use the information of anchor A_l to localize all the nodes on the strip.

Theorem V: Given a Smart Fabric with triangular node placement, if the shape of the fabric is a strip with Length $\geq$ Width and **Acute (60°)** or **Obtuse (120°)** angle orientation, 2D localized coordinates for grid nodes can be computed using anchors A_j and A_k placed anywhere along the Width of the strip and an additional chosen anchor along the Length by using the following equations.

Equilateral Region Nodes

ROW:

if $\left(h_{n_iA_l} \leq h_{A_jA_k}\right)$ **:**

$$row_{n_i} = \left(h_{n_iA_j} + h_{n_iA_k} - h_{A_jA_k}\right) \times \frac{\sqrt{3}}{2} \tag{6.12}$$

else:

$$row_{n_i} = -\left(\left(h_{n_iA_j} + h_{n_iA_k} - h_{A_jA_k}\right) \times \frac{\sqrt{3}}{2}\right) \tag{6.13}$$

COL:

$$col_{n_i} = \left(\frac{h_{A_jA_k}}{2}\right) - \left(\frac{h_{n_iA_k} - h_{n_iA_j}}{2}\right) \tag{6.14}$$

Deterministic Region Nodes

ROW:

if $\left(h_{n_iA_l} \leq max\left(h_{n_iA_j}, h_{n_iA_k}\right)\right)$ **:**

$$row_{n_i} = \left(h_{n_iA_j} + h_{n_iA_k} - h_{A_jA_k} - |D_l|\right) \times \frac{\sqrt{3}}{2} \tag{6.15}$$

else:

$$row_{n_i} = -\left(\left(h_{n_iA_j} + h_{n_iA_k} - h_{A_jA_k} - |D_l|\right) \times \frac{\sqrt{3}}{2}\right) \tag{6.16}$$

COL:

$$col_{n_i} = \left(\frac{h_{A_jA_k}}{2}\right) - \left(\frac{h_{n_iA_k} - h_{n_iA_j}}{2}\right) - \left(\frac{D_l}{2}\right) \tag{6.17}$$

Zero-Gap Region Nodes

ROW:

if $\left(h_{n_iA_l} < \left(h_{n_iA_j} \vee h_{n_iA_k}\right)\right)$ **:**

$$row_{n_i} = \left(h_{n_iA_j} + h_{n_iA_k} - h_{A_jA_k} - Z_l\right) \times \frac{\sqrt{3}}{2} \tag{6.18}$$

else:

$$row_{n_i} = -\left(\left(h_{n_iA_j} + h_{n_iA_k} - h_{A_jA_k} - Z_l\right) \times \frac{\sqrt{3}}{2}\right) \tag{6.19}$$

COL:

$$col_{n_i} = \left(\frac{h_{A_jA_k}}{2}\right) - \theta_f \times \left(\frac{Z_l}{2}\right) \tag{6.20}$$

Thus, using the above stated conditions for anchor placement and equations, localization can be achieved for all the grid nodes for an infinitely long angular strip.

CHAPTER 7

PARALLELIZATION

7.1 Introduction

In today's world where complex algorithms, heavy computations, and time efficiency are considered to be of great significance, computing systems have evolved to be capable of handling heavy-duty tasks methodically and systematically. Before 2005, most of the computing systems were serial processes until Intel introduced their first Dual-core processor [40]. This idea served the purpose of multiple single-core processors deployed on a single system working together to serve serial processes. Modern computers perform numerous tasks with each core only running through a single process at a time. It requires the processor to constantly switch between the processing threads or instruction streams called concurrency. This results in the wastage of processor cycles during the switching leading to the low efficiency of the computing system.

To utilize the core capacity of these systems to the fullest, multiprocessing or parallel programming approach is introduced. The core idea behind both these approaches is to have tasks of similar type running in parallel across multiple cores. In a traditional serialized approach, all the computing instructions are lined up and interface with the processor using a single thread. Hence, all the tasks are queued and need to wait for the previous task to finish. Parallel programming uses the data from the user to divide the main task into the required number of sub-tasks that run on separate cores and are re-assembled into a queue or any other similar data structure once completed. This provides flexibility and efficiency to the users. In this chapter, we discuss a few ways in which parallel programming can be implemented based on the application requirement.

7.2 Parallel Programming Approaches

Parallel programming can be achieved in many ways by exploiting the cores on a single physical machine or across a High-Performance Computing (HPC) cluster with multiple nodes; each with its processors. In this thesis, we successfully implemented and tested the two popular parallelization modules in python viz. **multiprocessing** and **mpi4py**.

7.2.1 Multiprocessing

The python **multiprocessing** module is one of the most popular packages used to implement parallelization on a single multi-core system. It provides spawning of sub-processes using the API instead of using traditional "threading". This allows the user to leverage the multiple cores on a given system. The multiprocessing module can be roughly utilized using either the **Pool** or the **Process** class. Mane [41] in his article gives a brief overview and ideal implementation scenarios for both classes.

7.2.1.1 POOL for multiprocessing

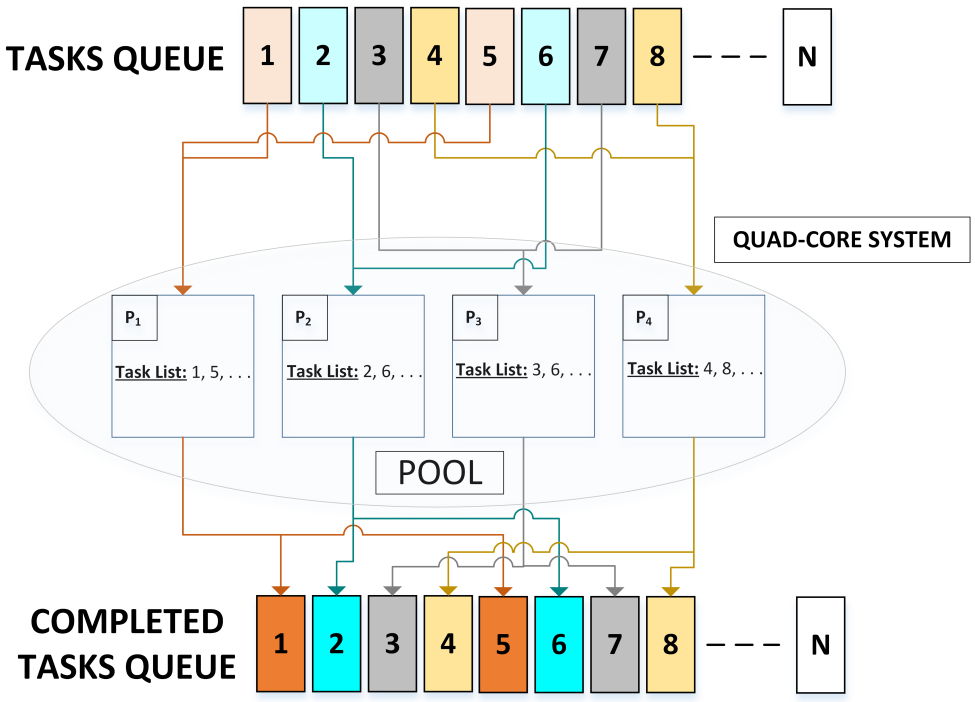


Figure 7.1: Pool-multiprocessing Illustration

Although the function of the multiprocessing module as a whole stays the same (spawning multiple processes) for Pool and Process, the Pool class works differently than the Process class and they both have different use cases. In the Pool class, the number of sub-processes spawned depends on user input and it is ideally equal to the number of CPU cores in the system. Figure 7.1 shows a task queue with N tasks and a Quad-core system (i.e. 4 processors). Here, we invoke the maximum number of processes equal to the CPU count in a pool and the tasks from the task queue are distributed/assigned to these processes in a First In First Out (FIFO) manner. Each core is running a process and hence we achieve parallelization. However, note that each sub-process is performing multiple tasks and they are lined up in a serialized fashion.

```
import multiprocessing as mp
import numpy as np

with mp.Pool(processes=mp.cpu_count()) as task_pool:

    process_output = [task_pool.apply_async(adaptive_localization, args=
                      (i)) for i in range(samples)]

    result = np.array([p.get() for p in process_output])
```

Listing 7.1: Pool Example

The Pool multiprocessing module is efficient in scenarios where a large number of tasks need to execute in parallel. A Pool with a number of processes as many as CPU cores can be created and a list of the tasks can be passed to the processes. Pool collects the return values from the processes in the form of a list and passes it to the parent process. The code snippet in Listing 7.1 shows an implementation of the *adaptive_localization* method discussed in Chapter 5 using the Pool class of multiprocessing.

7.2.1.2 PROCESS for multiprocessing

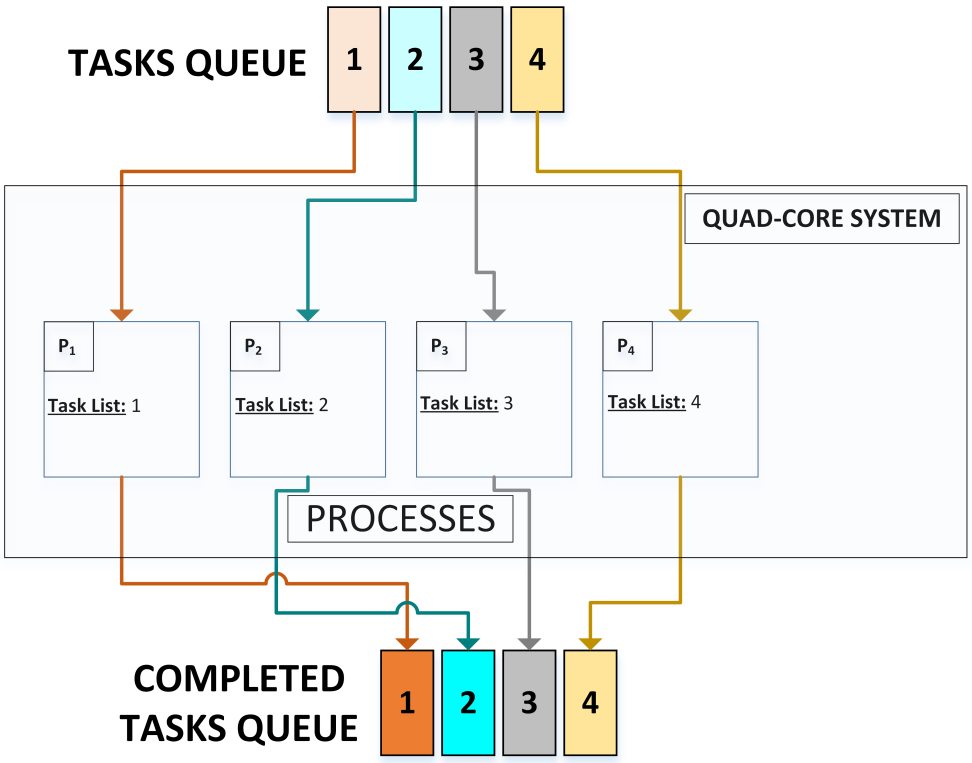


Figure 7.2: Process-multiprocessing Illustration

The Process module works in a slightly different manner such that it spawns a separate process for each task in the task queue. Figure 7.2 shows a task queue with 4 tasks for simplicity. The Process class creates a separate process for each task in the queue and hence we see 4 individual processes P_1, P_2, P_3, and P_4 working on one task each in the system.

```
import multiprocessing as mp
import numpy as np

q = mp.Queue()

processes = [mp.Process(target=adaptive_localization, args=(i)) for i in
            range(samples)]
```

```
for p in processes: p.start()
for p in processes: p.join()


result = array([q.get() for p in processes if p is not None])
```

Listing 7.2: Process Example

The Process class is efficient if you have a small number of tasks to execute in parallel as setting up a Pool has an additional overhead. However, if the number of tasks is large in number, spawning a process for each task will result in a memory error. Additionally, in Pool, the allocated processes per core execute serially. Hence, in case of a long IO operation, it has a wait time until the IO operation is completed leading to an increase in execution time. However, the Process class suspends the process of executing IO operations once it has generated the output and schedules another process.

7.2.2 Message Parsing Interface (MPI)

As mentioned earlier, processes can be executed that run across cores from separate machines to achieve parallelization for scalable applications. "High-Performance Computing" (HPC) is the application of "supercomputers" or a cluster of powerful computers to solve computationally heavy problems that are either too large for standard computers or would take too long. Also, it is difficult to find a single powerful machine with high RAM to run a single application. HPC is also becoming an affordable resource to the research individuals in the scientific community due to the availability of quality open-source software and commodity hardware. This analogy gave birth to Beowulf class clusters and cluster of workstations [42]. A cluster of computer workstations can be viewed as a set of computers connected through fast local area networks (LANs), with each node (computer used as a server) running its instance of an operating system that work together constituting a single powerful system [43].

In our research, we exploit the **Keck Cluster** provided by the College of Engineering for parallelization. Unlike the multiprocessing module, the **mpi4py** module is a message passing

library standard based on the consensus of the MPI Forum. It is considered to be a portable message-passing interface designed to work between the parallel computers in a cluster. The main advantage of working with a cluster is to have access to large computational power over a distributed domain.

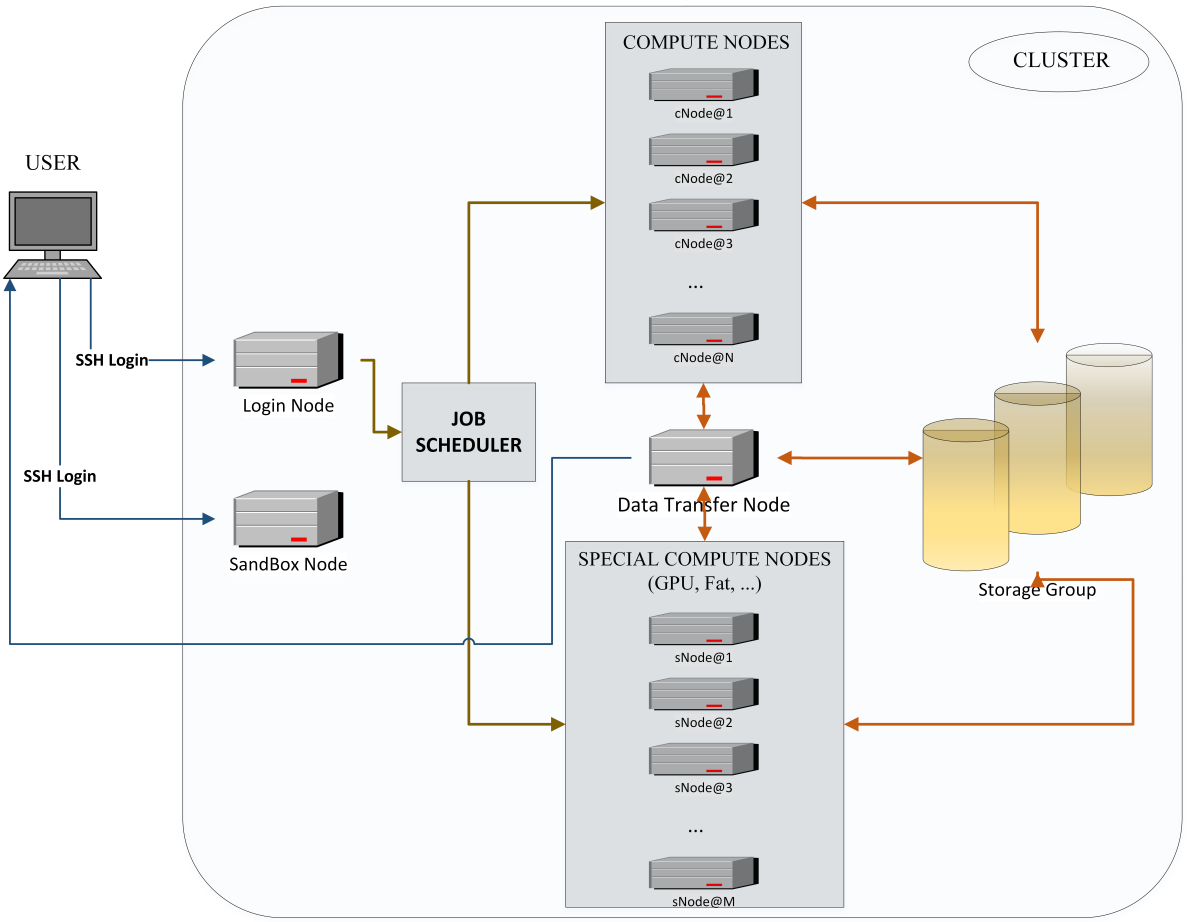


Figure 7.3: Parallelization on HPC cluster Illustration

Figure 7.3 shows a rough illustration of an HPC cluster. It is a collection of multiple server nodes connected via Fast LAN. Different kinds of server nodes are present that serve various user requirements. The **Login Node**, as the name states are used to access the cluster as a whole. Once, logged in, the user can write job scripts with specifications on the number of cores and types of computing nodes needed to run the tasks. The clusters use various batch queuing systems like Univa Grid Engine, Slurm, etc. as job schedulers. These jobs are then submitted to the regular **Compute Nodes** or the **Special Compute**

nodes (GPU, Fat) per user requirements. The **Storage Unit** stores the user data from the completed jobs or intermediate data created during the job. All these components are interconnected using Fast LAN. Additionally, there could be a **SandBox Node** or a Compile node which can be used for development purposes to test-run the code.

```
from mpi4py import MPI
import numpy as np

tags = enum('READY', 'DONE', 'EXIT', 'START', 'IDLE', 'CONTINUE')

cw = MPI.COMM_WORLD   # get MPI communicator object
size = cw.size        # total number of processes
rank = cw.rank        # rank of this process
status = MPI.Status()   # get MPI status object

if rank == 0:
    s = 0
    c = 0
    nJobs = 10 # User Input
    samples = 60 # User Input
    processes = size-1
    pids = []
    result = []

    for j in range(1, jobs+1):
        while c < processes:
            data = cw.recv(source=MPI.ANY_SOURCE, tag=MPI.ANY_TAG,
                   status=status)
            source = status.Get_source()
            tag = status.Get_tag()

            if tag == tags.READY:
```

```
                if source not in pids:
                    if s < len(samples):
                        cw.send(obj=[samples[s]], dest=status.Get_source()
                        , tag=tags.START)
                        s += 1
                    else:
                        if (j == nJobs):
                            cw.send(obj=None, dest=source, tag=tags.EXIT)
                        else:
                            cw.send(obj=None, dest=source, tag=tags.IDLE)
                            proc_ids.append(source)
                else:
                    cw.send(obj=None, dest=source, tag=tags.CONTINUE)

            elif tag == tags.DONE:
                result.append(data)

            elif tag == tags.IDLE or tag == tags.EXIT:
                c += 1

        output = np.array([p for p in result])

else:
    while True:
        cw.send(None, dest=0, tag=tags.READY)
        args = cw.recv(source=0, tag=MPI.ANY_TAG, status=status)
        tag = status.Get_tag()

        if tag == tags.START:
            p_result = adaptive_localization(args[0])
            cw.send(p_result, dest=0, tag=tags.DONE)
```

```
        elif tag == tags.IDLE:
            cw.send(None, dest=0, tag=tags.IDLE)

        elif tag == tags.CONTINUE:
            continue

        elif tag == tags.EXIT:
            break

    cw.send(None, dest=0, tag=tags.EXIT)
```

Listing 7.3: MPI Implementation

Listing 7.3 shows the implementation of **adaptive_localization** method using the MPI module. This is the approach developed to run multiple samples of SF grids on an HPC cluster. Here, we have a parent process number 0 which sends and receives data from child processes. Tags are initiated with commands to be sent and received between master-child processes. The parent process issues five commands - START, EXIT, IDLE, CONTINUE, DONE and receives four commands - READY, DONE, IDLE, EXIT. The parent process actively monitors the job status and the child-process queue and issues command to the child based on queue status. e.g. when it receives a READY command from the child, it sends the task over given that there are tasks in the task queue. If the task queue is empty, it issues an EXIT command to the child.

On the other hand, the child process issues three commands - READY, DONE, IDLE, EXIT and receives four commands - START, IDLE, CONTINUE, EXIT. The child initiates a START command to let the parent know that it is ready to receive the tasks. Once it receives a task, it performs computations and sends it back to the parent using the tag DONE. The child process gracefully terminates when it receives the EXIT tag from the parent. These results from individual processes (for respective samples) are consolidated on the parent.

7.3 Analysis of Parallelization Techniques

This section states the analysis of the above-discussed techniques for parallelization. Here, we use sample SF grid data and analyze the processing time for these grids using (a) serial, (b) multiprocessing, and (c) MPI approach. Table 7.1 shows the notations used to define the processing time equations for each of these approaches.

Table 7.1: Notations use to define parallelization equations

NOTATION	DESCRIPTION
T_S	Processing time for serially programmed method
T_P	Processing time for a method programmed using multiprocessing (Pool)
T_M	Processing time for a method programmed using MPI (mpi4py)
nSamples	Number of samples to be processed
nCores	Number of cores requested at Login

$$T_S = \sum_{i=1}^{nSamples} T_i \tag{7.1}$$

IF $(nCores < nSamples)$

$$T_P = max(T_i) + max(T_j) \tag{7.2}$$

where,

$\forall\ i \in \{1, 2, \ldots , \text{nCores}\}$

$\forall\ j \in \{\text{nCores}, \ldots , \text{nSamples}\}$

ELSE:

$$T_P = max(T_i) \tag{7.3}$$

where,

$\forall\ i \in \{1, 2, \ldots , \text{nSamples}\}$

$$T_M = max(T_i) \tag{7.4}$$

where,

$\forall\ i \in \{1, 2, \ldots , \text{nSamples}\}$

The equations 7.1, 7.2, 7.3, and 7.4 state the processing times required for each of the modules described in 7.2 to run an algorithm on SF grid data-sets. Let's say we have $nSamples$ of a Smart Fabric grid with P_M percentage missing nodes and V voids. Now, even if P_M and V are the same for all samples, the structure of the grid varies due to the varied placement of voids and the shape of voids. Due to that, the processing time is different for each sample.

For a serialized computation, the processing time would be the sum of processing time for all samples as shown in 7.1. Now, for the Pool multiprocessing module, depending on the system configurations, if we have enough cores to run a process for each sample, the processing time is the maximum time taken by a sample in the set. Else the time taken is the summation of the maximum time taken by a sample in each set as shown in 7.3. However, note that it is not an ideal scenario to have one powerful system to run scalable applications. Hence we will be considering the equation 7.2 henceforth. In MPI, the $nCores$ are requested such as $nCores = nSamples + 1$. Thus, the processing time taken is the maximum time taken by a sample in that set.

Table 7.2: Analysis data for 20 samples (hours) - Serial vs. Pool vs. MPI Programming for grid_simulator

		GRID SIZE (#nodes)					
		9X9 (58)	**19X19 (292)**	**29X29 (706)**	**39X39 (1300)**	**49X49 (2074)**	**59X59 (3028)**
	Serial	1.16	6	16.33	38.33	82.66	174.33
MODULE	**Pool**	0.14	0.7	2.03	4.8866	10.5213	18.765
	MPI	0.0783	0.43	1.134	2.26	4.93	10.1775

Table 7.2 shows the analysis data for different types of computing modules. As we can see, the serial computing numbers are exceptionally high as the time for each sample is added to the processing time. The Pool takes less time for computation but some overhead to create a process pool. Although Pool can have the same number of cores as MPI, note

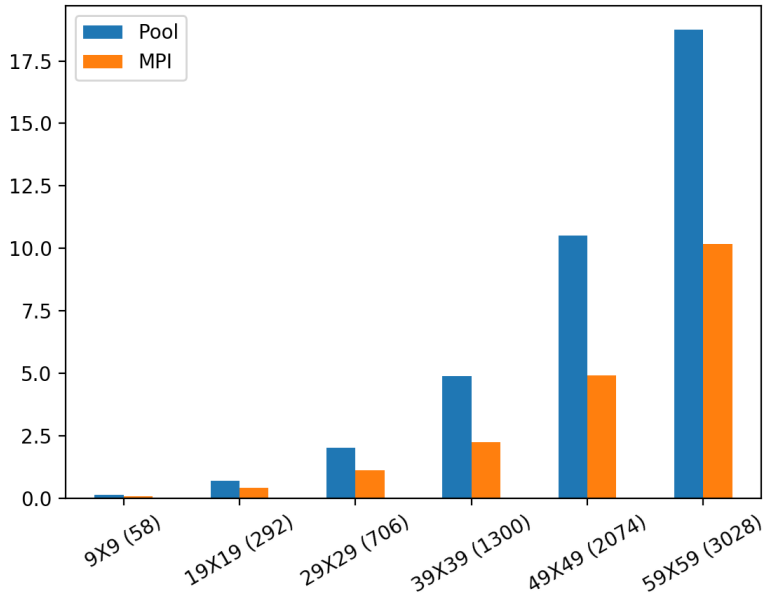


Figure 7.4: Pool vs. MPI implementation on grid_simulator method

that a cluster has dedicated nodes for computation, unlike a single machine that has multiple processes running. Hence, even with the same number of cores, the Pool cannot exploit the system efficiently. On the other hand, MPI guarantees the job to start with the requested number of cores with high computational power. Figure 7.4 shows a plot of Pool vs. MPI module for the **grid_simulator** method for 20 samples. For a smaller grid size, we do not see much difference in the processing time. However, as the grid size increases, we see that the processing time for MPI is approximately half as compared to Pool.

BIBLIOGRAPHY

[1] N. Tyler, "While smart textiles for wearables remains in its infancy, its potential is huge," *newelectronics, The site for electronic design engineers*, pp. 14–16, 2016.

[2] G. A. Pendharkar and A. P. Jayasumana, *Virtual Coordinate Systems and Coordinate-Based Operations for IoT*. Cham: Springer International Publishing, 2019, pp. 159–207. [Online]. Available: https://doi.org/10.1007/978-3-319-93557-7_10

[3] R. Negra, I. Jemili, and A. Belghith, "Wireless body area networks: Applications and technologies," *Procedia Computer Science*, vol. 83, pp. 1274 – 1281, 2016, the 7th International Conference on Ambient Systems, Networks and Technologies (ANT 2016) / The 6th International Conference on Sustainable Energy Information Technology (SEIT-2016) / Affiliated Workshops. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S187705091630299X

[4] W. H. O. (WHO), "Current health expenditure as a percentage of gross domestic product (gdp)," *Global Health Observatory (GHO)*, 2015. [Online]. Available: https://www.who.int/gho/health_financing/health_expenditure/en/

[5] CMS, "National health expenditure data - historical," *U.S. Centers for Medicare and Medicaid Services*, 2017. [Online]. Available: https://www.cms.gov/Research-Statistics-Data-and-Systems/Statistics-Trends-and-Reports/NationalHealthExpendData/NationalHealthAccountsHistorical

[6] W. H. O. (WHO), "World report on ageing and health," *Ageing and life-course*, 2015.

[7] J. Habetha, "The myheart project - fighting cardiovascular diseases by prevention and early diagnosis," in *2006 International Conference of the IEEE Engineering in Medicine and Biology Society*, vol. Supplement, Aug 2006, pp. 6746–6749.

[8] R. Paradiso, G. Loriga, N. Taccini, A. Gemignani, and B. Ghelarducci, "Wealthy, a wearable health-care system: New frontier on etextile," *Journal of Telecommunications and Information Technology*, vol. 4, pp. 105–113, 01 2005.

[9] A. Mazzoldi, D. de rossi, F. Lorussi, E. Scilingo, and R. Paradiso, "Smart textiles for wearable motion capture systems," *AUTEX Res. J.*, vol. 2, pp. 199–203, 12 2002.

[10] M. Chen, Y. Ma, J. Song, C.-F. Lai, and B. Hu, "Smart clothing: Connecting human with clouds and big data for sustainable health monitoring," *Mobile Networks and Applications*, vol. 21, no. 5, pp. 825–845, 2016. [Online]. Available: https://doi.org/10.1007/s11036-016-0745-1

[11] P. Pandian, K. Mohanavelu, K. Safeer, T. Kotresh, D. Shakunthala, P. Gopal, and V. Padaki, "Smart vest: Wearable multi-parameter remote physiological monitoring system," *Medical Engineering & Physics*, vol. 30, no. 4, pp. 466 – 477, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1350453307000975

[12] B. Mukhopadhyay, S. Sarangi, and S. Kar, "Novel rssi evaluation models for accurate indoor localization with sensor networks," in *2014 Twentieth National Conference on Communications (NCC)*, Feb 2014, pp. 1–6.

[13] D. Turner, S. Savage, and A. C. Snoeren, "On the empirical performance of self-calibrating wifi location systems," in *2011 IEEE 36th Conference on Local Computer Networks*, Oct 2011, pp. 76–84.

[14] P. N. Pathirana, N. Bulusu, A. V. Savkin, and S. Jha, "Node localization using mobile robots in delay-tolerant sensor networks," *IEEE Transactions on Mobile Computing*, vol. 4, no. 3, pp. 285–296, May 2005.

[15] R. Hovorka, L. Chassin, M. Wilinska, V. Canonico, J. Akwe, M. Federici, M. Massi-Benedetti, I. Hutzli, C. Zaugg, H. Kaufmann, M. Both, T. Vering, H. Schaller,

L. Schaupp, M. Bodenlenz, and T. Pieber, "Closing the loop: The adicol experience," *Diabetes technology & therapeutics*, vol. 6, pp. 307–18, 07 2004.

[16] P. Rubel, F. Gouaux, J. Fayn, D. Assanelli, A. Cuce, L. Edenbrandt, and C. Malossi, "Towards intelligent and mobile systems for early detection and interpretation of cardiological syndromes," in *Computers in Cardiology 2001. Vol.28 (Cat. No.01CH37287)*, Sep. 2001, pp. 193–196.

[17] A. Halteren, R. Bults, K. Wac, N. Dokovsky, G. Koprinkov, I. Widya, D. Konstantas, V. Jones, and R. Herzog, "Wireless body area networks for healthcare: the mobihealth project," *Studies in health technology and informatics*, vol. 108, pp. 181–93, 02 2004.

[18] R. Paradiso, A. Gemignani, E. P. Scilingo, and D. De Rossi, "Knitted bioclothes for cardiopulmonary monitoring," in *Proceedings of the 25th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (IEEE Cat. No.03CH37439)*, vol. 4, Sep. 2003, pp. 3720–3723 Vol.4.

[19] M. Di Rienzo, F. Rizzo, G. Parati, G. Brambilla, M. Ferratini, and P. Castiglioni, "Magic system: a new textile-based wearable device for biological signal monitoring. applicability in daily life and clinical setting," *Conference proceedings : ... Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE Engineering in Medicine and Biology Society. Conference*, vol. 7, pp. 7167–9, 02 2005.

[20] D. C. Dhanapala and A. P. Jayasumana, "Anchor selection and topology preserving maps in wsns — a directional virtual coordinate based approach," in *2011 IEEE 36th Conference on Local Computer Networks*, Oct 2011, pp. 571–579.

[21] J. Newsome and D. Song, "Gem: Graph embedding for routing and data-centric storage in sensor networks without geographic information," in *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, ser. SenSys '03.

New York, NY, USA: Association for Computing Machinery, 2003, p. 76–88. [Online]. Available: https://doi.org/10.1145/958491.958501

[22] D. C. Dhanapala and A. P. Jayasumana, "Directional virtual coordinate systems for wireless sensor networks," in *2011 IEEE International Conference on Communications (ICC)*, June 2011, pp. 1–6.

[23] B. Donnet, B. Gueye, and M. A. Kaafar, "A survey on network coordinates systems, design, and security," *IEEE Communications Surveys Tutorials*, vol. 12, no. 4, pp. 488–503, Fourth 2010.

[24] A. Gunathillake, A. V. Savkin, and A. P. Jayasumana, "Maximum likelihood topology maps for wireless sensor networks using an automated robot," in *2016 IEEE 41st Conference on Local Computer Networks (LCN)*, Nov 2016, pp. 339–347.

[25] D. C. Dhanapala and A. P. Jayasumana, "Topology preserving maps—extracting layout maps of wireless sensor networks from virtual coordinates," *IEEE/ACM Transactions on Networking*, vol. 22, no. 3, pp. 784–797, June 2014.

[26] A.-M. Kermarrec, A. Mostefaoui, M. Raynal, G. Tredan, and A. C. Viana, "Large-scale networked systems: From anarchy to geometric self-structuring," in *Distributed Computing and Networking*, V. Garg, R. Wattenhofer, and K. Kothapalli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 25–36.

[27] M. Shah and A. Sardana, "Searching in internet of things using vcs," in *Proceedings of the First International Conference on Security of Internet of Things*, ser. SecurIT '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 63–67. [Online]. Available: https://doi.org/10.1145/2490428.2490437

[28] M. Li, P. Jia, Y. Xu, and Y. Yuan, "Traveling path tracking algorithm in virtual coordinate system for intelligent vehicle," in *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*, vol. 03, Oct 2012, pp. 1183–1187.

[29] P. Leone and K. Samarasinghe, "Greedy routing on virtual raw anchor coordinate (vrac) system," in *2016 International Conference on Distributed Computing in Sensor Systems (DCOSS)*, May 2016, pp. 52–58.

[30] A. P. Jayasumana, R. Paffenroth, and S. Ramasamy, in *2016 IEEE International Conference on Communications (ICC)*, May 2016, pp. 1–6.

[31] J. Sheu, M. Ding, and K. Hsieh, "Routing with hexagonal virtual coordinates in wireless sensor networks," in *2007 IEEE Wireless Communications and Networking Conference*, March 2007, pp. 2929–2934.

[32] A. Rao, S. Ratnasamy, C. Papadimitriou, S. Shenker, and I. Stoica, "Geographic routing without location information," in *Proceedings of the 9th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 96–108. [Online]. Available: https://doi.org/10.1145/938985.938996

[33] P. Cheng, T. Han, X. Zhang, R. Zheng, and Z. Lin, "A single-mobile-anchor based distributed localization scheme for sensor networks," in *2016 35th Chinese Control Conference (CCC)*, July 2016, pp. 8026–8031.

[34] H.-J. Jin and J. Weissmüller, "A material with electrically tunable strength and flow stress," *Science*, vol. 332, no. 6034, pp. 1179–1182, 2011. [Online]. Available: https://science.sciencemag.org/content/332/6034/1179

[35] J. Bensmail, F. M. Inerney, and N. Nisse, "Metric dimension: from graphs to oriented graphs," *10th Latin & American Algorithms*, no. 02098194, pp. 111–123, 2019.

[36] E. W. Weisstein, "Np-hard problem," http://mathworld.wolfram.com/NP-HardProblem.html, accessed: 2020-17-03.

[37] J. D´iaz, O. Pottonen, and E. J. van Leeuwen, "Planar metric dimension is np-complete," *CoRR*, vol. abs/1107.2256, 2011. [Online]. Available: http://arxiv.org/abs/1107.2256

[38] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.

[39] S. Pritam, "Virtual coordinate based techniques for wireless sensor networks: A simulation tool and localization & planarization algorithms," Master's thesis, Walter Scott, Jr. College of Engineering, 2013.

[40] A. Marowka, "Performance study of the first three intel multicore processors." *Scalable Computing: Practice and Experience*, vol. 10, 01 2009.

[41] P. Mane. (2017) Python multiprocessing: Pool vs process – comparative analysis. [Online]. Available: https://www.ellicium.com/python-multiprocessing-pool-process/

[42] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, and B. Chapman, "High performance computing using mpi and openmp on multi-core parallel systems," *Parallel Computing*, vol. 37, no. 9, pp. 562 – 575, 2011, emerging Programming Paradigms for Large-Scale Scientific Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111000159

[43] wiki, "Computer cluster," 2019, [Online; accessed 22-February-2019]. [Online]. Available: https://en.wikipedia.org/wiki/Computer_cluster

[44] J. Andreu-Perez, D. R. Leff, H. M. D. Ip, and G. Yang, "From wearable sensors to smart implants-–toward pervasive and personalized healthcare," *IEEE Transactions on Biomedical Engineering*, vol. 62, no. 12, pp. 2750–2762, Dec 2015.

[45] M. J. Gardner and D. G. Altman, "Confidence intervals rather than p values: estimation rather than hypothesis testing." *BMJ*, vol. 292, no. 6522, pp. 746–750, 1986. [Online]. Available: https://www.bmj.com/content/292/6522/746

[46] T. H. Illangasekare, Q. Han, and A. P. Jayasumana, "Environmental underground sensing and monitoring," in *Underground Sensing*, S. Pamukcu and L. Cheng, Eds. Academic Press, 2018, pp. 203 – 246. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780128031391000047

[47] P. Bose, P. Morin, I. Stojmenović, and J. Urrutia, "Routing with guaranteed delivery in ad hoc wireless networks," *Wirel. Netw.*, vol. 7, no. 6, p. 609–616, Nov. 2001. [Online]. Available: https://doi.org/10.1023/A:1012319418150

[48] Qing Cao and T. Abdelzaher, "A scalable logical coordinates framework for routing in wireless sensor networks," in *25th IEEE International Real-Time Systems Symposium*, 2004, pp. 349–358.

[49] D. B. Johnson, D. A. Maltz, and J. Broch, *DSR: The Dynamic Source Routing Protocol for Multihop Wireless Ad Hoc Networks*. USA: Addison-Wesley Longman Publishing Co., Inc., 2001, p. 139–172.

[50] R. Jin, H. Wang, B. Peng, and N. Ge, "Research on rssi-based localization in wireless sensor networks," in *2008 4th International Conference on Wireless Communications, Networking and Mobile Computing*, 2008, pp. 1–4.

[51] B. Karp and H. T. Kung, "Gpsr: Greedy perimeter stateless routing for wireless networks," in *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 243–254. [Online]. Available: https://doi.org/10.1145/345910.345953

[52] C. Finn and D. L. Williams, "An aeromagnetic study of mount st. helens," *Journal of Geophysical Research: Solid Earth*, vol. 92, no. B10, pp. 10 194–10 206, 1987. [Online]. Available: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/JB092iB10p10194

[53] I. Bose, J. Irazoqui, J. Moskow, E. Bardes, T. Zyla, and D. Lew, "Assembly of scaffold-mediated complexes containing cdc42p, the exchange factor cdc24p, and the effector cla4p required for cell cycle-regulated phosphorylation of cdc24p," *The Journal of biological chemistry*, vol. 276, pp. 7176–86, 04 2001.

[54] D. Niculescu and B. Nath, "Dv based positioning in ad hoc networks," *Telecommunication Systems*, vol. 22, pp. 267–280, 01 2003.

[55] E. Parnell, M. Viering, K. Rhodes, and P. Geyer, "A test of insulator interactions in drosophila," *The EMBO journal*, vol. 22, pp. 2463–71, 06 2003.

[56] H. Frey, S. Ruehrup, and I. Stojmenović, *Routing in Wireless Sensor Networks*, 05 2009, pp. 81–111.

[57] J. N. Al-Karaki and A. E. Kamal, "Routing techniques in wireless sensor networks: a survey," *IEEE Wireless Communications*, vol. 11, no. 6, pp. 6–28, 2004.

[58] I. Amundson and X. Koutsoukos, *A Survey on Localization for Mobile Wireless Sensor Networks*, 09 2009, vol. 5801, pp. 235–254.

[59] A. P. Jayasumana, Q. Han, and T. H. Illangasekare, "Virtual sensor networks - a resource efficient approach for concurrent applications," in *Fourth International Conference on Information Technology (ITNG'07)*, 2007, pp. 111–115.

[60] R. Flury and R. Wattenhofer, "Randomized 3d geographic routing," in *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*, 2008, pp. 834–842.

[61] T. R. Babu, A. Chatterjee, S. Khandeparker, A. V. Subhash, and S. Gupta, "Geographical address classification without using geolocation coordinates," in

Proceedings of the 9th Workshop on Geographic Information Retrieval, ser. GIR '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2837689.2837696

[62] J. Dong, K. E. Ackermann, B. Bavar, and C. Nita-Rotaru, "Secure and robust virtual coordinate system in wireless sensor networks," *ACM Trans. Sen. Netw.*, vol. 6, no. 4, Jul. 2010. [Online]. Available: https://doi.org/10.1145/1777406.1777408

[63] J. Seibert, S. Becker, C. Nita-Rotaru, and R. State, "Newton: Securing virtual coordinates by enforcing physical laws," *IEEE/ACM Transactions on Networking*, vol. 22, no. 3, pp. 798–811, 2014.

[64] D. Zage and C. Nita-Rotaru, "Robust decentralized virtual coordinate systems in adversarial environments," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 4, Dec. 2010. [Online]. Available: https://doi.org/10.1145/1880022.1880032

[65] S. Beckery, J. Seibert, D. Zage, C. Nita-Rotaru, and R. Statey, "Applying game theory to analyze attacks and defenses in virtual coordinate systems," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, 2011, pp. 133–144.

[66] Qing Fang, Jie Gao, L. J. Guibas, V. de Silva, and Li Zhang, "Glider: gradient landmark-based distributed routing for sensor networks," in *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, vol. 1, 2005, pp. 339–350 vol. 1.

[67] J. Bruck, J. Gao, and A. Jiang, "Map: Medial axis based geometric routing in sensor networks," *Wireless Networks*, vol. 13, pp. 835–853, 2007.

[68] A. Cvetkovski and M. Crovella, "Hyperbolic embedding and routing for dynamic graphs," in *IEEE INFOCOM 2009*, 2009, pp. 1647–1655.

[69] M. . Tsai, H. . Yang, and W. . Huang, "Axis-based virtual coordinate assignment protocol and delivery-guaranteed routing protocol in wireless sensor networks," in *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*, 2007, pp. 2234–2242.

[70] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: A decentralized network coordinate system," *ACM SIGCOMM Computer Communication Review*, vol. 34, 08 2004.

[71] L. Lehman and S. Lerman, "A decentralized network coordinate system for robust internet distance," in *Third International Conference on Information Technology: New Generations (ITNG'06)*, 2006, pp. 631–637.

[72] T. S. E. Ng and H. Zhang, "A network positioning system for the internet," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '04. USA: USENIX Association, 2004, p. 11.

[73] T. S. E. Ng and Hui Zhang, "Predicting internet network distance with coordinates-based approaches," in *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 1, 2002, pp. 170–179 vol.1.

[74] P. Francis, S. Jamin, Cheng Jin, Yixin Jin, D. Raz, Y. Shavitt, and L. Zhang, "Idmaps: a global internet host distance estimation service," *IEEE/ACM Transactions on Networking*, vol. 9, no. 5, pp. 525–540, 2001.

[75] M. Pias, J. Crowcroft, S. Wilbur, T. Harris, and S. Bhatti, "Lighthouses for scalable distributed location." 01 2003, pp. 278–291.

[76] M. Costa, M. Castro, R. Rowstron, and P. Key, "Pic: practical internet coordinates for distance estimation," in *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, 2004, pp. 178–187.

[77] L. J. Guibas, C. Holleman, and L. E. Kavraki, "A probabilistic roadmap planner for flexible objects with a workspace medial-axis-based sampling approach," in *Proceedings 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human and Environment Friendly Robots with High Intelligence and Emotional Quotients (Cat. No.99CH36289)*, vol. 1, 1999, pp. 254–259 vol.1.

[78] N. Amenta, M. Bern, and D. Eppstein, "The crust and the beta-skeleton: Combinatorial curve reconstruction." *Graphical Models and Image Processing*, vol. 60, pp. 125–135, 01 1998.

[79] N. Amenta, S. Choi, and R. K. Kolluri, "The power crust, unions of balls, and the medial axis transform," *Comput. Geom. Theory Appl.*, vol. 19, no. 2–3, p. 127–153, Jul. 2001. [Online]. Available: https://doi.org/10.1016/S0925-7721(01)00017-7

[80] H. Choi, S. Choi, and H. Moon, "Mathematical theory of medial axis transform," *Pacific J Math*, vol. 181, 12 1997.

[81] J. Li, J. Jannotti, D. S. J. De Couto, D. R. Karger, and R. Morris, "A scalable location service for geographic ad hoc routing," in *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 120–130. [Online]. Available: https://doi.org/10.1145/345910.345931

[82] L. Tang and M. Crovella, "Virtual landmarks for the internet," in *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 143–152. [Online]. Available: https://doi.org/10.1145/948205.948223

[83] K. L. Calvert, M. B. Doar, and E. W. Zegura, "Modeling internet topology," *IEEE Communications Magazine*, vol. 35, no. 6, pp. 160–163, 1997.

APPENDIX A

LOCALIZATION AND ANCHOR DATA

This section entails the data for the plots in Chapter **5** for all the network sizes - 19X19, 24X24, 29X29, 34X34. Here, we have the information about the percentages of localized node for each void corresponding to a certain percentages of missing nodes in the Smart Fabric. The data has been rounded up to 2 decimal points.

The average value or the mean of data is presented in addition to the confidence interval for the designated value. In statistics, a confidence interval (C. I.) is defined as an estimated range of values which is likely to include an unknown population parameter (in this case the mean), the estimated range being calculated from a given set of sample data (from the 60 samples). This is called an estimation approach [45].

Additionally, the selection of a confidence level (%) for an interval determines the probability that the confidence interval produced will contain the true parameter value. Confidence levels that are commonly used are 0.90, 0.95, and 0.99. These levels correspond to percentages of the area of the normal density curve. Hence, for a confidence level of 95% (0.95), the probability of observing a value outside of this area is less than 0.05. In our example, we have a confidence level of 95%. The C.I. is calculated using the *scipy.stats* library.

Following tables A.1, A.2, A.3, and A.4 hold the data for all the plots from Chapter 5.

Table A.1: Localization and Anchor data for SF Grid with size $19X19$ (60 samples)

NUMBER OF VOIDS	PERCENTAGES OF MISSING NODES (%)																					
	10					20				30				40				50				
	% Ln			Anchors		% Ln		Anchors		% Ln		Anchors		%Ln		Anchors		%Ln		Anchors		
	Avg	C.I.		Avg	C.I.	Avg	C.I.		Avg	C.I.	Avg	C.I.		Avg	C.I.		Avg	C.I.		Avg	C.I.		Avg	C.I.	
1	95.72	95.06	96.38	6	5 6	93.75	92.52	94.97	6	5 6	92.57	91.1	94.04	6	6 7	90.93	89.06	92.8	6	6 7	89.34	86.74	91.94	7	6 7
2	93.67	93.05	94.29	6	6 7	90.56	89.53	91.59	7	7 8	87.02	85.54	88.5	7	7 8	81.65	79.78	83.52	8	7 9	74.09	70.8	77.37	7	7 8
3	94.08	93.44	94.71	6	6 7	88.43	87.37	89.49	7	7 8	83.21	81.62	84.81	8	7 9	77.83	75.71	79.94	8	7 9	69.35	66.43	72.28	7	6 8
4	94.6	94.07	95.13	6	6 7	88.2	87.15	89.26	8	7 8	83.03	81.46	84.6	8	8 9	75.33	72.84	77.82	8	8 9	65.59	63.08	68.11	8	7 9
5	94.54	93.87	95.21	7	6 7	87.43	86.44	88.42	8	8 9	80.98	79.27	82.67	8	7 9	71.39	69.24	73.54	8	8 9	64.26	61.18	67.34	7	6 8

Table A.2: Localization and Anchor data for SF Grid with size $24X24$ (60 samples)

NUMBER OF VOIDS	PERCENTAGES OF MISSING NODES (%)																					
	10					20				30				40				50				
	% Ln			Anchors		% Ln		Anchors		% Ln		Anchors		%Ln		Anchors		%Ln		Anchors		
	Avg	C.I.		Avg	C.I.	Avg	C.I.		Avg	C.I.	Avg	C.I.		Avg	C.I.		Avg	C.I.		Avg	C.I.		Avg	C.I.	
1	96.2	95.63	96.71	6	6 7	94.9	93.91	95.9	6	5 6	93.71	92.35	95.07	7	6 8	92.1	90.42	93.78	7	6 7	91.16	88.8	93.52	7	6 8
2	94.85	94.22	95.48	8	7 8	91.74	90.79	92.69	8	7 9	88.9	87.47	90.34	8	7 9	85.02	83.02	87.01	8	7 9	79.43	76.84	82.02	9	7 10
3	94.28	93.8	94.76	8	7 8	88.92	88.02	89.81	9	8 9	84.97	83.54	86.4	9	8 10	81.83	80.28	83.38	10	9 11	74.89	72	77.77	9	8 10
4	93.37	92.82	93.91	8	7 8	88.4	87.27	89.54	9	8 10	83.11	81.6	84.63	10	9 10	77.88	76.21	79.56	10	9 11	71.81	69.5	74.12	10	9 11
5	93.72	93.19	94.25	8	7 9	88.44	87.64	89.24	10	10 11	81.14	79.54	82.75	10	9 11	74.6	72.66	76.55	11	10 12	68.06	65.7	70.41	10	9 11

Table A.3: Localization and Anchor data for SF Grid with size $29X29$ (60 samples)

NUMBER OF VOIDS	PERCENTAGES OF MISSING NODES (%)																							
	10					20				30				40				50						
	% Ln			Anchors		% Ln			Anchors	% Ln			Anchors	%Ln			Anchors	%Ln			Anchors		
	Avg	C.I.		Avg	C.I.	Avg	C.I.		Avg	C.I.	Avg	C.I.		Avg	C.I.	Avg	C.I.	Avg	C.I.		Avg	C.I.	Avg	C.I.	
1	97.04	96.54	97.55	6	6 7	96.91	96.14	97.68	7	6 8	95.03	93.97	96.1	7	6 7	93.33	91.75	94.91	7	6 8	94.67	92.81	96.52	7	6 8
2	95	94.56	95.41	7	7 8	92.68	91.83	93.54	8	7 9	89.91	88.63	91.19	9	8 10	87.39	85.61	89.2	10	9 11	80.23	77.36	83.23	10	9 11
3	94.81	94.33	95.3	9	8 9	90.63	89.77	91.5	9	8 10	85.77	84.09	87.44	10	10 11	81.78	79.67	83.89	10	9 11	75.52	72.58	78.45	10	9 12
4	93.7	93.22	94.14	9	8 9	89.54	88.77	90.31	10	9 11	84.58	83.29	85.86	11	10 12	79.93	78.06	81.79	12	11 13	68.13	64.41	71.85	12	11 13
5	93.93	93.47	94.40	9	9 10	89.49	88.7	90.27	11	10 12	83.16	81.86	84.47	12	11 13	76.56	74.79	78.33	13	11 14	70.31	67.89	72.72	13	11 14

Table A.4: Localization and Anchor data for SF Grid with size $34X34$ (60 samples)

NUMBER OF VOIDS	PERCENTAGES OF MISSING NODES (%)																							
	10					20				30				40				50						
	% Ln			Anchors		% Ln			Anchors	% Ln			Anchors	%Ln			Anchors	%Ln			Anchors		
	Avg	C.I.		Avg	C.I.	Avg	C.I.		Avg	C.I.	Avg	C.I.		Avg	C.I.	Avg	C.I.	Avg	C.I.		Avg	C.I.	Avg	C.I.	
1	97.16	96.77	97.55	7	6 7	95.69	94.91	96.46	7	6 8	95.7	94.72	96.69	8	7 9	94.77	93.53	96	8	7 9	95.33	93.88	96.79	7	6 8
2	95.6	95.2	96	8	8 9	93.17	92.46	93.89	10	9 11	92	91.05	92.95	9	8 10	88.86	87.34	90.38	10	9 11	85.44	83.29	87.58	11	9 12
3	95.27	94.9	95.63	9	9 10	91.95	91.24	92.66	10	9 11	88.4	87.07	89.73	11	10 12	84.45	82.34	86.55	12	10 13	77.89	75.02	80.73	12	11 14
4	94.52	94.1	94.94	10	9 10	90.83	90.12	91.54	11	10 12	85.92	84.96	86.88	12	11 13	81.77	80.13	83.42	13	12 14	74.94	72.47	77.41	12	11 14
5	94.34	93.9	94.77	11	10 11	90.55	89.89	91.21	12	11 12	84.76	83.75	85.77	13	12 15	77	74.17	79.79	15	13 16	69.67	66.28	73.06	15	13 16

Table A.5: Localization and Anchor data for SF Grid with size $39X39$ (60 samples)

NUMBER OF VOIDS	PERCENTAGES OF MISSING NODES (%)																					
	10				25				40				55				70					
	% Ln		Anchors		% Ln		Anchors		% Ln		Anchors		%Ln		Anchors		%Ln		Anchors			
	Avg	C.I.	Avg	C.I.	Avg	C.I.	Avg	C.I.	Avg	C.I.	Avg	C.I.	Avg	C.I.	Avg	C.I.	Avg	C.I.	Avg	C.I.		
1	97.37	96.92 97.83	7	6 7	95.99	95.26 96.72	8	7 8	95.8	94.66 96.94	7	6 8	93.94	92.37 95.52	8	6 9	77.28	69.32 85.24	9	7 11		
2	96.14	95.75 96.53	9	8 9	93.04	92.17 93.91	10	9 11	89.49	87.57 91.4	11	10 12	80.93	78.23 83.63	13	11 15	65.58	59.67 71.48	10	8 12		
3	95.13	94.75 95.51	9	9 10	91.45	90.71 92.19	12	11 13	85.9	84.37 87.43	15	13 16	74.36	70.14 78.59	13	12 15	58.55	53.51 63.59	10	8 12		
4	94.74	94.28 95.19	10	10 11	88.59	87.69 89.49	12	12 13	82.37	80.37 84.36	15	13 16	72.31	69.23 75.4	15	13 18	50.69	45.79 55.58	12	11 14		
5	94.04	93.62 94.45	11	10 12	87.71	86.68 88.74	14	12 15	81.22	79.62 82.83	16	14 18	65.89	62.23 69.55	17	15 19	47.18	42.45 51.9	12	10 14		

APPENDIX B

SOURCE CODE

This section contains the source code methods for the Thesis. The programming language used is Python 3.6 and some of the main used libraries are - *collections*, *numpy*, *warnings*, *scipy*, *networkx*, *matplotlib*, *mpi4py*. The code is divided in two main sections - Admin or Driver Methods and Adaptive Localization Methods.

B.1 GRID SIMULATOR

This section contains the code for Grid Simulator including the Grid Generator and helper methods for the same.

B.1.1 DRIVER METHOD

```
def simulate_network(L, B, sX, sY, percent_missing, voids):
    '''Simulates Grid based on user input
    :param L: Length of Grid
    :param B: Breadth of Grid
    :param sX, sY: Base coordinates for the grid
    :param percent_missing: Percentages of missing nodes in SF Grid
    :param voids: Number of voids in SF Grid
    :return Adjacency Matrix for the simulated grid
    '''
    locations, totalNodes = rectangle(L, B, sX, sY)
    innerNodes = totalNodes - (2 * (L + 1) + (2 * (B + 1) - 4))
    node_densities = round((int(percent_missing) * innerNodes) / 100)
    border_nodes, adjacency_matrix = init_comp(locations)
    iterno = 0
    bn_total = len(border_nodes)
```

```
    b_nodes = border_nodes.copy()
    while True:
        iterno += 1
        if iterno == 1:
            pass
        else:
            if len(new_adjacency_matrix) == bn_total + (innerNodes -
                node_densities) and not dis_nodes:
                break
            else:
                b_nodes = border_nodes.copy()
        ran_voids = multiple_voids_generator(node_densities, voids)
        d_nodes = cont_void(L, B, percent_missing, totalNodes, locations,
                  b_nodes, adjacency_matrix, ran_voids)

        with open(str(L) + "x" + str(B) + "_" + str(percent_missing) +
        "_Fabric_Network.txt") as file:
            original_coor = np.array([[float(digit) for digit in line.
                            split()] for line in file])
        new_adjacency_matrix = adj_matrix(original_coor)[0]
        dis_nodes = find_disconn_nodes(L, new_adjacency_matrix)
    if new_adjacency_matrix.size == 0:
        return None
    return new_adjacency_matrix
```

Listing B.1: Grid Simulator

B.1.2 HELPER METHODS

```
def rectangle(L, B, sX, sY):
    '''Method to plot the initial rectangular Grid
    :param L: Length of Grid
    :param B: Breadth of Grid
    :param sX, sY: Base coordinates for the grid
```

```
    '''
    nPoints = (L+1)*(B+1)
    xval = sX
    yval = sY
    arraypoints = np.empty([nPoints,2])
    f = open(str(L) + "x" + str(B) + "_Fabric_Network.txt", "w")
    for rowElement in range(nPoints):
        X, Y = xval, yval
        arraypoints[rowElement] = int(X), int(Y)
        yval += 1
        if yval > (B+sY):
            yval = sY
            xval += 1
            f.write(str(X) + '\t' + str(Y) + '\n')
        else:
            f.write(str(X) + '\t' + str(Y) + '\n')
    f.close()
    return arraypoints, nPoints
```

Listing B.2: Rectangle plot

```
def init_comp(locations):
    '''Method to compute border nodes and the adjacency matrix for the
    ploted rectangle
    :param locations: physical coordinates
    :return: border nodes, adjacency matrix
    '''
    bn = []
    adjacency_matrix = (adj_matrix(locations)[0]).tolist()
    for row in range(len(adjacency_matrix)):
        for col in range(len(adjacency_matrix)):
            if adjacency_matrix[row][col] > 1:
                adjacency_matrix[row][col] = 0
    for node in range(1, len(adjacency_matrix) + 1):
```

```
        neighbor = adjacency_matrix[node - 1].count(1)
        if neighbor == 4:
            pass
        else:
            bn.append(node)
    return bn, adjacency_matrix
```

Listing B.3: Initial Rectangle Computation

```
def adj_matrix(pos_coor):
    '''Method to Compute the adjacency matrix using position
    coordinates
    :param pos_coor: physical coordinates
    :return: adjacency matrix, list of nodes
    '''
    pos = (())
    nodes = []
    if type(pos_coor) == np.ndarray:
        a = pos_coor.tolist()
    else:
        a = pos_coor
    for i in range(len(a)):
        tup = tuple(a[i])
        pos = pos + (tup,)
    list_nodes = list(pos)
    for i in range(1, len(list_nodes) + 1):
        nodeid = i
        nodes.append(nodeid)
    dist_matrix = d.pdist(pos_coor)
    adjacency_matrix = d.squareform(dist_matrix)
    return adjacency_matrix, nodes
```

Listing B.4: Adjacency Matrix Computation

```
def manual_void_creator(L, B, missing, absent_nodes, points,
    node_locations):
```

```
    '''Method to manually create a void using given data points (used
    by "cont_void" function)
    :param L: Length of Grid
    :param B: Breadth of Grid
    :param missing: Percentages of missing nodes
    :param: absent_nodes: Nodes to be deleted
    :param: points: Data points
    :param: node_locations: Physical Coordinates of data points
    '''
    f = open(str(L) + "x" + str(B) + "_" + str(missing) + "_Fabric_Network
   .txt", "w")
    for i in range(points):
        if i+1 in absent_nodes:
            pass
        else:
            X, Y = node_locations[i]
            f.write(str(int(X)) + '\t' + str(int(Y)) + '\n')
    f.close()
```

Listing B.5: Manual Void Creator

```
def random_node_neighbors(node, matrix, bn, visited_nodes, deleted_nodes):
    '''Method to sample random number of neighbors to expand the void.
    This method expands from bfs method as it finds a path to expand the
    void
    :param node: The Grid node
    :param matrix: Adjacency matrix
    :param: bn: Border nodes
    :param: visited_nodes: Already visited nodes
    :param: deleted_nodes: Nodes selected for deletion
    :return: random number of neighbors for the given grid node
    '''
    neighbors = [ni + 1 for ni, x in enumerate(matrix[node - 1]) if x == 1
                 and ni + 1 not in bn and ni + 1 not in visited_nodes
```

```
                and ni + 1 not in deleted_nodes]
    if len(neighbors) > 1:
        no_neighbors = random.randrange(1, len(neighbors) + 1)
    elif len(neighbors) == 1:
        no_neighbors = 1
    else:
        no_neighbors = 0
    return random.sample(neighbors, k=no_neighbors)
```

Listing B.6: Random Neighbor

```
def find_node_neighbors(L, node, bn, deleted_nodes):
    '''Method to avoid deleting the boundaries of the void when
    creating multiple voids
    :param L: Length of Grid
    :param: node: The grid node
    :param: bn: Border nodes
    :param: deleted_nodes: Nodes selected for deletion
    :return: Nodes to be deleted to create a void
    '''
    boundries = [node - (L + 1), node + (L + 1), node - 1, node + 1, node
                 + L, node - L, node + (L + 2), node - (L + 2)]
    return [x for x in boundries if x not in bn and x not in deleted_nodes
    ]
```

Listing B.7: Nodes to be deleted

```
def bfs_connected_component(graph, start, bn, missing_nodes, deleted):
    '''Breadth First Search (BFS) method to create a continuous void
    :param graph: Grid graph created using networkx module
    :param: start: The start node for void creation
    :param: bn: Border nodes
    :param: missing_nodes: Percentages of missing nodes
    :param: deleted_nodes: Nodes selected for deletion
    :return: Nodes to be deleted to create a void
    '''
```

```
    explored = []
    queue = [start]
    visited = [start]
    while queue:
        node = queue.pop(0)
        explored.append(node)
        neighbours = random_node_neighbors(node, graph, bn, visited,
                     deleted)
        for neighbour in neighbours:
            if len(visited) >= missing_nodes:
                    return visited
            if neighbour not in visited:
                queue.append(neighbour)
                visited.append(neighbour)
    return visited
```

Listing B.8: BFS for nodes to be deleted

```
def multiple_voids_generator(N, K):
    '''Method to generate random sized partition values for a given
    percentage
    :param N: Total percentage of missing nodes
    :param K: required number of partitions
    :return: partitioned value of N
    '''
    res = np.ones(K, dtype=int)
    positions = [i for i in range(K)]

    while N - K > 0:
        if positions == []:
            positions = [i for i in range(K)]
        pos = random.choice(positions)
        positions.pop(positions.index(pos))
        if N-K > 2:
```

```
            x = random.randrange(1, (N-K+1)//2)
        else:
            x = N-K
        res[pos] += x
        N -= x
    return res
```

Listing B.9: Random-sized voids

```
def cont_void(L, B, missing, nNodes, locations, bn, adjacency_matrix,
    partitions):
    '''Method to generate a single continuous void
    :param L: Length of Grid
    :param B: Breadth of Grid
    :param: missing_nodes: Percentages of missing nodes
    :param: nNodes: Total number of nodes in grid
    :param: locations: Physical node coordinates
    :param: bn: Border nodes
    :param: adjacency_matrix: Adjacency Matrix of grid
    :param: partitions: Partitions for given missing_nodes
    :return: partitioned value of N
    '''
    deleted_nodes = []
    for p in partitions:
        while True:
            start_node = random.randrange(1, nNodes + 1, step=1)
            if start_node not in bn and start_node not in deleted_nodes:
                break
        deleted_nodes += bfs_connected_component(adjacency_matrix,
                         start_node, bn, p, deleted_nodes)
        for d_node in deleted_nodes:
            bn += find_node_neighbors(L, d_node, bn, deleted_nodes)
    manual_void_creator(L, B, missing, deleted_nodes, nNodes, locations)
```

```
    return len(deleted_nodes)
```

Listing B.10: Continuous voids

```
def short_path(G, source, target):
    '''Method to compute shortest path
    :param G: Graph object created from networkx
    :param source: source node
    :param target: target node
    :return: shortest path, path length
    '''
    (length, path) = nx.single_source_dijkstra(G, source)
    try:
        a = path[target]
        return len(a) - 1, a
    except KeyError:
        raise nx.NetworkXNoPath("node %s not reachable from %s" % (source,
        target))
```

Listing B.11: Shortest path

```
def graph_generator(adjacency_matrix, nodes):
    '''Method to construct the graph using adjacency matrix and node list
    :param: adjacency_matrix: Adjacency Matrix of grid
    :param: nodes: Node list
    :return: Graph object
    '''
    edges = []
    r = 1
    for i in range(len(adjacency_matrix)):
        for j in range(len(adjacency_matrix)):
            if 0 < adjacency_matrix[i][j] <= r:
                edge = nodes[i], nodes[j]
                edges.append(edge)
    G = nx.Graph()
    G.add_nodes_from(nodes)
```

```
    G.add_edges_from(edges)
    return G
```

Listing B.12: Graph generator

```
def find_disconn_nodes(L, adjacency_matrix):
    '''Method to find any disconnected nodes in the grid
    :param: L: Length of grid
    :param: adjacency_matrix: Adjacency Matrix of grid
    :return: Boolean
    '''
    anchor_nodes = [1, L + 1]
    node_IDs = [i for i in range(1, len(adjacency_matrix) + 1)]
    G = graph_generator(np.array(adjacency_matrix).transpose(), node_IDs)
    path_matrix = np.empty([len(node_IDs), len(anchor_nodes)])
    for node in node_IDs:
        try:
            path_matrix[node - 1] = int(short_path(G, node, anchor_nodes
                                        [0])[0]), int(
                short_path(G, node, anchor_nodes[1])[0])
        except:
            return True
    return False
```

Listing B.13: Find disconnected nodes

```
def plot_edges(L, B, percent_missing, voids):
    '''Method to plot the grid with graph edges
    :param: L: Length of gri
    :param: B: Breadth of grid
    :param: percent_missing: Percentages of missing nodes
    :param: voids: Number of voids in grid
    '''
    nodes = []
    edges = []
    points = np.loadtxt(str(L) + "x" + str(B) + "_" + str(percent_missing)
```

```
        + "_Fabric_Network.txt")
x, y = points.T
a = list(zip(x, y))
GnodeXY = np.asarray(a)
pos = (())
for i in range(len(a)):
    tup = tuple(a[i])
    pos = pos + (tup,)
list_nodes = list(pos)
for i in range(1, len(list_nodes) + 1):
    nodeid = i
    nodes.append(nodeid)
dist_matrix = d.pdist(GnodeXY)
sqform_matrix = d.squareform(dist_matrix)
for i in range(len(sqform_matrix)):
    for j in range(len(sqform_matrix)):
        if 0 < sqform_matrix[i][j] <= 1:
            edge = nodes[i], nodes[j]
            edges.append(edge)
G = nx.Graph()
for i in nodes:
    G.add_node(i, pos=list_nodes[i - 1])
G.add_edges_from(edges)
posi = nx.get_node_attributes(G, 'pos')
nx.draw(G, pos=posi, node_color = 'blue', edge_color = 'lightsteelblue
', node_size = 30)
plt.axis('equal')
plt.savefig(str(L) + 'x' + str(B) + '_' + str(percent_missing) + '_' +
str(voids) + '.png' , dpi=200)
```

Listing B.14: Graph plot with edges

B.2 ADAPTIVE LOCALIZATION METHOD

This section contains the code for the Adaptive Localization algorithm. It includes the driver script, MPI driver function, Adaptive Localization method, and the other helper methods.

B.2.1 DRIVER METHODS

```
'''
Script to create a job script and submit job with gievn inputs
'''
#!/bin/bash
if [ -f job.sh ]; then
  rm job.sh
fi
usage() {
  echo ""
  echo "Usage: $0 -n nCores -s nSamples -l nwLength -b nwBreadth -e eArgs
  (y/n)"
  echo -e "\t-n Number of processors needed"
  echo -e "\t-s Number of samples needed"
  echo -e "\t-l Length of the SFN"
  echo -e "\t-b Breadth of the SFN"
  echo -e "\t-e Additional arguements" 1>&2
}
while getopts ":n:s:l:b:t:e:" options; do
  case "${options}" in
    n)
      nCores=${OPTARG}
      ;;
    s)
      nSamples=${OPTARG}
      ;;
```

```
    l)
      nwLength=${OPTARG}
      ;;
    b)
      nwBreadth=${OPTARG}
      ;;
    e)
      eArgs=${OPTARG}
      ;;
    :)
      echo "Error: -${OPTARG} requires an argument."
      exit_abnormal
      ;;
    *)
      exit_abnormal
      ;;
  esac
done
if [ -z "$nCores" ] || [ -z "$nSamples" ] || [ -z "$nwLength" ] || [ -z "
$nwBreadth" ]; then
  echo "Required parameters are empty"
  usage
fi
if [ -f job.sh ]; then
    rm job.sh
fi
cp sub_script.sh job.sh
sed -i 's/$nCores/'$nCores'/g' job.sh
sed -i 's/$L/'$nwLength'/g' job.sh
sed -i 's/$B/'$nwBreadth'/g' job.sh
if [ "$eArgs" = "y" ]; then
  echo "Requesting additional parameters..."
```

```
  read -p "Enter the list of percentages of missing nodes (with spaces): "
  per_missing
  read -p "Enter the list of voids (with spaces): " voids
  qsub job.sh -s $nSamples -l $nwLength -b $nwBreadth -p $per_missing -v
  $voids
else
  echo "No additional parameters"
  qsub job.sh -s $nSamples -l $nwLength -b $nwBreadth
fi
exit 0
```

Listing B.15: Driver script

```
"""MPI Driver Method that gets triggered by the job. This method handles
   inputs from spawned processes and the processing on master process"""
# Define MPI message tags
tags = enum('READY', 'DONE', 'EXIT', 'START', 'IDLE', 'CONTINUE')
# Initializations and preliminaries
comm = MPI.COMM_WORLD   # get MPI communicator object
size = comm.size        # total number of processes
rank = comm.rank        # rank of this process
status = MPI.Status()   # get MPI status object
'''==========================================================================
Main Process
=========================================================================='''
if rank == 0:
    L, B, per_missing_nodes, num_voids, void_type, startX, startY, colors,
    samples = get_inputs()
    res_dir = str(L) + "x" + str(B)
    parent_dir = os.getcwd()
    path = os.path.join(parent_dir, res_dir)
    try:
        os.mkdir(path)
    except OSError:
```

```
        pass
    os.chdir(path)
    delete_missingper_files(L, B)
    plt.figure(figsize=(14, 8))
    data_f = open(str(L) + "x" + str(B) + "_data_file.txt", "w")
    data_f.write('Data Analysis for ' + str(L) + 'x' + str(B) + ' network'
    )
    data_f.write('\n\nLength: {}, Breadth: {}'.format(str(L), str(B)))
    total_nodes = (L + 1) * (B + 1)
    for ind_1, missing in enumerate(per_missing_nodes):
        data_f.write("\n\n-----------------------------------------")
        data_f.write("\nTotal number of nodes in the network for {}%
        missing nodes: {}".format(str(missing), str(total_nodes - round((
        missing/100)*total_nodes))))
        f_1 = open(str(L) + "x" + str(B) + "_correct_nodes.txt", "w")
        f_2 = open(str(L) + "x" + str(B) + "_anchors.txt", "w")
        for ind_2, voids in enumerate(num_voids):
            processes = size-1
            s_index = 0
            closed_processes = 0
            result = []
            proc_ids = []
            while closed_processes < processes:
                data = comm.recv(source=MPI.ANY_SOURCE, tag=MPI.ANY_TAG,
                status=status)
                source = status.Get_source()
                tag = status.Get_tag()
                if tag == tags.READY:
                    if source not in proc_ids:
                        if s_index < len(samples):
                            comm.send(obj=[samples[s_index], startX,
                            startY, L, B, void_type, missing, voids], dest
```

```
                        =status.Get_source(), tag=tags.START)
                        s_index += 1
                    else:
                        if (ind_1 == len(per_missing_nodes) - 1) and (
                        ind_2 == len(num_voids) - 1):
                            comm.send(obj=None, dest=source, tag=tags.
                            EXIT)
                        else:
                            comm.send(obj=None, dest=source, tag=tags.
                            IDLE)
                            proc_ids.append(source)
                else:
                    comm.send(obj=None, dest=source, tag=tags.CONTINUE
                    )
            elif tag == tags.DONE:
                result.append(data)
            elif tag == tags.IDLE or tag == tags.EXIT:
                closed_processes += 1
        per_correct_r = np.array([p[0] for p in result])
        anchors_r = np.array([p[1] for p in result])
        identical_r = np.array([p[2] for p in result])
        mean_correct, lower_correct, upper_correct =
        confidence_interval(per_correct_r.mean(), per_correct_r.std(),
        len(per_correct_r))
        f_1.write(str(mean_correct) + ' ' + str(lower_correct) + ' ' +
        str(upper_correct) + '\n')
        mean_anchor, lower_anchor, upper_anchor = confidence_interval(
        anchors_r.mean(), anchors_r.std(), len(anchors_r))
        f_2.write(str(round(mean_anchor)) + ' ' + str(round(
        lower_anchor)) + ' ' + str(round(upper_anchor)) + '\n')
        mean_identical, lower_identical, upper_identical =
        confidence_interval(identical_r.mean(), identical_r.std(), len
```

```
            (identical_r))
            data_f.write("\n\nVoid: {}".format(str(voids)))
            data_f.write("\nAverage percentage of nodes placed correctly:
            {}; Lower limit: {}; Higher limit: {}".format(str(mean_correct
            ), str(lower_correct), str(upper_correct)))
            data_f.write("\nAverage number of anchors: {}; Lower limit:
            {}; Higher limit: {}".format(str(round(mean_anchor)), str(
            round(lower_anchor)), str(round(upper_anchor))))
            data_f.write("\nAvergae number of identical nodes: {}; Lower
            limit: {}; Higher limit: {}".format(str(round(mean_identical))
            , str(round(lower_identical)), str(round(upper_identical))))
            delete_void_files(L, B, missing, voids)
        f_1.close()
        f_2.close()
        plot(L, B, num_voids, missing, colors[ind_1])
        delete_missingper_files(L, B)
    data_f.write('\n-----------------EOF-------------------')
    data_f.close()
    plt.subplots_adjust(bottom=0.2)
    lgd = plt.legend(loc='upper left', bbox_to_anchor = (-1.15, -0.10),
    fancybox=True, ncol=5)
    plt.savefig(str(L) + "x" + str(B) + " network results.png",
    bbox_extra_artists=(lgd,), bbox_inches='tight', dpi=200)
    plt.clf()
'''=============================================================================
Child Processes
=============================================================================='''
else:
    name = MPI.Get_processor_name()
    while True:
        comm.send(None, dest=0, tag=tags.READY)
        pos = comm.recv(source=0, tag=MPI.ANY_TAG, status=status)
```

```
        tag = status.Get_tag()
        if tag == tags.START:
            res_dir = str(pos[3]) + "x" + str(pos[4])
            parent_dir = os.getcwd()
            path = os.path.join(parent_dir, res_dir)
            try:
                os.chdir(path)
            except OSError:
                pass
            proc_result = network_computation(pos[0], pos[1], pos[2], pos
            [3], pos[4], pos[5], pos[6], pos[7])
            comm.send(proc_result, dest=0, tag=tags.DONE)
        elif tag == tags.IDLE:
            comm.send(None, dest=0, tag=tags.IDLE)
        elif tag == tags.CONTINUE:
            continue
        elif tag == tags.EXIT:
            break
    comm.send(None, dest=0, tag=tags.EXIT)
```

Listing B.16: MPI Driver

```
def adaptive_network_reconstruction(pos, L, B, adjacency_matrix, m, v):
    '''Adaptive localization method for localization
    :param pos: Sample number
    :param: L: Length of Grid
    :param B: Breadth of Grid
    :param adjacency_matrix: Adjacency Matrix of Grid
    :param m: Percentages of missing nodes
    :param v: Number of voids
    :return: Number of iterations, number of anchors, localized nodes
    '''
    r_nodes = 0
    iteration = 0
```

```
derived_nodes = []
no_anchors = []
corrected_positions = {}
anchors = OrderedDict([])
weights = OrderedDict([])
loc_nodes = []
steps = []
map_nodeids_vcs = vcs_two_anchors(L, adjacency_matrix)
node_IDs = [i for i in range(1, len(adjacency_matrix) + 1)]
weights[0] = compute_boundary(adjacency_matrix.tolist())
inner_nodes = sorted(list(set(node_IDs) - set(weights[0])))
w = 0
while 1:
    iteration += 1
    if iteration == 1:
        anchors = OrderedDict([(1, [0, 0]), (L + 1, [0, L])])
        filter_nodes = {}
    else:
        if len(anchors) > 2:
            if r_nodes - temp <= 0:
                del anchors[last_added]
                no_anchors.append(last_added)
            else:
                temp = r_nodes
                w = 0
        else:
            temp = r_nodes
        if (r_nodes == 100) or (iteration > (len(adjacency_matrix) -
        1) // 3):
            pos_coor_1, id_count = identical_node_placement(pos_coor_1
                                    , positions)
            if id_count:
```

```
                r_nodes = len(derived_nodes) + id_count
                r_nodes = round((r_nodes/len(corrected_positions)) *
                100, 2)

            break
        if set(derived_nodes) == set(list(anchors.keys()) + no_anchors
        ):
            pos_coor_1, id_count = identical_node_placement(pos_coor_1
            , positions)
            if id_count:
                r_nodes = len(derived_nodes) + id_count
                r_nodes =
                round((r_nodes/len(corrected_positions)) * 100, 2)
            break
        if not w:
            weights = compute_weights(weights, inner_nodes,
            map_nodeids_vcs)
            w_keys = list(weights.keys())
        while 1:
            anchor_elect = list(set(weights[w_keys[w]]).intersection(
            set(derived_nodes)) - set(list(anchors.keys()) +
            no_anchors))
            if anchor_elect:
                break
            else:
                w += 1
        new_anchor = random.choice(anchor_elect)
        anchors[new_anchor] = pos_coor_1[pos_IDs_1.index(new_anchor)]
        last_added = new_anchor
        path_matrix = vcs_multiple_anchors(list(anchors.keys()),
        adjacency_matrix, node_IDs)
        map_nodeids_vcs = dict(zip(node_IDs, path_matrix.tolist()))
```

```
        if corrected_positions and filter_nodes:
            for node, value in corrected_positions.items():
                filter_nodes = {key1: {key2: [val2 for val2 in val1 if
                val2 != value] for key2, val1 in filter_nodes[key1].
                items()} for key1, filter_nodes[key1] in filter_nodes.
                items()}
    for node in map_nodeids_vcs:
        if node not in derived_nodes:
            deltas = delta_calculator(map_nodeids_vcs, anchors, node,
            L, B)
            corrected_pos, filter_nodes = delta_correction(node,
            filter_nodes, deltas)
            corrected_positions[node] = corrected_pos
    filter_nodes, corrected_positions = data_refiner(filter_nodes,
    corrected_positions)
    similar_pairs, corrected_positions = node_identifier(
    corrected_positions, filter_nodes)
    unidentified_pairs_list = [item for sublist in similar_pairs for
    item in sublist]
    for node, values in corrected_positions.items():
        if type(values) == dict:
            if node not in unidentified_pairs_list:
                similar_pairs.append([node])
    unidentified_pairs_list = [item for sublist in similar_pairs for
    item in sublist]
    positions = deepcopy(corrected_positions)
    for node, values in corrected_positions.items():
        if node not in unidentified_pairs_list and node not in
        derived_nodes:
            old = find_old_neighbors(node, adjacency_matrix)
            new = find_new_neighbors(values, positions)
            if old == new:
```

```
                    filter_nodes = {key1: {key2: [val2 for val2 in val1 if
                    val2 != corrected_positions[node]] for key2, val1 in
                    filter_nodes[key1].items()} for key1, filter_nodes[
                    key1] in filter_nodes.items()}
                    if node in filter_nodes:
                        del filter_nodes[node]
                    filter_nodes = filter_node_refiner(filter_nodes)[0]
                else:
                    corrected_positions[node] = []
        temp_derived_nodes = []
        for node, values in corrected_positions.items():
            if type(values) == dict or not values:
                corrected_positions[node] = []
            else:
                if node not in derived_nodes:
                    temp_derived_nodes.append(node)
        derived_nodes += temp_derived_nodes
        pos_coor_1 = []
        pos_IDs_1 = []
        for node, values in corrected_positions.items():
            pos_IDs_1.append(node)
            if values:
                pos_coor_1.append((values[0], values[1]))
            else:
                pos_coor_1.append(None)
        r_nodes = len(derived_nodes)
        r_nodes = round((r_nodes/len(corrected_positions)) * 100, 2)
        loc_nodes.append(r_nodes)
        steps.append(iteration)
    return iteration, len(anchors.keys()), r_nodes
```

Listing B.17: Adaptive Localization

B.2.2 HELPER METHODS

```
def compute_weights(weights, inner_nodes, node_data):
    '''Method to compute node weights
    :param: weights: empty weight list or previosly computed weights
    :param: inner_nodes: Nodes that are not on the borders
    :param: node_data: VCs of all grid nodes
    :return: weights of nodes
    '''
    for node in inner_nodes:
        new_w = int(min(node_data[node]))
        old_w = [k for k,v in weights.items() if node in v]
        if old_w:
            old_w = old_w[0]
            if old_w > new_w:
                weights[old_w].remove(node)

                if weights[old_w] == []:
                    del weights[old_w]
            else:
                continue
        if new_w in weights.keys():
            weights[new_w].append(node)
        else:
            weights[new_w] = [node]
    return weights
```

Listing B.18: Compute node weights

```
'''Method to optimize the solution space and find corresponding deltas
:param: node_data: VCs of all grid nodes
:param anchors: Elected anchors for the grid
:param node: Node of interest
:param L: Length of grid
:param B: Breadth of grid
```

```
:return: deltas for the node of interest
'''
def delta_calculator(node_data, anchors, node, L, B):
    try:
        test_ab = []
        eq_delta = [None]*len(node_data[node])
        deltas = {}
        val_counter = 0
        V_A = list(map(int, node_data[node]))
        xl_bound, xu_bound = max(0, L - V_A[1]), V_A[0]
        y = min((V_A[0] + V_A[1] - L) // 2, V_A[0])
        anchors = list(anchors.items())
        for b in range(y + 1):
            a = xl_bound
            while xl_bound <= a <= xu_bound:
                eq_delta[0] = int(V_A[0] - (a + b))
                eq_delta[1] = int(V_A[1] - L + (a - b))
                if len(V_A) > 2:
                    for i in range(2, len(V_A)):
                        eq_delta[i] = int(V_A[i] - abs(a - anchors[i
                        ][1][0]) - abs(b - anchors[i][1][1]))
                if all((0 <= eq_delta[n] <= V_A[n]) for n in range(len(V_A
                ))) and a <= L and b <= B:
                    delta = sum(eq_delta)
                    if [a, b] not in test_ab:
                        val_counter += 1
                        if delta in deltas.keys():
                            deltas[delta].append((a, b))
                        else:
                            deltas[delta] = [(a, b)]
                        if val_counter == 1:
                            temp = [a, b]
```

```
                    else:
                        pass
                else:
                    pass
                a += 1
            if xl_bound == 0:
                pass
            else:
                xl_bound += 1
            xu_bound -= 1
        if val_counter == 1:
            test_ab.append(temp)
        return deltas
    except ValueError:
        return None
```

Listing B.19: Delta Optimization

```
'''Methods to refine the deltas and nodes
:param: filter_nodes: Nodes with multiple deltas
:param node_positions: VCs for the grid nodes
:return: refined sets of nodes with refined deltas
'''
def filter_node_refiner(filter_nodes):
    nodes = [k for k in filter_nodes.keys()]
    updated_nodes = []
    for node in nodes:
        if len(filter_nodes[node]) == 0:
            updated_nodes.append(node)
            del filter_nodes[node]
        else:
            filter_nodes[node] = {key: value for key, value in
            filter_nodes[node].items() if value != []}
            if filter_nodes[node] == {}:
```

```
                updated_nodes.append(node)
                del filter_nodes[node]
    return filter_nodes, updated_nodes

def delta_refiner(filter_nodes, node_positions):
    for node in node_positions:
        if node in filter_nodes:
            if len(filter_nodes[node]) == 1 and len(filter_nodes[node][min
            (filter_nodes[node])]) == 1:
                node_positions[node] = filter_nodes[node][min(filter_nodes
                [node])][0]
                filter_nodes = {k: {ki: [vi for vi in v if vi !=
                node_positions[node]] for ki, v in filter_nodes[k].items()
                } for k, filter_nodes[k] in filter_nodes.items()}
            else:
                node_positions[node] = deepcopy(filter_nodes[node])
    return filter_nodes, node_positions

def data_refiner(filter_nodes, node_positions):
    while True:
        filter_nodes = filter_node_refiner(filter_nodes)[0]
        filter_nodes, node_positions = delta_refiner(filter_nodes,
        node_positions)
        filter_nodes = filter_node_refiner(filter_nodes)[0]
        check_point = []
        for k, v in filter_nodes.items():
            if len(v) == 1:
                for ki, vi in v.items():
                    if len(vi) == 1:
                        check_point.append(k)
                    else:
                        continue
```

```
            else:
                continue
        if not check_point:
            break
    return filter_nodes, node_positions
```

Listing B.20: Node and Delta refinement

```
'''Method to find neighbors from Adjacency Matrix
:param: node: Node of interest
:param ad_mat: Adjacency Matrix of the grid
:return: neighbors of the node of interest
'''
def find_old_neighbors(node, ad_mat):
    old_ne = [None]*4
    for ne, x in enumerate(ad_mat[node - 1]):
        if 0 < x <= 1:
            check = node - (ne + 1)
            if check > 1:
                old_ne[0] = ne + 1
            elif check == 1:
                old_ne[1] = ne + 1
            elif check == -1:
                old_ne[2] = ne + 1
            elif check < -1:
                old_ne[3] = ne + 1
    return old_ne
```

Listing B.21: Neighbors using Adjacency Matrix

```
'''Methods to find neighbors from the deltas of a node
:param: ab_val: Given delta coordinate
:param node_positions: VCs of the grid nodes
:return: neighbors of the node of interest
'''
def find_new_neighbors(ab_val, node_positions):
```

```
    try:
        a, b = ab_val
    except TypeError:
        return None
    neighbors = [(a, b - 1), (a - 1, b), (a + 1, b), (a, b + 1)]
    nei_node_IDs = []
    for nei_node in neighbors:
        if nei_node in node_positions.values():
            nei_node_ID = [ki for ki, vi in node_positions.items() if vi
== nei_node][0]
            nei_node_IDs.append(nei_node_ID)
        else:
            nei_node_IDs.append(None)
    return nei_node_IDs
```

Listing B.22: Neighbors using Deltas

APPENDIX C

A SURVEY OF VIRTUAL COORDINATE SYSTEMS

This appendix contains shortened version of the chapter, **"Virtual Coordinate Systems and Coordinate-Based Operations for IoT"** that appeared as a part of the EAI/Springer Innovations in Communication and Computing book series (EAISICC) [1]. The complete publication with detailed algorithms can be found in the Springer book, **"Performability in Internet of Things"** [2].

[1]G.A.Pendharkar A.P.Jayasumana
Department of Electrical and Computer Engineering, Colorado State University
Fort Collins, CO, USA
email: Gayatri.Pendharkar@Colostate.edu; Anura.Jayasumana@Colostate.edu

F. Al-Turjman (ed.), *Performability in Internet of Things*, EAI/Springer Innovations in Communication and Computing, https://doi.org/10.1007/978-3-319-93557-7_10

C.1 Introduction

Internet of Things (IoT) is expanding into diverse environments including manufacturing, environmental sensing (atmospheric, underground and underwater) [46], smart cities, smart grids and precision agriculture. Wireless Sensor Networks (WSNs) are the key building block in many of such IoT systems, where devices capable of sensing, actuating, communicating and computing, provide the interface to physical plants, environments, terrains and phenomena. Such IoT networks have greatly benefitted and created novel approaches in different fields. Sensor nodes are deployed, for example, on farms to measure micro-climates and soil conditions to improve yield, and for monitoring human presence in houses and offices to reduce the wastage of energy for heating and lighting. Decreasing costs, increasing capabilities, and advances in sensor networking technologies now make it possible to deploy large-scale networks of wireless sensor and actuator nodes that self-organize and adapt to carry out needed functionality robustly and adaptively.

Large-scale WSNs embedded in complex physical spaces depend on scalable and robust algorithms and protocols. Node localization and routing are among essential functions for many such network operations. Node localization refers to identifying the positions of different nodes in the network. In complex 2D and 3D networks, node location information by itself cannot facilitate routing. Furthermore, obtaining location information in the form of physical coordinates is costly and unreliable at best, or is simply infeasible. Coordinate systems also play a vital role in other IoT applications where a network is formed from heterogeneous devices, with each device acting as the fundamental node or unit. This chapter focuses on Virtual Coordinates (VCs) that are more economical to obtain, less susceptible to parametric variations and interferences, and in many cases, provide equal or better routing performance compared to physical coordinates or geographical coordinates.

The outline of this chapter is as follows. In Section C.2, we discuss the physical coordinates, complexities associated with obtaining them and their limitations. Next, we introduce the concept of VCs, and how they serve as an alternative to physical coordinates. Section

C.3 describes a classification for VC schemes. In a network, several methodologies could be used as a foundation for coordinate assignment. Election of arbitrary or selective anchor nodes and structure embedding are examples of techniques that assist in VC selection. In Section C.4, we present the attributes that distinguish the different Virtual Coordinate Systems (VCSs). These attributes also provide a set of metrics to compare the different VCSs and analyze their strengths and deficiencies. Section C.5 describes in detail the different VC techniques. It defines the techniques for coordinate assignment followed by a brief description of representative routing algorithms associated with it. We also provide comparison tables using the parameters from Section C.4 to facilitate portrayal of weaknesses and assets of each technique. Section C.6 summarizes and concludes the chapter.

C.2 Physical Coordinates vs. Virtual Coordinates

The common and familiar methods for dealing with points in a physical space are based on physical coordinates, e.g., (X, Y) in case of 2D and (X, Y, Z) in case of 3D. Thus, the use of physical or geographical coordinates (GC) for IoT networks deployed in 2D and 3D has been the obvious and default choice, and many of the WSN protocols rely on the availability of accurate GCs. The process of obtaining the GCs of nodes is termed physical localization [47], and the routing protocols based on these geographic coordinates are known as Geographic Routing (GR).

Two categories of routing protocols have emerged for large-scale networks of sensors, namely, address based protocols and content-based protocols. The former relies on explicit node addresses, i.e. a set of coordinates defined using a specific algorithm. The latter defines the set of destinations with the use of certain attributes [48]. The content based protocols rely on the use of approaches such as flooding and random walks to reach the destinations and hence have issues such as large overhead, limited scalability and excessive uses of resources, e.g. excessive bandwidth and power consumption. The traditional Internet in comparison, works on the principle of maintaining per node/subnet routing state, which

grows as a function of the network size and number of destinations [48] [49]. With constraints related to memory space per node with IoT subnets such as WSNs, approaches requiring such large amounts of data per node are infeasible.

Two fundamental limitations are faced by GCs in large-scale IoT subnets or WSNs. First is the difficulty and infeasibility of obtaining the physical coordinates. Due to confines in cost per unit and energy budget, it is unfeasible for individual sensors to be Global Positioning System (GPS) enabled. GPS is not available indoors, and even outdoors its resolution may not be sufficient for dense networks. The alternative is to use analog measurements, such as RSSI or Time-of-Arrival (TOA) to estimate distances to other nodes, and thereby obtain node positions [14] [13] [12]. Many such techniques have been proposed in literature [12] [50], but they are not accurate as they are susceptible to phenomena such as noise, fading, multipath and interference, and errors in localization tend to accumulate [12]. Thus, such localization techniques have not been demonstrated in large-scale networks outside laboratory settings, and of course not in harsh and complex environments. The second limitation of geographic coordinates occurs even if one assumes the ability to obtain coordinates with sufficient accuracy, e.g. with manual calibration or using GPS. Specifically, GCs do not help achieve high routability in networks occupying complex physical shapes [51] [52] [53] [54] [55] [32] [56]. It is quite possible for two nodes to be physically very close to each other but separated by a long distance in the communication topology. Two nodes within proximity could be separated by obstacles or voids, e.g. a metal partition or concrete floor, creating a hole in the communication topology; in such cases, the packets will have to follow a long multi-hop path around the obstacle. Such scenarios are extremely common in many 3D IoT application environments, including those inside buildings, factories, and warehouses. Thus, the routing schemes must be able to overcome local minima created by concave geographical voids.

The existing geographic routing algorithms mostly focus on 2D networks [51] [57] [58]. However, these 2D algorithms are not effective nor scalable to 3D environments due to

many causes. The geometric differences between 2D and 3D networks results in significantly increased computational complexity. Harsh and complex environments with 3D obstacles reflect or absorb radio signals rendering the GF and RSSI methods ineffective, Complex geographical topologies deployed on 3D surfaces or 3D volumes contains voids, causing the decoupling of GCs from the communication topology making them ineffective for many network operations [59]. A common element of most GR schemes is Greedy Forwarding (GF), in which a packet is forwarded to a node that is physically closer to the destination [52] [53] [54] [55] [32]. Some of the GR protocols are Nearest Forward Progress and Greedy Forwarding [32] [56]. In the presence of voids or obstacles in the network, these protocols fail due to the inability to bypass complex shaped voids in the network. Overcoming such local minima requires backtracking or some other scheme to navigate around the voids. For example, Greedy Perimeter Stateless Routing (GPSR) algorithm [51] attempts to navigate around the voids by following a certain direction. Such schemes become extremely costly or inefficient when the voids have complex shapes even in 2D deployments. With 3D deployments, such methods fail except in cases of very simple shapes of voids. A few proposed approaches for 3D geographic routing work albeit with some flaws. Greedy Random Greedy routing (GRG), a randomized geographic routing algorithm routes the packets based on a random walk algorithm, but only for network with Unit Ball Graph (UBG) topology [60]. Greedy Hull Greedy (GHG) routing [47] constructs network hulls using planarization for routing; again, it applies to specific network types such as UBG, and GHG, and must deal with complexities due to planarization computation. Furthermore, errors in node positions may lead to unrecoverable routing failures, which significantly degrades the performance of GR protocols [48]. A method to obtain the geographical addresses of an area without using geological information like GPS is addressed in [61]. This technique uses text processing, address pre-processing, and clustering to achieve accurate positions. This approach mostly provides an efficient but complex location discovery method for major e-commerce organizations.

To overcome the challenges associated with measurement and use of physical coordinates in IoT, coordinate frameworks have been developed that do not depend on measurement of geographic location or distance information, yet can be used for functions such as routing and as well as localization [20]. We call such coordinate systems Virtual Coordinate Systems. A VCS defines each node in the sensor network with a coordinate vector of some dimension that may be different from the dimension of the space the network is deployed in. Over the years, different types of VCS have emerged that use different parameters of interest for VC election. A VCS depends on measures such as connectivity, packet loss and topology. Some of the systems are significantly different from the Euclidean coordinate framework while others are Euclidean frameworks where node relationships and connections are preserved but not the actual physical distances. We use the generic term Virtual Coordinate Routing (VCR) to denote routing schemes specifically developed for or based on a VCS.

One of the fundamental techniques for VCS is the graph embedding. The nodes are spread across the network with node connectivity information embedded inside. In this technique, a map or a sub-map of the topology with connectivity information is embedded which is later used by the routing algorithm to route the data efficiently around the network while capturing the voids and obstacles.

Many of the VC assignment techniques use a set of anchor nodes to build the coordinate framework. Anchors are nodes in the network selected randomly or through specific techniques, and the coordinate vector of each node for example may be the shortest hop distances to these anchors. The number of anchors becomes the VCS dimensionality of the network hence making it a parameter of interest. Routing is achieved using these VCs by Greedy Forwarding (GF) or some other technique. VCs of nodes are used for distance evaluation between nodes as well as for node identification (ID) [22]. If the VCS is based on anchors as the reference points, then the anchor selection could significantly affect the routing performance. Selection of an adequate number of anchors with apt placement helps acquire non-identical VCs for each node.

VCs have also been used in the context of Internet and overlay networks - to obtain maps or coordinates that reflect or captures properties such as the delay or other network measurement parameters. With the advent of several VCS, to preserve the network topology, most of the overlay networks need optimum selection of the neighboring nodes and communication paths depending on proximity, network delays, and round-trip time (RTT) [23]. Gathering this sort of information in real-time could lead to a large amount of measurement traffic in the network. To achieve this, Network Coordinate Systems (NCS) have been proposed. This essentially couples network measurement parameters with the VCS. Maximum Likelihood Topology Maps [24] rely on the packet reception probability function to capture the graph topology. Topology preserving maps [25] too retain the graph topology, yet are also homeomorphic to physical layout.

With increased interest on VCS for large-scale sensor and IoT networks, there have been several related developments related to VCS. Several concepts are in place to develop security means to address attacks on VCS. Several attacks that could potentially disrupt the VC formation are identified and techniques for alleviating them are proposed in [62]. A decentralized VCS capable of withstanding any sort of insider attacks is proposed in [63]. Technique to construct a robust coordinate assignment technique with less cost of communication that can sustain malevolent attacks is presented in [64]. These methods use spatial and temporal correlations for statistical analysis of real time data sets. A game theory based model to detect the best attacks and defense strategies are presented in [65]. A self-structuring algorithm that allows each node in a network to identify its position and all the nodes to collaboratively impose a geometric structure to the network is presented in [26]. The distributed algorithm runs on every entity or node without providing any prior knowledge of geographical location. With IoT expected to connect a massive number of nodes in near future, there is a significant need to have sophisticated searching criterions; such approaches may also be VCS based as demonstrated in [27]. The method suggests the use of VCS in finding network statistics like delays, latencies, etc. using a decentralized approach. The

real-time traveling path tracking algorithm for smart vehicles with encoders installed on the left and right side of the wheels to capture the rolling distance of the vehicles [28] relies on a VCS framework. The VCS in this case is fixed on the ground with factors such as the vehicle position and heading angle accounting for the experimental results of the path. The techniques give very accurate results despite obstacles, fog, rain, etc.

Several interesting schemes for routing and related operations using VCS have also appeared in recent literature. A method to use Greedy Routing on a Virtual Raw Anchor Coordinate (VRAC) System is considered in [29]. The VRAC coordinate computation involves measurement of roughly three raw node distances to be used as coordinates. Given that a saturated graph or network exists, greedy routing provides guaranteed packet delivery using VRAC system. Physical coordinate computation is a costly and complex technique for large scale networks. Technique presented in [30] for deriving topology maps of the networks from the anchor based VCs does not require a complete VC set. Using the theory of low-rank matrix completion, the topology maps are extracted for 2-D and 3-D networks using small subsets of VCs. A distributed protocol viz. Hexagonal Virtual Coordinate (HVC) to construct a VCS is presented in [31]. Using this HVC information, a source node can find an auxiliary routing path to the destination. This algorithm provides suitably placed landmarks and unique VCs throughout the network irrespective of any voids.

The above techniques are examples of VCS based or relate methods that make use of diverse parameters to devise coordinate schemes and thus network algorithms such as routing. VCS based routing possesses certain advantages over traditional routing techniques such as substantially high routing capability without relying on location information, consistent performance regardless of presence of voids and no localization errors. They may face problems such as identical coordinates and local minima due to lost directionality [22] which can be resolved by modifying the VC assignment algorithms. Identical coordinates arise if sufficient number of anchors are not deployed, and local minima in these coordinate spaces appear as virtual voids in the network. In this chapter, we will review different VCSs, together with

their properties and the corresponding routing techniques.

C.3 Classification of Virtual Coordinate Systems

As mentioned in the previous section, Geographic Routing (GR) uses geographical coordinates while Virtual Coordinate Routing relies on some VCS. Former depends on the physical distance while latter depends on some distance measure defined in the corresponding coordinate space. VCS approach relies exclusively on the relative distances (e.g., hop count) among nodes in the network. The general idea is to define a VCS and use it to induce a routing protocol based on the proposed VCs.

An anchor based VCS overlays VCs on the nodes of a network based on their network distance from some fixed reference points (anchors or landmarks). The coordinates are computed during an initialization phase. From then on, the VCs serve in place of the geographic location for the purposes of network operations such as packet forwarding. As a VCS does not require precise location information or distance measurements, it is not sensitive to localization errors.

We classify the VCS into four categories as below,

C.3.1 Virtual Coordinate Systems embedding a graph/tree topology

This technique as the name suggests embeds a graph in the network; the graph may be a tree (say) or a topology that is more complex. Based on that topology, the nodes in the network are spread relative to each other with connectivity information as a part of the embedding. Once the structure is established and the connectivity information is known, a routing algorithm may be developed to route across the network acquiring maximum efficiency and avoiding failures.

Examples: *Gradient landmark based VCS (GL-VCS)* [66], *Medial Axis Protocol for VCS (MAP-VCS)* [67], *Graph Embedding for VCS (GEM-VCS)* [21], and *Hyperbolic Embedding in Dynamic Graphs for VCS (HEDG-VCS)* [68].

C.3.2 Virtual Coordinate Systems based on Hop Distances to Anchors

The second approach is the most frequently used approach to establish a VCS. Each node here is first characterized using its hop distance to a specific set of nodes called anchors. This information may be used directly as a set of coordinates, or processed to extract a coordinate system with more desirable properties. Due to its sheer simplicity and effectiveness of the results, this category provides effective and flexible algorithms for VC assignment and routing.

Examples: *Anchor-Based Virtual Coordinate System* [48], *Axis-Based Virtual Coordinate Assignment Protocol* (ABVCap-VCS) [69], and *Directional Virtual Coordinate System* (DVCS) [22].

C.3.3 Topological Coordinate Systems

This approach talks about the techniques that help extract the topology of the network using node connectivity information and a few other parameters. It helps retrieve the informational graph that helps understand the network map without using any physical distance information.

Example: *Topology Preserving Maps (TPMs)* [30].

C.3.4 Virtual Coordinate Systems using Network Measurement Parameters

This technique listed avails of the network measurement parameters such as delays, RTT or hop distances to configure a VCS that captures specific underlying network properties. Examples include coordinate systems motivated by the need to estimate delays without performing direct delay measurements [70], hence reducing the consumption of network resources considerably. It models the Internet as a geometric space, depicting the position of all the present nodes in the Internet by a coordinate in the space. These techniques attempt to retain the physical topology of the network, the connectivity, shape, delay or some other property.

Examples: *Vivaldi - A decentralized Network Coordinate System* [70], and Maximum Likelihood Topology Maps for Wireless Sensor Networks Using an Automated Robot [24].

In the following sections we will discuss examples from these three classes of VCs. Prior to that we present parameters that are useful for comparison of the different VCS.

C.4 Attributes of Virtual Coordinate Systems

The purpose of a VCS, as stated earlier, is to serve one or more purposes related to networking a large set of nodes. A VCS for example, often acts as a proxy for node locations for efficient routing or topology control. A good VCS based on a parameter such as delay or packet loss may allow estimation of such parameters with reduced network measurements, i.e., with minimal cost and effort. Different systems have their own algorithms for assignment of coordinates to the nodes in the network. Such attributes can also be used to compare different VCSs, and to select the proper VCS for a given criterion.

C.4.1 Use of Anchors

Anchors correspond to a set of nodes in the network that act as intermediaries to the other nodes in the network for calculating the VCs. Basically, the number of anchors determines the coordinate dimension for the nodes in the network. Intuitively, the higher the number of anchors, the higher the cost of generating coordinates and the more accurate the node position in the corresponding space. In fact, if all the nodes are anchors, the coordinate system corresponds to the (hop) distance matrix of the graph.

C.4.2 Efficiency of routing/measurements

The primary purpose of a VCS is to serve some network related function(s) such as routing or delay estimation. Thus, the efficacy of the coordinate set to meet the stated purpose is of importance. The efficiency may be quantified by performing routing or the appropriate network measurement scheme. This parameter indicates the precision of the coordinates that are assigned to the nodes in the network.

C.4.3 Susceptibility to local minima issue

Even with an ideal implementation of the algorithm of interest (e.g., routing) for which the VCS is targeted, there could be cases that cause the algorithm to fail. These anomalies impair the desired functionalities i.e., sensing and communication. Examples of issues with VCS include the following: identical coordinate assignment, local minima, formation of logical voids or holes, etc. which would lead to geographically correlated problem areas such as coverage holes and routing voids.

C.4.4 Ability to deal with node failures and changing topologies

In an IoT or a WSN, a node may fail at random points in space and time, or new nodes may be added to the network. As such events change the network topology and connectivity, they may cause disruptive changes to the coordinate system or the resulting algorithms. This attribute aims at capturing whether the VCS is susceptible to such changes, and if so the ease or difficulty with which coordinate system may be restored following such an event. A robust coordinate system will maintain performance within a narrow margin even when such events occur in the network.

C.4.5 Ability to capture the network shape and voids

The node deployment could be of any shape and contain voids of different shapes disrupting communication among the nodes. Ability to capture such topology properties is an important aspect of a coordinate system. Hence, the efficacy of the coordinate system to retain these topological assets is vital.

C.4.6 Applicability to 3-D networks

Node placement of a WSN in the physical space determines the kind of network it is. A planar network with nodes along two axes is a 2-D network. If the surface on which the nodes are placed is not planar, we may call it a 3-D surface network. A 3-D volume network refers to a network deployed in a three-dimensional volume. We use the term 3-D networks

to refer to 3-D surface networks, 3-D volume networks, as well as networks containing both 3-D surfaces and 3-D volumes. It is also important to note that some of the VC techniques have been extended to networks with no associated physical dimensionality such as social networks [30]. Ability of the protocol to implement the VCS in these sort of networks increases the number of application areas as well.

C.4.7 Distributed Computability of VCs

In a WSN, the computation can occur at every fundamental unit, i.e. at the individual nodes or it could occur at a centralized unit, i.e. a centralized server. These are two different types of methods for the WSNs. Consider for example a WSN consisting of many nodes sensing the environmental conditions. In the centralized approach, these nodes send the sensed data to a centralized server known as the Base Station (BS). Due to energy constraints per node, centralized approach proves efficient in those terms. Here, all the nodes are grouped into clusters and then, one representative node is assigned as the cluster head (CH). This node collects all the data within the cluster and sends the data to the BS. Now, only the CH nodes are required to perform long distance transmission hence saving the energy consumption for the other nodes [71]. On the other hand, in the distributed approach, the computation is autonomous; down to the single fundamental unit of the network. The distributed approach takes into consideration the battery restraints per node and the density of the WSN. Here the computations occur based on communication amongst the neighboring nodes. Distributed approach is more desirable than the centralized one due to several reasons. In centralized computing, if the BS fails, the entire network may fail. Additionally, in case of individual node failures in the network, the recovery or repair must be done at the respective central node unlike with distributed algorithms, where the node recovery can be done at its own level. All the VCS discussed in this paper are based on a distributed computing approach. The selection of these cluster heads or landmarks for centralized network is based on a landmark selection algorithm. The set of anchors can be predetermined [72] [73] [74]

or randomly selected [75] [76]. Distributed VCS are independent of explicitly designated infrastructure components, requiring any node in the system to act as a reference node. Examples of such systems include PIC [76], Vivaldi [70], and PCoord [71] [72].

C.4.8 Directionality

VCS are an efficient way of characterizing the node locations thus replacing the geographical coordinate assignment approach. VCS offers a lot of attractive properties such as high routability, consistent performance in presence of physical voids in the network and efficient connection information embedded in the VCs. In some VCS, when there is a mapping of the physical domain to a virtual domain, coordinates become insensitive to directional information. Many deficiencies associated with VCS are due to the missing directionality information and the lack of knowledge of the physical layout. Some of the VCS techniques discussed in this paper can capture or extract directionality information while others do not.

C.4.9 Applicability to WSNs

Certain VCS are not useful for resource limited networks such as WSNs or IoT subnets. There are different performance metrics of interest such as routing performance, efficacy of latency estimation, and boundary detectability, which are important in different contexts. Although our focus is on IoT subnets and WSNs, coordinate systems are also of interest for other applications, e.g. Coordinate spaces that captures latency information among devices.

C.5 Virtual Coordinate Systems

C.5.1 Virtual Coordinate Systems based on an Embedded Graph/Tree Topology

In this section, we will consider several VCS which are based on discovery of the global topological structure of the network, e.g., by embedding of a graph structure such as a tree in the topology. Such a structure may then be used to characterize positions of the nodes and for routing.

C.5.1.1 Gradient Landmark based VCS (GL-VCS)

GLIDER [66] is a novel technique which only uses the node connectivity information and a few selected landmark (anchor) nodes to achieve distinctive node coordinates. This approach is divided into two phases - the global planning phase and a local greedy routing phase. The global pre-processing step helps in mapping the topology of the network using the node connectivity to account for obstacles or holes in the sensor field. Following this phase, the network is partitioned into tiles with each tile containing a subset of network nodes. These tiles are expected to have a trivial topology and simple greedy forwarding methods using local VCs that help achieve good routability.

Consider a communication graph G=(N,E) where, N is the set of sensor nodes and E is the set of unweighted edges. The graph (hop) distance between two nodes is the shortest hop count (number of edges) between them. For a packet traversing from a source node to the destination node, the successor to the source node is always the node which reduces the hop count to the destination node. An auxiliary atlas M(G) is constructed which is shared with every node. It helps achieve awareness about the global topology of the network and connectivity information by partitioning the nodes into tiles and mining the adjacency relations between these tiles. Each of the partitions or tiles is uniquely identified by a landmark or anchor node. It is critical to make an equitable selection of landmark nodes to achieve best results for this algorithm.

C.5.1.2 Medial Axis Protocol for VCS (MAP-VCS)

Medial Axis Protocol (MAP) is a coordinate assignment and routing protocol that runs without any geographic information and performs routing and load balancing efficiently. MAP constructs the medial axis of the network field by selecting the set of nodes, each of which has at least two closest nodes on the boundary. The routing algorithm uses these coordinates to locally route the packets in the network. Many of the techniques discussed rely on the optimum selection of landmark nodes that are used to compute the local landmark

coordinates of the nodes in the system. However, landmark selection is a complex problem which doesn't have a conventional method. MAP ensures the retention of the geometrical and topological features of the network working as the backbone scheme. It is a protocol serving many applications like robot path planning [77] and surface reconstruction [78] [79]. Medial axis for MAP uses only the connectivity information. This can be represented by a graph whose size is directly proportional to the size of the sensor network.

MAP works without using any location information and uses only the graph connectivity information. The protocol is extremely light weight and compact. The medial axis is represented by a graph whose size is directly proportional to the size, geometry and complexity of the sensor network. It requires no maintenance whatsoever once constructed. Since the medial axis serves as a skeleton, MAP has a good coverage throughout the network. It is also robust to variations in the network model following the steps of naming and routing in both discrete and continuous domains.

C.5.1.3 Graph Embedding for VCS (GEM-VCS)

With increasing size of WSNs, scalability is one of the main factors to be taken care of. With a larger network, the number of measurements increase, causing a humungous amount of data across the network. Also, each sensor node has limited memory, storage, communication range, battery power and computational ability. Hence, it is important to utilize resources per node efficiently. There are several techniques to retrieve the data sensed by nodes – local storage, external storage and the data-centric approach. The data-centric approach is the most energy efficient approach which relies on proficient routing mechanism. The Graph Embedding (GEM) is a routing technique for a sensor network with data-centric storage and information processing. GEM is devoid of any geographical information and gives decent results even in presence of physical voids.

GEM is a basically a set-up that is a graph with labelled nodes, embedded in the original sensor network topology in a distributed and efficient manner. Many of the existing overlay

protocols for routing are not suitable for sensor networks as each hop in the overlay could be several hops in actual network. Hence, it is crucial to have an underlying system for routing. GEM works in such a way that every node is aware of its neighboring nodes through the labels assigned to them. These nodes can perform routing by exploiting these labels. Additionally, the data names can be mapped to the labels to use data-centric storage. This approach is based on constructing a Virtual Polar Coordinate Space (VPCS) without any physical placement information of the topology. There are two techniques developed to embed this virtual apace with the network topology – the first one requires for the nodes to find out distances between them and their neighbors unlike the second one. Virtual Polar Coordinate Routing (VPCR) is a routing algorithm that uses the VPCS.

C.5.1.4 Hyperbolic Embedding in Dynamic Graphs for VCS (HEDG-VCS)

This is a routing and embedding technique which allows for external addition of nodes after the network has been formed without disturbing the existing network. To attain this, a simple routing algorithm called as Gravity-Pressure (GP) routing is introduced. Given that a path exists in the network between any two nodes, this method guarantees successful routing even when a few nodes or links between the nodes are removed after the final graph is formed. This method focuses on – constructing a graph with greedy embedding which allows the addition of a random number of nodes to the network without making any changes to the already formed graph. The second step follows with the greedy routing algorithm which works even in the unexpected failures or downtimes of nodes or links in the graph.

C.5.2 Virtual Coordinate Systems based on Hop Distances to Anchors

This section describes VCS that rely on the hop distances to a subset of nodes. They may be generated in a distributed manner, and therefore are especially useful for large-scale networks where obtaining all the information centrally and distributing the coordinates back to the networks are not trivial tasks.

Table C.1: Comparison of virtual coordinate systems embedding a graph/tree topology (category A)

Parameters	VCS technique			
	GL-VCS	MAP-VCS	GEM-VCS	HEDG-VCS
Use of anchors	✓	-	-	-
Efficient Routing	✓	✓	✓	✓
Asserting the local minima	✓	-	✓	✓
Ability to deal with node failures and changing topologies	-	-	✓	✓
Ability to capture the network shape and voids	-	-	✓	-
Use in 3D networks	-	-	-	-
Distributed computation of VCs	✓	✓	✓	✓
Directionality	-	-	-	✓
WSN applicability	✓	✓	✓	✓

C.5.2.1 Anchor-based Virtual Coordinates

Anchor-based coordinate framework provides an efficient addressing mechanism for self-organization and routing without the need for physical location information. In anchor-based VC schemes, few of the nodes in the network are marked as landmarks or anchors. Once the coordinates are evaluated, each node in the network is identified by a vector, called its virtual coordinates or logical coordinates, consisting of the hop counts to each of the anchors. Figure C.1 illustrates the anchor-based virtual coordinates for a network where the four corner nodes are selected as anchors. This coordinate system is the basis for routing schemes such as LCR-VCS [48] and CSR [80], as well as several derivative coordinate schemes and routing methods [81].

C.5.2.2 Axis-based Virtual Coordinate Assignment Protocol (ABVCap)

ABVCap is another approach based on assignment of VCs in WSNs without the need for geographical coordinates, generation of which requires GPS or distance measurement. It is a VC Assignment technique that provides packet delivery across a network. ABVCap is based on the VC Assignment Protocol with quite a few improvisations. Each node in the WSN is static or quasi-static and has a unique ID and same transmission range (default = 1.5) [69]. It assigns at least one 5-tuple VC (u.lo, u.la, u.rp, u.up, u.dn) per node in the network by following assigned steps. A node in the network may have more than one 5-tuple VC. Such nodes are perceived as multiple virtual nodes at one location. Once the VC Assignment is

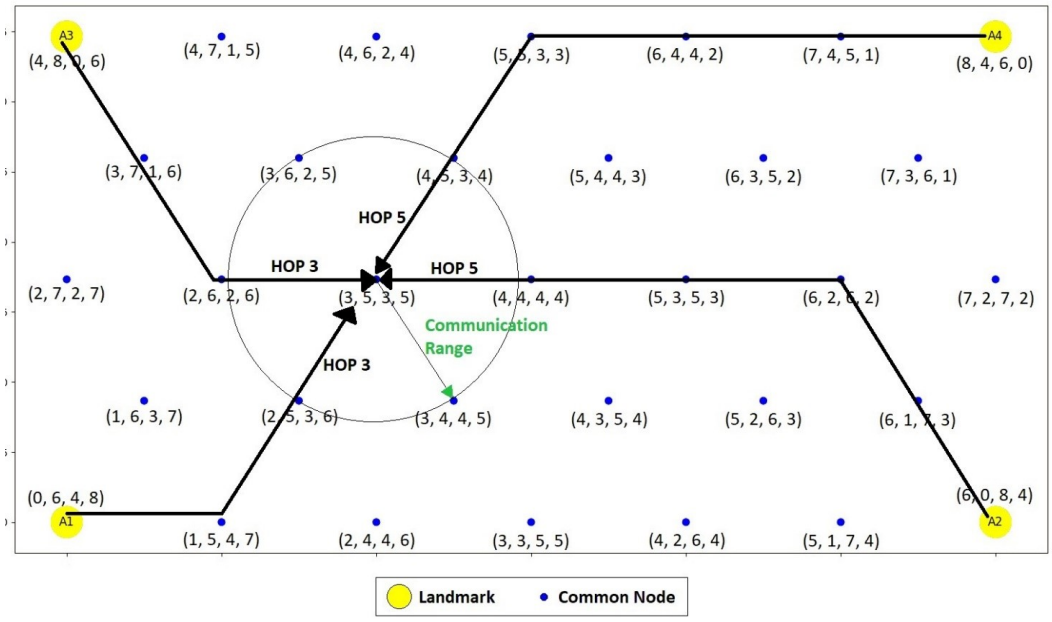


Figure C.1: The logical coordinate framework for LCR-VCS for a rectangular network with triangular grid placement

complete, the routing protocol follows. It consists of two phases – Choosing a VC among multiple VCs for the node, and routing the received packet based on that VC. Figure C.2 shows the ABVCap protocol.

C.5.2.3 Directional Virtual Coordinate Systems (DVCS)

With new emerging techniques for assignment of VCs in a system, there have been several advances to overcome the limitations in the existing techniques. Directional Virtual Coordinate System (DVCS) is a systematic approach designed to eliminate the problem of lost directionality in the conventional anchor-based VCS. DVCS starts with the anchor based VCs for all the nodes across the network. It uses a transformation that combines two anchor-based coordinates at a time to regain the lost directional information, DVCS help mitigate the logical voids associated with anchor-based VCS by providing more geographic-like set of coordinates. This coordinate assignment technique also allows for novel routing strictly with the help of deterministic algorithms for constrained tree network. DVCR significantly outperforms existing VCS routing schemes Convex Subspace Routing (CSR) [80] and Logical

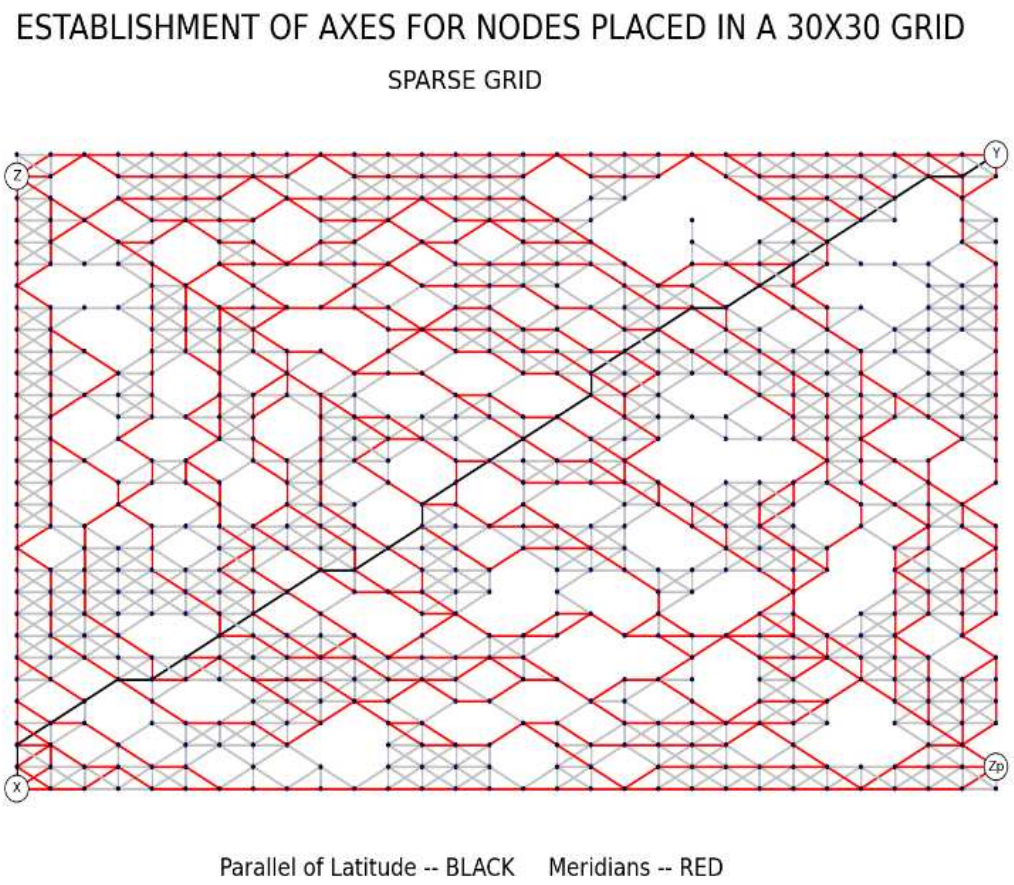


Figure C.2: Establishment of Axes - Parallel of Latitude and Meridians

Coordinate Routing (LCR) [48], while achieving a performance like or better than geographical routing scheme Greedy Perimeter Stateless Routing (GPSR), but without the need for node location information [22].

DVCS proposes novel technique for transformation of traditional VCs to directional VCS. Properties of DVCS are discussed with assignment of coordinates in a constrained tree network. Also, an efficient DVCS routing protocol which uses DVCS coordinate assignment is elucidated. The routing protocol is compared with other protocols like with CSR, LCR and geographical routing scheme called Greedy Perimeter Stateless Routing (GPSR) [51]. DVCR outperforms, CSR and LCR with a noticeable value achieving similar performance as GPSR.

C.5.3 Topology Preserving Maps

Table C.2: Comparison table for decentralized virtual coordinate systems (category B)

Parameters	VCS technique		
	LCR-VCS	ABVCap	DVCS
Use of anchors	✓	✓	✓
Efficient Routing	✓	✓	✓
Asserting the local minima	✓	✓	✓
Ability to deal with node failures and changing topologies	-	-	-
Ability to capture the network shape and voids	-	✓	✓
Distributed computation of VCs	✓	✓	✓
Directionality	-	-	✓
WSN applicability	✓	✓	✓

C.5.3.1 Topology Preserving Maps – Extracting Layout Maps of Wireless Sensor Networks from Virtual

The elementary anchor-based VCS characterizes each node with a coordinate vector consisting of distances to each of the anchor nodes. In the process, the layout information of the WSN such as physical voids, obstacles, shape, and even relative physical positions of sensor nodes with respect to (x,y) directions are lost. TPM technique uses the Singular Value Decomposition (SVD) scheme to recover the network layout in 2D and 3D network surfaces or volumes by isolating and removing the radial component that dominates the VCs. The topological coordinates (TCs) are computed using the coordinates of a subset of nodes. Topology preservation error (E_{TP}), defined to capture the among and degree of node flips, is used to evaluate 2-D TPMs. The defined method extracts TPMs with less than 2% error. Topology coordinates provide an economical and efficient alternative to geographical coordinates [30].

This scheme achieves a map that is homomorphic to the physical layout of the network absorbent of the information about node connectivity, physical layout, and physical voids. The topology map itself is not a physical map, but a distorted version of it taking into consideration the connectivity parametric.

C.5.4 Coordinate Systems using Network Properties

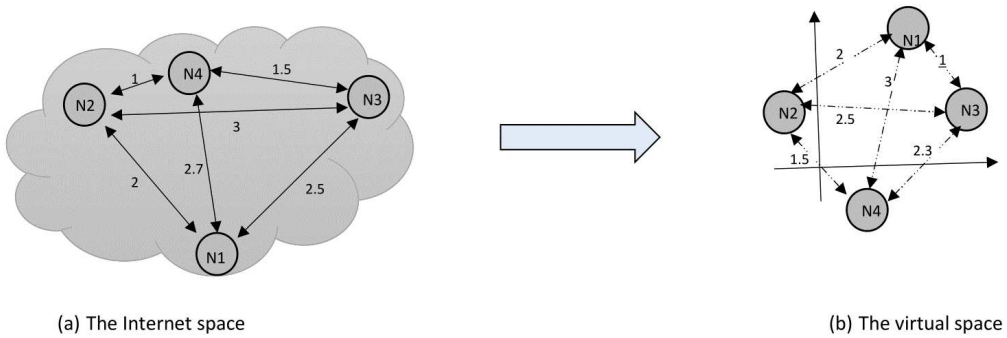


Figure C.3: Mapping of Internet space to the virtual space

C.5.4.1 Vivaldi (Network Co-ordinate System)

Emerging network applications and services are highly intelligible and flexible due to the ability of choosing their own communication paths amongst the available ones. These communication paths are chosen based on certain network measurement parameters such as latency. However, explicit measurements by injecting probes in network would generate a huge amount of measurement traffic in the network making it infeasible to obtain measurements. This is the conventional way of attaining network information for efficiently choosing communication paths in a network. Network measurements benefit several areas such as peer-to-peer file sharing applications, Content distribution systems and Decentralized web caches. To make these measurements viable with minimum effort and cost, NCS have been proposed. With constraints, such as limited resources per node, NCS allows the host to perform measurements with minimal resource consumption [82].

The foundation of an NCS is to model the Internet as any geometric space and characterize the position of all the nodes in the network by coordinate in this space [23] [83]. The distance between any of the nodes could be given as the geometric distance between them. Network systems usually involve construction of overlays and look-ups. With such structures, it proves to be difficult to take measurement as compared to measurement of node distances in a standard planar network. Moreover, injecting probes in the network for these proximity measurements would further lead to complications and unnecessary overheads. Additionally, in the case of ever-changing topologies, it is impractical to take measurements

each time the node changes positions or fails in the network. The Fig. 14 shows modelling the internet as a geometric space and mapping it to a virtual space.

In the real world, the hosts can be present anywhere. These hosts are connected to each other and the distances shows one of the network measurement parameters (say round-trip time). These distances are mapped into the virtual space using an estimate from a conversion distance function in the geometric space.

To achieve a consent between requirements for optimum performance for the overlay networks and scalability constraints imposed by underlying IP networks, NCS for estimating network distances and latencies have been proposed [23].

C.5.4.2 Maximum Likelihood Topology Maps for Wireless Sensor Networks Using an Automated Robot (ML-TM)

Topology maps play a vital role in characterization of a physical network of sensor nodes while maintaining the node connectivity information. It's a non-linear mapping of a physical network to a topology map. Maximum Likelihood-Topology Maps (ML-TM) is a novel concept that creates a topology map using a packet reception probability function, which is sensitive to the distance between nodes [24]. This method retains the physical shape of the network more accurately than other topology maps in comparison. It supersedes many existing range-based and range-free localization scheme to determine node addresses in terms of cost.

This technique proposes to generate a map of the network with the use of a mobile robot. The robot traverses through network using a defined geometric path and hence, finding the Maximum Likelihood-Topology Coordinates using a binary matrix. Whilst moving through the deployed network, the robot gathers a binary matrix based on the packets received from the nodes from different locations. Using that information, the topology coordinates are calculated by the binary matrix and a packet receiving probability function which is sensitive to the distance. It overcomes the flaws of RSSI algorithms [14] which extract the distances from received power, hence comes across significant errors due to RF communication effects.

The proposed topology map preserves the dimensions and shapes of features such as physical voids and network boundaries. It outperforms the RSSI geographical localization and hop based topology maps.

Table C.3: Comparison table for virtual coordinate systems using network measurement parameters (category C + category D)

Parameters	VCS technique		
	VIVALDI	ML-TM	TPMs
Use of anchors	-	-	✓
Efficient Routing	✓	✓	✓
Asserting the local minima	-	-	✓
Ability to deal with node failures and changing topologies	-	✓	✓
Ability to capture the network shape and voids	✓	✓	✓
Use in 3D networks	✓	-	✓
Distributed computation of VCs	✓	✓	✓
Directionality	-	✓	-
WSN applicability	✓	✓	✓

C.6 Conclusion

A Virtual Coordinate System provides an attractive and an economical method to characterize the location of nodes in a network for networking functions such as routing, placement and topology control. A VCS does not rely on geographical information such as GPS coordinates or distance measurements, and thus can be useful in many harsh and complex environments. This chapter surveyed three categories of VC assignment techniques. They were compared with respect to parameters such as the level of computation involved, presence of directional information in the resulting coordinates, and the applicability to sensor and IoT networks. There has been significant research to localize nodes in the geographic domain as it is the familiar and obvious choice. However, it is important to note that the performance of geographic coordinate systems for operations such as routing deteriorates in the presence of concave voids. In fact, overcoming local minima in geographic domain purely based on node locations is highly ineffective with 3D-volume and 3D-surface networks of complex shapes. Such networks can be expected to be very common with emerging

IoT applications. Connectivity based VCSs have shown to be much more effective in such cases compared to geographical coordinates. While many of the V are based on connectivity information, others rely on parameters such as packet loss and path delay to determine coordinates.