BOSTON UNIVERSITY
COLLEGE OF ENGINEERING

Dissertation

A TASK BASED APPROACH FOR MODELING DISTRIBUTED APPLICATIONS ON MOBILE AD HOC NETWORKS

by

PRITHWISH BASU

B.Tech., Indian Institute of Technology, New Delhi, 1996
M.S., Boston University, 1999

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
2003

Dedication

To Mimi and Babiji

For their love, sacrifice, constant encouragement and support

and

To Paul Erdös

For his exemplary creativity and dedication to research

Acknowledgments

Many people have remarked that writing this portion of the dissertation is more difficult than writing the thesis itself. Indeed realizing that to be true I embark on this tough task which is delightful in its own way.

First and foremost, I thank my advisor, Prof. Thomas D.C. Little, without whose inspiration, encouragement and support, this dissertation would not have come to fruition. During my stay of over six years in the Multimedia Communications Laboratory, Prof. Little has always been there as a constant source of ideas, guidance, encouragement, and friendship. He is the one from whom I learned how to think critically about any research problem. I greatly admire his inimitable knack of thinking out-of-the-box on almost any technical topic of discussion while keeping a probing eye on detail. I am still trying to learn that art. Prof. Little's futuristic visions of the applications of our research have really shaped the way I think about wireless mobile computing. I will always remember the lab meetings on Tuesdays which were very productive and never dull thanks to his unconventional insights into the problem domain and his focused approach towards steering the research goals of the lab. His mode of advising was perfectly suited to my style of doing research – enough freedom to let me develop my own ideas and timely advice to check my thought processes from leaving the domain of pragmatism.

Prof. Little's impact on my education was not limited to shaping my ability to think critically on research problems. My technical writing and presentation skills improved dramatically over the years following his valuable tips and advice. He always told me that a significant outcome of the process of earning a doctorate is becoming an "independent researcher." I can claim to be one now thanks to his invaluable guidance and the confidence he had in my capabilities. These words are inadequate to express my gratitude towards him.

I benefited from the advice and guidance of many other faculty members at Boston University. In particular, Professors John Byers and Jeffrey Carruthers and were especially

generous with their time, support and encouragement, of which this dissertation has been the primary beneficiary. Prof. Avresky was a big source of encouragement and guidance during the initial phases of my dissertation research. In general, the entire faculty and staff of the Electrical and Computer Engineering Department and some in the Manufacturing Engineering Department (notably Prof. Yannis Paschalidis) have been very supportive and helpful, and I remain grateful to all of them for making my education at BU thoroughly enjoyable.

I owe much gratitude to Dr. Jason Redi, my supervisor during my summer internship at BBN Technologies and later a thesis reader. Many discussions with him have helped me gain significant knowledge in the practical aspects of the field of mobile ad hoc networking.

I am deeply indebted to my colleagues at MCL, particularly Rajesh Krishnan during the initial years and Ke Wang afterward. Many a times, interesting informal discussions with them unfolded into ideas for subsequent publications. I really enjoyed working with them during the course of my stay at BU. Other colleagues whom I have enjoyed collaborating with at MCL are Ashok Narayanan, Gulrukh Ahanger, Leslie Kuczynski, Naved Khan, Karthikeyan Srinivasan, and Salma Abu Ayyash. Because of them my life in the MCL was never a bit boring. Besides specific advice and support, my colleagues helped me maintain a keen sense of humor and equilibrium through the never-ending process of earning a doctorate degree.

My apartment mate Arnab Majumdar deserves special credit for keeping me interested in many theoretical problems in mobile ad hoc networks some of which I am still grappling with. Discussions with a physicist at wee hours in the morning with Bob Dylan or Tom Waits playing in the background definitely helps one get a fresh perspective. The Senguptas of New Bedford, Palash Banerjee, Ekta Aggarwal, Shantanu Desai, Arindam Mandal, Shameek Gupta, Reetabrata Mookherjee, Pavi Gupta, Vivek Mittal, T. Girish Pai, Nitin Thaper, Rahul Bhotika, Pratul Agarwal, and Arvind Gopalan are a few friends that have borne the burden of chatting with a talkative, perennial Ph.D. candidate to keep

him in high spirits. I will also never forget the special encouragement that I periodically received from Arijit Sarcar, my friend and mentor during my early days at BU. Many other friends go unmentioned here; I sincerely beg their forgiveness for keeping their anonymity.

This dissertation is based upon work supported by the National Science Foundation under Grant No. ANI-0073843. I thank them for their support. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

Last, I thank my parents, sister, grandparents, and other family members for all that they have given me. This dissertation is a tribute to their love, patience, and support.

A TASK BASED APPROACH FOR MODELING DISTRIBUTED APPLICATIONS ON MOBILE AD HOC NETWORKS

(Order No.)

PRITHWISH BASU

Boston University, College of Engineering, 2003

Major Professor: Thomas D.C. Little, Associate Professor of Electrical and Computer Engineering

ABSTRACT

Mobile ad hoc networks (MANETs) are inherently prone to network partitioning and link failures due to node mobility. Distributed applications executed on such environments must cope with these events. In this dissertation, a novel application execution framework and model called a task graph is presented that enables a class of distributed applications suitable for MANETs. The framework allows the representation of the logical requirements of a distributed application via a set of sub-tasks that are assigned to computing devices with suitable attributes for their execution. Each sub-task is represented as a node in a graph and patterns of data-flow between them induce dependencies among the corresponding nodes, thus resulting in a task graph (TG) representation. Application requirements can be expressed in terms of attributes assigned to nodes and edges of the task graph.

Discovery of resources, required for assignment of sub-tasks of the application to physical devices, is achieved by matching of node and edge attributes of the task graph with corresponding attributes of physical resources in the MANET. This process is referred to as embedding of the task graph into the dynamic MANET topology. The corresponding optimization problem is shown to be computationally hard (NP-complete) with respect to average graph dilation even when nodes in TG possess distinct attributes. For tree task graphs with the same property, an exact optimal algorithm is proposed with polynomial time complexity. Because the optimal solution is impractical, requiring global attribute and topological information, and quickly invalidated by node mobility, we propose an alternative

approach that can be performed efficiently in distributed fashion. The proposed distributed protocol utilizes a greedy algorithm that uses local search for discovering suitable candidate nodes with matching attributes while satisfying the constraints imposed by the task graph structure. The proposed protocol is also capable of recovering from node and route failures inherent in a MANET by scalable mechanisms of disconnection detection, re-instantiation, and TG-patching.

The proposed algorithms are evaluated using a set of metrics proposed for quantifying the performance of MANETs and that of distributed applications executed on them. Simulation results are reported for application scenarios with varying complexity of task graphs, network traffic, and device mobility patterns. These results demonstrate that instantiation of large task graphs occurs reasonably fast even under high mobility scenarios. For example, a 63 node binary tree task graph takes less than 10 seconds to instantiate on average on a reasonably dense MANET consisting of 100 devices that are moving randomly with an average velocity of 10 m/s. Effective throughput is high up to medium degrees of mobility and degrades gracefully with further increase in mobility. Finally, we demonstrate the viability of the approach by the implementation of a proof-of-concept prototype using a network of devices including desktop workstations and handheld computers.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Wireless communication has revolutionized society in the last few years. In particular, cellular telephony has benefited hundreds of millions of people all around the globe serving as an efficient ubiquitous voice communication medium. Recently, two wireless networking technologies have begun to transform the vision of "anytime, anywhere data communication" to reality. These are wireless local area networks (WLAN) and digital packet data transmission technology over cellular infrastructures. While the former brings broadband wireless data communication to localized settings such as offices, homes, airports and campuses; the latter purports to make the same available over a much larger geographical area. The biggest advantage of these wireless technologies is that they offer seamless communication between possibly mobile users.

In spite of all the advantages of the cellular and WLAN models, they expect a complex infrastructure to have been established before the users can communicate. More specifically, cellular data communication needs the installation of communication towers at strategic locations over a large geographical area, and WLANs need wireless base stations or access points for facilitating communication between two mobile users with wireless connectivity. However, if an existing communication infrastructure is expensive or inconvenient to set up or use, or if it is absent, mobile users with wireless connectivity can still commu-

nicate with each other by the formation of a mobile *ad hoc* network (MANET). Nodes in a MANET can act as both hosts and routers since they can generate as well as forward packets, respectively. Since there is no existing communication infrastructure (e.g., a wired or a fixed wireless base station), MANET nodes are expected to act cooperatively to establish the network "on-the-fly" and route data packets, possibly over multiple hops.

Traditional applications of such rapidly deployable network architectures are in battlefield and disaster relief scenarios, one of the first examples being the DARPA Packet Radio Network PRNET [47]. However, considerable interest in MANETs has been shown recently in the commercial sector owing to the miniaturization in size of computing devices, the proliferation in their number, and the increasing demand of people to stay connected all the time. The advent of wireless networking technologies such as IEEE 802.11 [26] and low power technologies such as Bluetooth [17] and Zigbee [72] can propel ad hoc networks to becoming an enabling platform for what is known as *pervasive computing* or *ubiquitous computing* [70].

A MANET is a rapidly deployable, autonomous system of possibly mobile devices which are connected by wireless links to form an arbitrary graph at any instant of time. With the increase in popularity of portable devices and wireless connectivity standards, MANETs are likely to gain popularity in the near future, especially in settings where a networking infrastructure is expensive, cumbersome, or impossible to construct. We can conceive of scenarios in which the environment surrounding us consists of a large number of specialized as well as multipurpose devices, many of them portable, and linked through wireless connections, albeit with fluctuating link availability. Ideally, such pervasive networks can enable a broad range of distributed applications that need exchange of information between multiple devices.

In this dissertation, we propose a set of techniques for modeling and intelligent execution of distributed applications (or tasks) on a network of possibly mobile devices in the vicinity of the user. Our technique takes into account the nature and requirements of

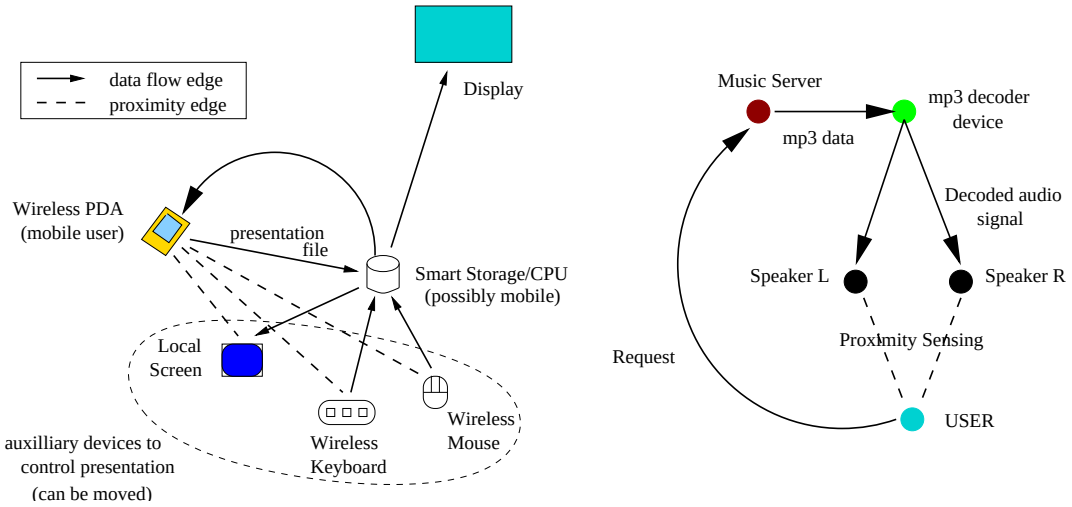
a given application task for making decisions about the set of devices that the application should run on, if there is freedom to do so. We also propose to make the above process adaptive to network partitions and failures which are common in MANETs.

1.1 Problem Description

Significant previous research on MANETs has focused on networking issues such as channel access [48, 15], routing [46, 61, 44, 40] and clustering [28, 35, 62]. In contrast, issues related to application design have received little attention in this context. When a large number of computing devices become equipped with wireless connectivity, and form an ad hoc network, they can offer services to other devices for performing several tasks. In such a situation, since the service providing devices may themselves be mobile, a user cannot rely on one particular device for a certain service since its reachability and availability is not guaranteed. Instead, a user must be prepared to access the required service from any of several devices in the MANET providing similar services, if possible.

We propose and justify a novel scheme for modeling and executing distributed applications on MANETs that rely on services offered by other devices scattered across the network. We introduce the abstraction of a *Task Graph* (TG) for representing higher-level tasks (or applications) that a user may want to perform, in terms of smaller sub-tasks. It is a graph composed of *nodes* and *edges*, where the nodes represent the *classes* of devices/services needed for processing data related to the task while the edges represent necessary associations between different nodes for performing the task. A service class can represent a hardware specific service such as a printer, a photocopier, or a digital picture frame, or it can represent a software service such as a PDF to Postscript format conversion proxy.

When a task is to be executed, specific devices need to be selected at runtime, and then made to communicate with one another according to the specifications of the TG. More specifically, for each class of device in TG, one suitable instance needs to be



(a) Smart Presentation Task (b) Stereo Music Service

Figure 1.1: Smart Office and Home Applications

chosen to take part in task execution. We refer to this process as *embedding* a TG on a MANET. When a participating device becomes unavailable, a new substitute device with similar capabilities needs to be selected to continue the task. Hence, efficient algorithms are needed for achieving dynamic selection of devices (and their re-selection in events of failure) specified in TG while honoring the structure of TG. We propose a distributed approach for solving this problem. We also propose to use the following metrics for measuring the success of our algorithms: average time for instantiation, quality of the instantiation (the technical term we use for this is *dilation*), frequency of disruption of tasks due to mobility or route failures and subsequent re-instantiation, average effective throughput that is achieved after application data transmission begins, and delay experienced by application data units (ADUs).

We now illustrate two pervasive computing applications that are enabled by some of the ideas presented in this dissertation. Imagine a large hall with a number of display screens, projectors, wireless keyboards and pointing devices. Also assume that there are a number of "smart storage" devices which can run popular file-viewing software. These

devices can store a presentation file in their local storage and can render them on a display screen using appropriate software. Devices are untethered and therefore can be moved around in the room freely. The presenter carries only a PDA (with wireless networking capability) that hosts the presentation file. The file (or pieces of it) is transferred from the PDA to a suitable smart storage device depending on the type of presentation software and other factors such as network distance (in hops) and quality of the route in terms of delay, bandwidth and error rate. The presenter needs a pointing device, a keyboard, perhaps, and a local screen to control the presentation, but does not care about which particular devices actually perform the presentation task. Figure 1.1(a) depicts the presentation task with a graph where the vertices stand for categories of devices involved in the task and the edges stand for data flow between devices or other factors like physical proximity to the user.

Assuming that there is no infrastructure support in the hall, a multihop ad-hoc network is formed among several devices therein. In this scenario, a protocol is required to discover the most suitable display screen, smart storage device, keyboard, mouse and local screen with respect to the presenter's location and data requirements. The presenter can move around in the hall either naturally or for displaying multiple items at the same time, and a different set of peripherals may need to be discovered for optimal effect. Note that the nearest smart storage device containing the suitable viewer may not be the best one for the nearest keyboard and the nearest display to communicate with. The discovery process must consider the relationships between vertices in the corresponding task graph. The presentation task can then be achieved seamlessly without bothering the user with configuring the chosen peripheral devices.

Figure 1.1(b) illustrates an example of an application for future smart homes. Imagine a user wearing a proximity sensor embedded on her shirt roaming around in the house while listening to music. When she requests a particular song from a music server (hosted in the house or somewhere in the Internet), the music server streams the digital music file to a suitable digital music player in the house which may be a hardware player or a soft-

ware decoder residing on a PC. The proximity sensor in the user's shirt senses the nearest set of speakers, and the selected decoder device starts streaming the audio to the selected speakers. If the user moves to another room, the music *follows* the user automatically since a new set of devices is dynamically selected, triggered by the proximity sensor.

1.2 Contributions

The following are the key contributions of this dissertation:

- We propose and demonstrate a graph theoretic methodology for the logical representation of distributed applications in terms of its components and the information flow (data-flow) between the components. The methodology loosens the currently tight coupling between network hosts and the services provided by them. We propose distributed protocols that discover efficient mappings between logical application components and actual physical devices that would execute them. This is performed while taking the whole nature of the task into account. We formally model this as a task graph embedding problem.

- We show that both the general as well as a key restricted case of our problem are NP-complete and hence it is important to search for heuristic solutions that are efficient.

- We propose an optimal embedding algorithm for the restricted case where the task graph is a tree, and use this as a basis of a solution for general task graphs. We also propose a greedy solution approach which aims at finding a possibly suboptimal embedding faster. A distributed version of the greedy protocol was used to perform the actual embedding.

- We propose a protocol framework for mobility adaptation since mobility of devices is a common feature of MANETs that can disrupt a running application. Our protocols adapt to network partitions by first detecting the disconnections between relevant

devices and then by rediscovering replacement devices onto which the affected application components are mapped. This feature of the protocol makes it truly self organizing.

- The protocols were validated by extensive simulations and by the development of a proof-of-concept prototype.
- We also describe several novel applications that could be enabled by the techniques proposed in this dissertation.

1.2.1 Significance

Existing distributed applications involve a tight coupling between hosts and the services they provide. In MANETs, such tight coupling may result in sudden application disruptions because of mobility of devices. Since there may be other devices in the vicinity of the user offering similar services that an application desires, the execution of the application can be continued by intelligent selection of devices as proposed by our framework. We believe that a design philosophy based on the loose coupling between a service and the device(s) offering it helps in the development of more robust distributed applications in a mobile environment. Some application scenarios that are enabled by our proposal are listed in Section 1.3.

1.3 Additional Examples of Application Scenarios

In this section we describe a few other application scenarios from diverse domains which motivated this research. This list is by no means exhaustive. None of the applications presented in this section exist today. As wireless networking (ad hoc or not) becomes ubiquitous, we expect applications of the task graph based modeling techniques proposed in this dissertation to be seen in many more domains which are not obvious to us right now.

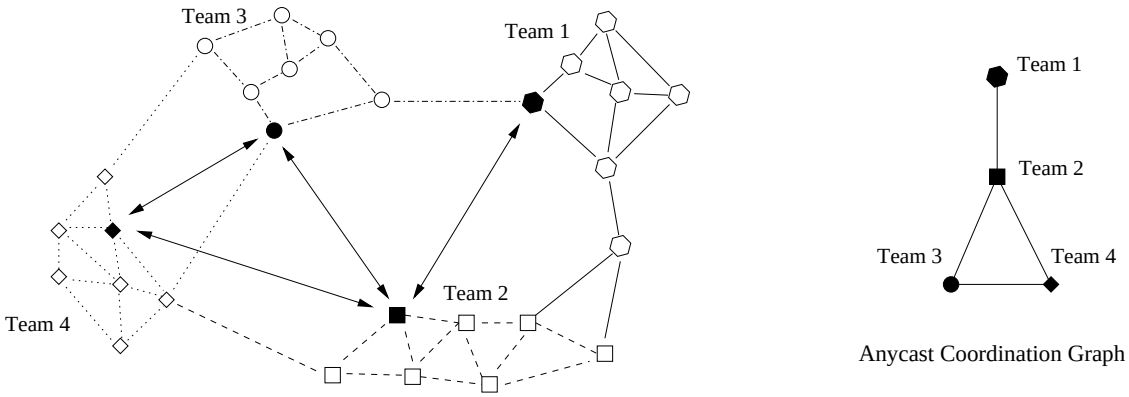


Figure 1.2: Scalable Coordination in Multi-team Field Operations

Scalable Coordination in Multi-team Field Operations Imagine an generic scenario where multiple teams are participating in a certain field activity to achieve a collective goal, with each team having a specific role in the overall endeavor. The various participants can communicate with each other during the activity over a wireless ad hoc network. In general, two types of communication patterns are expected between the participants: (1) intra-team and (2) inter-team. The intra-team communication is likely to be of a multicast[1] nature since there can be significant overlap between the activities performed by the members of a team. However, the inter-team communication is likely to be of point-to-point nature, as a particular team may want to communicate with only a subset of the rest of the teams, and not all of them. Therefore, instead of forming one large multicast tree where all the participants are senders and receivers, one can form a number of smaller multicast groups, each corresponding to a team of workers, connected together by a small *anycast*[2] tree. The exact patterns of the inter-team communication depend upon the specifics of the activity. Figure 1.2 shows a hypothetical schematic of one such application involving four teams. The communication pattern between teams is represented by a so called *anycast communication*

[1]Multicast is a network group communication paradigm [69] in which a subset of nodes in the network communicate with each other; a data packet sent by a node is intended for delivery to all other nodes in the aforementioned subset.

[2]Anycast is a paradigm where a data packet is delivered to exactly one node in a set of nodes all of which can satisfy a given criterion [60].

graph which is shown on the right. The dark shapes denote the participants who are chosen to act as representatives of each team for the purpose of carrying out the inter-team communications. The intra-team communication on the other hand can occur by multicast.

An example of the above class of applications can be in disaster relief which is one of the foremost applications of mobile ad hoc networks. There are various teams of workers involved in such as setting, namely, law enforcement officers, fire fighters, rubble removers, rescue workers, paramedics, and ambulance personnel among others. Each of these groups of people have specialized roles in the rescue operations, and they are expected to be in constant communication with each other through their personal handheld devices. However, usually all rescue workers do not need to communicate with all others. For example, the fire fighters receive data from sensors and communicate with paramedics, who in turn need to communicate with the ambulance personnel only. Hence, just as explained in the generic example, one can establish certain communication patterns between different groups of rescue workers such that only messages that are relevant to each group arrive at their PDAs.

The nodes in the anycast tree are similar to the nodes in a task graph, each belonging to a specialized type defined by their roles in the rescue initiative, e.g. Police, Ambulance, Paramedics etc. An edge in an anycast tree joins one member of a specialized team to a member of another specialized team, if those particular teams need to communicate (e.g., if fire fighters need to communicate with paramedics, one fire fighter's PDA acts as the gateway of communication, and so does one paramedic's PDA). If multicast communication is used, data from one member of any team is delivered to all members of all participating teams. This could result in significant wastage of network bandwidth. In contrast, communication along the edges of the anycast tree could result in more scalable management of network resources since the multicast groups corresponding to individual teams are much smaller in size. Our proposed instantiation algorithms can efficiently establish an anycast tree along which such communication can occur. If the selected anycast nodes are lost due to network

partitions caused by failure or mobility, then replacement nodes can be selected dynamically.

Similar multi-team field operations can be envisaged in battlefield scenarios. Each high-level operation can involve a number of teams from infantry, air force, marines, and strategic command physically spread across a certain geographical area. In a futuristic digital battlefield, these teams are likely to be connected by a wireless ad hoc network [64]. The plan of operation can involve collaboration and coordination among certain pairs of teams. As in the disaster relief example, an anycast tree can be constructed involving a single node from each of the teams and the command and control operations can be communicated along the edges of that anycast tree instead of an all-to-all multicast.

Another burgeoning application area that is gaining interest in academia as well as the military because of advances in mobile ad hoc networking is large scale distributed robotics. Complicated tasks can in principle be performed collaboratively by an army of mobile robots equipped with different types of sensors and actuators just as food is gathered cooperatively by an ant colony. Our ideas described in this section can be useful where there is division of labor among the robots.

Distributed Computing Consider the following pervasive computing application: a journalist is conducting an interview at a location where there is no significant computing or communications infrastructure. Nonetheless, she wants the transcript of the interview to be generated automatically in real time using state-of-the-art speech recognition technology. However, the PDAs or other low power computing devices that she or her crew is carrying are computationally inadequate to process the speech signal and convert it to text in real time. To this end, different parts of the captured speech signal at a PDA can be distributed wirelessly to nearby devices with greater computing power (e.g., powerful laptops with owners willing to spare their CPU cycles), low current load, and required software to perform the speech-to-text conversion. This task may require access to other services such as dictionary search etc. Distributed Speech Recognition technology can be used for

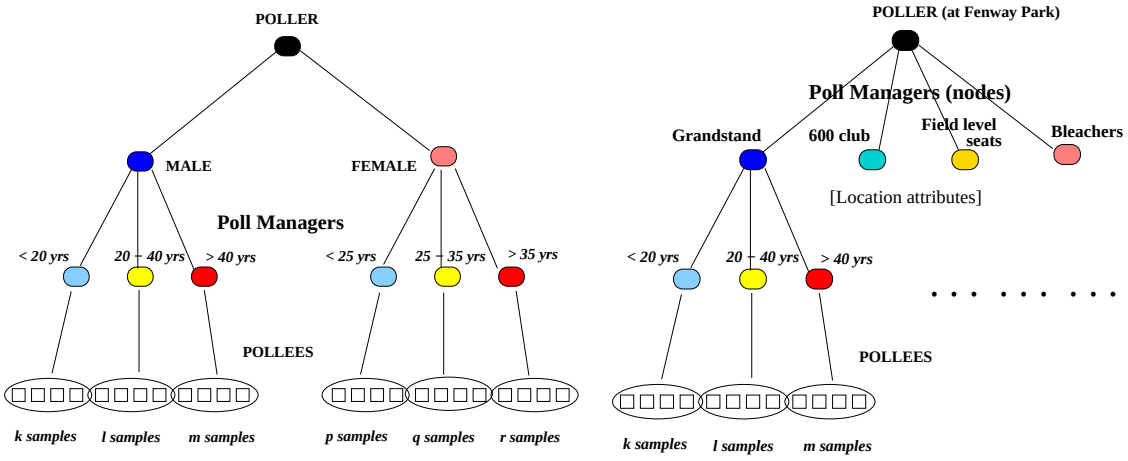


Figure 1.3: Task Graphs for Wireless Polling

the communication between the thin client and the more powerful servers in the network. After the conversion is done, the results can be seamlessly collected from the machines participating in the task, and then collated to generate a complete text transcript on the interviewer's PDA.

Distributed Computational Games Consider a group of people trying to play a game of chess against a grandmaster at a crowded marketplace. After every move made by the grandmaster, the game state is distributed to all *current* players who distribute the search in the game tree intelligently to their computing devices which may collaborate with each other over a MANET and suggest good moves to the players. This is essentially a distributed search problem which can have very large task graphs which obviously change over time.

Wireless Polling Systems Imagine a football stadium with 70,000 people in it watching a game. As Emmitt Smith crosses the goal line and creates a new rushing record, the stadium authorities decide to poll the people in the stadium with a question: "Was Walter Payton a better running back than Emmitt Smith?" Polling can be achieved over the

wireless ad hoc network formed by the PDAs in the stadium.

Now suppose that one wants to conduct a poll in a controlled fashion. Instead of broadcasting the query to *all* PDAs in the stadium and processing all responses, one wants only a fraction of people in the audience to reply as long as people from most profiles are represented proportionally in the poll results. The advantages of doing this are twofold: (i) Less wireless bandwidth will be consumed in the polling process, and (ii) The poll results are likely to represent samples from different sections of the population in a fair and controlled fashion. The extent of fairness and control in the polling process can be defined by the *poller* quantitatively by means of a task graph. A sample task graph depicting a structured poll is shown in Figure 1.3(a). The *POLLER* wants a specified proportion of votes (specified by parameters k, l, m, p, q, r) from males and females in particular age groups as shown in the figure. The simplest way to perform the poll would be as mentioned before: flood the query throughout the MANET and collect the responses. In addition, only $k, l, m, p, q,$ and r responses need to be processed by the *POLLER*. Since this wastes wireless bandwidth, expanding ring search [18] can be used until the requisite amount of responses have been gathered. However, even this suffers from one problem that virtually all *pollees* will respond to the single *POLLER* node which will be swamped with incoming traffic. In fact the nodes within a few wireless hops of the *POLLER* will be busy forwarding/routing the incoming packets towards it.

A task graph based solution can mitigate the above problems by delegating the task of polling to an intermediate layer of nodes which have enough computing resources and are less power constrained in their operation. We call these nodes *Poll Managers*. They conduct the polls based on the set of profiles that they are responsible for and act as *aggregators* of poll results which are processed and then returned back to the POLLER. If the Poll Managers are spatially spread out uniformly across the network (this will be the case for the task graph shown in Figure 1.3(b) which contains location based attributes), it will result in less channel contention and reduce hot spots in the network. Another advantage

of using intermediate poll managers is that they can localize the detection of mobility of a device in the middle of a poll transaction.

Vehicular Networks An interesting application of task based modeling can be seen in the following futuristic scenario that has been adapted from [21]. Bob is driving his car on a congested highway with Alice on the passenger seat. Suddenly Alice falls sick, and Bob has the task of driving her to the nearest medical center immediately. Bob has to find the fastest route to the hospital in heavily congested traffic. The current in-car navigation systems are incapable of solving this problem. Now imagine that Bob has a PDA in his car which is equipped with short range wireless communication capability such that it can communicate with other cars in its vicinity. Since the highway is congested, the cars in the neighborhood of Bob's car have a strong likelihood of being rich in timely information which they have *cached* in the recent past about the traffic conditions. Bob's PDA can attempt to obtain the necessary information about the fastest route from his current location to the nearest medical center by using information and services available in its neighborhood.

When Bob speaks his query into his PDA, it decides that to perform the task, the following services are necessary:

1. A GPS service to determine current location
2. A dynamic traffic information service to give information about the current levels of congestion in the streets ahead: this can be obtained from cars coming from the opposite direction
3. A map service to provide the location coordinates of nearby hospitals
4. A route calculation service to calculate the current fastest route from the current location to any of the nearby hospitals

Now only a subset of the aforementioned services maybe available in Bob's car (none may be available). Hence, Bob's PDA may have to discover suitable instances of these

services in its wireless neighborhood and help Bob discover the fastest route to the nearest hospital immediately. This task can be represented in the form of a task graph where the services are nodes and the data dependencies between the services are edges. Instantiating this task graph in this context would be equivalent to discovering the best instances of these services in Bob's vehicular neighborhood such that the best route is returned to him as soon as possible.

In this section, we have purported to give the reader a full flavor of the application space that can be enabled by the use of approaches proposed in this thesis. We strongly believe that a variety of other imaginative applications involving constrained resource discovery can be enabled as computing becomes more and more pervasive in future.

1.4 Organization of the Dissertation

The remainder of this dissertation is organized as follows: In Chapter 2 we discuss existing techniques for performing service discovery in infrastructure based networks. We also discuss how these techniques fall short of systematically modeling a distributed application logically in terms of its components, as we have done in this dissertation. In Chapter 3 we show how distributed tasks can be modeled with graphs (hence the name *task graphs*) and then formally present the task based resource discovery problem as a graph embedding optimization problem where the target network is a MANET without any regular structure. In Chapter 4 we show NP-completeness proofs of some variants of the proposed problem and give optimal as well as faster heuristic solutions for special cases of the problem. In Chapter 5 we propose a distributed algorithm for task based resource discovery. We also present techniques for adapting to disconnections due to mobility of nodes in the network. In Chapter 6 we present large scale performance evaluation of the proposed algorithms by simulation. The design and implementation of a proof-of-concept prototype that was built on a testbed of mobile wireless devices has been described in Chapter 7. In Chapter 8 we

extend the basic flat task graph model to a hierarchical one, and present algorithms for instantiation of hierarchical task graphs. In Chapter 9 conclusions and directions for future work are presented.

Chapter 2

Background and Related Work

The need for locating services rather than specific hosts or devices in the network has received much attention from the networking research community in recent years. In this chapter, we first describe the state of the art for mobile ad hoc networks and how they can be useful in the application scenarios that we envisage in Chapter 1. We then describe the existing techniques for service discovery in both wired and wireless networks and discuss their limitations toward our goal of achieving distributed application execution on MANETs. We then discuss how task graphs have been traditionally used to model complex parallel and distributed computing applications, and how they can be adapted to serve as a very useful tool in representing the class of applications presented in Chapter 1. We also discuss briefly the related communication paradigms of IP multicast and IP anycast in the context of the Internet as well as MANETs and describe how our work fits into that space. Throughout this chapter, we identify the gaps that exist between these diverse areas in networking, and argue how our ideas help in bridging the gap and opening up a new direction of research.

2.1 Mobile Ad hoc Networks

If an existing communication infrastructure is expensive or inconvenient to use or setup, or if it is absent, mobile users with wireless connectivity can still communicate with each other by the formation of a mobile *ad hoc* network (MANET). Nodes in a MANET can act as both hosts and routers since they can generate as well as forward packets, respectively. Since there is no existing communication infrastructure (e.g., a wired or a fixed wireless base station), nodes in a MANET are expected to act cooperatively to establish the network on-the-fly and route data packets possibly over multiple hops.

MANET nodes are equipped with wireless transmitters and receivers using antennas which can be omni-directional (broadcast), highly directional (point-to-point), steerable, or some combination thereof. At any instant of time, depending on the nodes' positions and their transmitter and receiver coverage patterns, transmission power levels and co-channel interference levels, a wireless connectivity in the form of a random, multihop graph or ad hoc network exists between the nodes. This ad hoc topology may change with time as the nodes move or adjust their transmission and reception parameters. MANETs have the following salient characteristics among others (from [25]):

1. Nodes in a MANET are free to move arbitrarily; thus, the network topology which is typically multihop may change randomly, dynamically and rapidly. The network may consist of both bidirectional and unidirectional links.

2. Bandwidth-constrained, variable capacity links: Wireless links will continue to have significantly lower capacity than their wired counterparts. In addition, the realized throughput of wireless communications after accounting for the effects of multiple access, fading, noise, and interference conditions, etc. is often much less than a radio's maximum transmission rate. A direct consequence of the above fact is that congestion is typically the norm rather than the exception. In other words, it is frequently the case that the aggregate application demand outstrips the network capacity.

3. Some or all of the nodes in a MANET may rely on exhaustible sources of energy such as batteries. For these nodes, an important system design criteria for optimization is energy conservation.

The principal focus of MANET researchers has been in the areas of medium/channel access, routing, and clustering, each of which is briefly summarized below.

Medium Access Control Two fundamentally different types of medium access schemes are used for scheduling data transmissions in MANETs: deterministic (slotted) access and asynchronous (random) access. Time division multiple access (TDMA) is a well known medium access scheme for deterministic bandwidth allocation and quality of service provision in ad hoc networks. According to TDMA, bandwidth can be allocated to the network links using a time schedule well known to all nodes. At every slot of such a schedule, links are activated for transmission such that collisions are avoided at receivers. Determination of the set of slot allocations in a TDMA based network governs the performance. Since system-wide clock synchronization is not always possible to achieve in a distributed ad hoc network setting, Bluetooth uses TDMA locally within piconets which are connected to form larger ad hoc networks called scatternets. Global synchronization of time references between different piconets is hard and is unspecified by the Bluetooth standard.

The most popular asynchronous random access protocols is Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) used by the IEEE 802.11 wireless LAN standard [26]. Since collisions occur at the receiver and not at the transmitter, simple carrier sense does not provide all the information necessary for collision avoidance (e.g., for the hidden terminal problem [48]). CSMA/CA tries to solve the hidden terminal problem by adding an RTS/CTS/DATA/ACK exchange for every packet transmission. The Distributed Coordination Function (DCF) or "ad hoc" mode of IEEE 802.11 allows the formation of a peer-to-peer ad hoc network. Random access MAC protocols cannot provide any bandwidth allocation guarantees like well synchronized TDMA based protocols. However, bandwidth

guarantees are generally harder to achieve in the absence of global slot synchronization.

Routing Protocols Many protocols have been described in the literature for routing over multiple hops in MANETs. They fall into the following categories: (1) proactive or reactive (on-demand), (2) distance vector based or link state based or source route based, and (3) geographic position based.

Notable reactive protocols include Dynamic Source Routing (DSR) [46] and Ad hoc On-demand Distance Vector (AODV) [61]. DSR is a source routing approach that discovers routes by *reactive flooding* followed by subsequent reply messages from the destination node. The packet header maintains the routes from source to destination and is utilized by intermediate *relay* nodes. DSR uses route caching aggressively for better performance. AODV operates on-demand but maintains destination indexed routing tables instead of source routes following a distance-vector approach. An example of a leading proactive protocol is Optimized Link State Routing (OLSR) [44] which advocates maintenance of routing tables by efficient flooding of link-state updates from each node.

Zone Routing Protocol (ZRP) [40] was the first solution that attempts to combine the merits of the proactive and the reactive approaches. It uses a proactive scheme to find routes within each defined *zone* and a reactive scheme to discover routes to a destination in another zone. Some protocols which utilize information about node positions include Location Aided Routing (LAR) [51] and Greedy Perimeter State Routing (GPSR) [49].

Performance of the above protocols depends upon factors such as node mobility and application traffic load, and hence no single protocol is the best for every scenario. While reactive protocols tend to be better under light to moderate loads even under high mobility, their proactive counterparts fare better when there is heavy traffic in the network under high mobility [45].

Scalability with Clustering As the size of a MANET grows, *flat* routing schemes do not scale well in terms of performance. Hence, *hierarchical* organization or "clustering" is beneficial in large MANETs for solving this problem. Several clustering strategies have been proposed in the past and in the recent future for imposing a semi-hierarchical structure upon nodes in a MANET in order to increase scalability in routing, improve bandwidth utilization, and to reduce delays for route queries [28, 35, 52, 62, 5, 12]. An excellent description of cluster based MANETs can be found in [65].

Application level issues in MANETs have not received much attention from the research community since the application space for MANETs has been limited to file sharing/transfers and streaming media. Devices in a MANET can indeed act as service providing entities to other devices in the network. With the availability of a large number of resources in a dense MANET, application and tasks involving multiple resources or services in the network can be performed cooperatively if suitable coordination protocols are running for the execution of the task. In this dissertation we focus on such applications of MANETs and develop a model to logically represent complex distributed applications. Since mobile devices can move away to an unroutable location and partition the network, a central goal of a task coordination protocol would be to detect such a situation and attempt recovery.

2.2 Service Discovery Techniques

Service discovery and location have been recognized as important primitives for ubiquitous computing. Service discovery refers to the process of determining the address of a node offering a particular service in a networked environment. Service location refers to the actual process of accessing the aforementioned service on the discovered host. Internet Engineering Task Force's (IETF) Service Location Protocol (SLP) [38] is a protocol which emphasizes the location of a host with desired capabilities and attributes that can provide a specific named service. SLP describes an IP multicast based approach to locate desired

service-to-host bindings.

Sun Microsystems' Jini [57] is another effort from the industry that attempts to decouple the service from a specific host. Jini also focuses on the automatic downloading of drivers for accessing devices that are not recognized by a client system.

Both of the above are centralized directory based schemes where a service registers itself with a centralized registration server with all its attributes; when a client queries for a service with specific attributes, the system returns candidate hosts that can satisfy the client's needs. MOCA is a variation of Jini without any centralized registry [13]. It is specifically designed for mobile computing devices. Every device running MOCA has a service registry component which only the applications running on the local and surrounding devices can benefit from.

The difference between the approach proposed in this dissertation and the above schemes is the following: what we are proposing is not merely service location or discovery, but a complete framework which enables the execution of complex distributed tasks on MANETs; locating a service is only one aspect of the process of task execution. Moreover, being centralized in nature, most of the schemes described above are unsuitable for use in MANETs. Our technique also allows dynamic, on demand *composition* of complex services from simpler service instances. This is not the focus of the aforementioned service discovery schemes. Hence, our approach operates at a logical layer above service discovery and can co-exist with any of these schemes.

Some researchers have proposed to capture *user-intent* for discovering appropriate devices suitable for the user's tasks. University of Washington's Portolano project [30] advocates the use of *data centric* networking for pervasive computing applications. This emphasizes the fact that *data* is of primary importance to the user and it should be appropriately massaged and routed to her by the pervasive computing infrastructure.

In MIT's Intentional Naming System (INS) [1], the user's intent is abstracted into

collections of attribute-value pairs or *names* that describe what the user application requires. Specific devices in the network that can offer the desired service are selected by special entities in the network called Intentional Name Resolvers (INR) when the latter are queried by the user. The INRs configure themselves into a self-configurable overlay network based on performance metrics and exchange metadata about names and corresponding network locations. Name resolution and message forwarding processes are integrated. The late binding feature of INS is similar in philosophy to the process of *instantiation* of a task graph node in our system (see Chapter 3).

Although similar in certain goals, INS differs from our system in two main aspects. First, it is designed for infrastructure based networks and not MANETs – some nodes in INS that are dedicated as name resolvers and Domain Space Resolvers are expected to be static and only some clients and end services are expected to be mobile. Network topology is not expected to change much dynamically either. Also, mobility in INS is different from that in MANETs – in the former, service mobility causes location attributes of the device providing the corresponding service to change and their goal is to maintain seamless communication with the same device using late binding. In contrast, we are concerned with mobility that causes a service providing device to become unreachable from another device that requires the former's service. Schemes proposed in this dissertation attempt to recover from such situations. Secondly, INS does not attempt to systematically represent or leverage the logical relationships between the components of a distributed task for the discovery of devices. This is our principal focus.

Hodes et al. [41] investigated means of composing services for heterogeneous mobile clients. Their work primarily focuses on controlling office equipment from mobile devices and design of client-device interfaces. They too have not addressed the issues involved in composing complex services from simple devices with specific interaction patterns between them. In general, none of the aforementioned approaches consider scenarios in which multiple specialized devices need to offer their services in a cooperative manner for the

provision of a more complex service, a case which we believe will be increasingly common in a ubiquitously networked world.

IBM's PIMA has a vision somewhat similar to ours. In their vision paper [4], they argue very briefly for the design of applications by decoupling concerns from specific physical devices. However, they do not mention any particular approach for realizing this vision in that paper. Our task graph concept on the other hand is a systematic and concrete approach which can help realize this vision.

Javaspaces [32] is an object oriented framework proposed by Sun Microsystems for dynamic communication, coordination, and sharing of objects between Java based network resources. AirJava [55] combines Sun's Jini technology and pico-cellular wireless technology to empower devices to discover each other. We believe that these are orthogonal to our research since they propose specific technologies to access services in the network.

2.3 Task Graphs

The concept of a task graph (TG) was originally developed and used in parallel computing and scheduling research community. In that context, TGs are used for representing programs that can be partitioned into sub-programs which can be parallely executed after they are allocated to different homogeneous processors connected by regular high-performance interconnects such as hypercubes or meshes. Key goals there are to find optimal task to processor assignment which results in minimum total completion time for the program or minimum total communication cost [19, 56, 43]. Our concept of a task graph is different from this classical one. We are not specifically concerned with tasks that are to be distributed among multiple homogeneous processors for speed-up; rather, most tasks that we envisage in this work will involve several homogeneous or heterogeneous devices that can be mobile and communicate with each other over failure prone wireless links. Moreover there is no goal of minimizing the total completion time of the task. However, if we are

interested in solving a large scale distributed computing task on a network of homogeneous mobile devices, then our definition of a task graph will be similar to the classical one. This makes our task graph formulation more general than the one used in the parallel computing context.

Also, MANETs are not likely to possess regular interconnection network structures such as hypercubes, meshes or butterflies as enjoyed by parallel computing platforms; therefore, the applicability of the mapping algorithms/heuristics developed for parallel computing is reduced in our application space. Another salient point of difference is the impracticality of centralized mapping algorithms for large TGs in MANETs. This is because all devices in a MANET are likely to possess limited resources and the network topology can rapidly change with time. Parallel computers or even a distributed network of workstations do not suffer from the above constraints.

The TG abstraction of a distributed task is advantageous in many ways. It is inherently distributed, as most pervasive applications and services of the future are likely to be, since more and more specialized devices will need to communicate with one another to offer more and more powerful services. It also offers hierarchical composability, as collections of devices can be logically grouped together to constitute a single node in a TG.

2.4 Multicast, Overlay Networks, and Anycast

IP Multicast is a classical networking paradigm that facilitates communication between a given group of hosts while utilizing network bandwidth efficiently [69, 27]. The possible styles of communication are "one-to-many" and "many-to-many." The Mbone [29] which was built in 1992 by the IETF is a virtual network consisting of IP routers that support multicast at the IP layer. It allows multicast packets to travel through routers that are set up to handle only unicast traffic by means of *tunneling* which is nothing but the mechanism of encapsulating multicast packets in traditional unicast packets. Due to the difficulty of

achieving community-wide consensus towards implementation of these abstractions at the IP layer, recently, researchers have proposed application layer solutions using overlay networks for implementation of these abstractions in the wired Internet. End System Multicast (ESM) is a recent proposal that advocates creation of application level overlays for supporting multicast [23]. MANETs do not suffer much from similar deployment problems as their wired counterparts since every host is likely to act as a router as well. Several approaches have been proposed by researchers to achieve multicast in MANETs. They have been compared against each other in [53].

The above approaches facilitate group communication between a given set of nodes in the network (which may be changing over time). In other words, a packet sent by one node gets replicated appropriately in the network and delivered to *all* the participants in the multicast group. However, our task execution model has significantly different requirements since (1) the set of devices in the network participating in the application is not pre-specified; only their logical specification exists (nodes in TG), and (2) the communication patterns between the logical nodes (edges in TG) can be different from the 1-to-n model. Therefore, a different set of primitives are needed for this purpose.

Anycasting is a networking paradigm that facilitates location and subsequent communication with *any* one of a set of distributed servers or service access points in the network [60]. The primary motivation for anycast in the Internet is for server selection within a group of replicated servers [39]. More recent proposals on anycasting advocate its use at the application layer [16]. However, Park and Macker in [59] propose an anycasting scheme for MANETs that is tightly coupled with the underlying routing scheme (which has to be link-state based). The Intentional Anycast feature in INS [1], although independent of the IP routing layer, is closely integrated with the underlying name resolution scheme in the network. It facilitates delivery of application messages to a device in the network that best satisfies the name of the requested service.

Our approach and focus in this dissertation are significantly different from the work

related to anycasting in [1, 59]. Although we too strongly believe that anycasting can serve as a powerful tool for decoupling the structure of distributed applications and specific physical devices that can execute them, we are not restricted to anycasting alone for a particular service. We instead use anycasting as a useful paradigm for improving the resilience of the entire distributed application in failure prone environments such as MANETs where mobility can cause services to become unreachable and thus unavailable during application execution. Also, the anycasting scheme presented in this dissertation operates at the application layer and can operate on any underlying routing and transport scheme.

There have been a few recent research efforts directed towards improving the reliability of distributed applications in the wide area Internet. MIT's Resilient Overlay Network (RON) architecture [2] focuses on the construction of an application layer overlay on top of the existing Internet routing substrate. RON allows distributed Internet applications to detect and recover from path outages and periods of degraded performance. The RON nodes monitor the functioning and quality of the Internet paths among themselves, and use this information to decide whether to route packets directly over the Internet or by way of other RON nodes, optimizing application-specific routing metrics. Our task execution protocols share some of RON's goals related to application resilience; however, owing to the dynamic nature of MANETs we propose very different techniques for recovery from disruptions. Moreover, the sources of application outages in both cases are different too. In RON, outages usually occur due to congestion whereas in MANETs, they occur due to unreachability of a resource due to physical mobility of devices.

In subsequent chapters, we propose and justify a task graph based framework for execution of complex distributed tasks on a set of mobile devices that form a MANET. We focus on the design of protocols for task graph based resource discovery in these MANETs such that they are adaptive to device mobility and failures during the execution of the task.

Chapter 3

A Task Based Approach for Resource Discovery in Mobile Ad Hoc Networks

In the past few decades, a variety of distributed applications have been enabled by many advances in computer networking. A distributed networked application or task is composed of several components or sub-tasks. These components often execute on different hardware devices and communicate among each other in order to yield a desired result. Traditional parallel and distributed computing platforms are comprised of high performance nodes internetworked with static high capacity links. However, as mentioned in Chapter 1, the computation and communication substrate offered by a MANET is potentially mobile and hence, prone to link failures. Therefore, it is necessary to develop a model for a distributed application which decouples the bindings between its logical components and the actual hardware devices that they are executed on until application runtime. Additionally, the model should utilize the component-level structure of an application in order to dynamically discover and select appropriate devices in the network with desired capabilities for hosting and executing the aforementioned application components.

In this chapter we describe in detail the methodology for modeling distributed applications on MANETs based on task graphs. We first lay the foundations of the modeling framework by introducing the necessary terminology and their definitions. We also propose a simple tuple-based framework for the specification of the components of a distributed application and show how it translates to a task graph structure. We then present the formulation of the task embedding (or anycasting) problem in the MANET framework and propose some important performance metrics that are optimized in the solution process.

3.1 Terminology

3.1.1 Preliminaries

Definition 3.1 A **device** in our context is a physical entity that performs at least one particular function such as interaction with its physical surroundings, computation, and communication with other devices. It may be equipped with an embedded processing element, sensors and actuators for interacting with the physical environment, a wireless communication port, and/or a user interface.

If a device primarily performs one specific function, it is called a "specialized device," otherwise, it is referred to as a "multipurpose device." Examples of the former type include digital cameras, speakers, printers, keyboards, display devices etc., while examples of the latter include Personal Digital Assistants (PDA) and portable notebook computers.

The capabilities of each device can be summarized in their *attributes*. Attributes can be static (i.e., time-invariant) or dynamic (i.e., time-variant). For example, a network digital camera can have a static attribute "resolution" which can take values like 320x240, 640x480 etc. Examples of dynamic attributes include location (absolute or relative, depending on the availability of GPS), available computational power, and current load. In this dissertation, we only consider devices with their principal attribute, (i.e., their primary function). Multi-

attribute extensions are possible and are considered elsewhere [1].

Definition 3.2 A **service** is a functionality provided by a device or a collection of cooperating devices. A service provided by a single device is referred to as a *simple* service whereas one provided cooperatively by a collection of devices is referred to as a *composite* service.

Multiple devices can exist in the MANET for providing the same service. For example, there can be multiple wireless cameras in the network which a user can choose from for taking a picture. We refer to this situation as "multiple instances of wireless camera services."

Definition 3.3 Service composition is the process of construction of an instance of a composite distributed service from other simple or composite service instances available in the current networked physical space.

A major portion of this thesis concentrates on the composition of composite services from simple services only. However, in Chapter 8, we present techniques for achieving service composition from composite services, as well.

Definition 3.4 A **node** is an abstract representation of a device or a collection of devices characterized by a minimal set of attributes that can offer a particular service.

A node is *simple* when it represents a single physical device. It is *complex* when it represents multiple simple nodes. We consider simple nodes only in this paper. We refer to the principal attribute of a node or a device as its *class* or *category* or *type*. Examples of classes include printer, speaker, joystick etc.

Definition 3.5 An **edge** is a necessary association between two "nodes" with attributes that must be satisfied for the completion of a task.

Examples of edge attributes include causal ordering, relative importance in the overall task, required data rate between nodes, allowable bit error rate, and physical proximity.

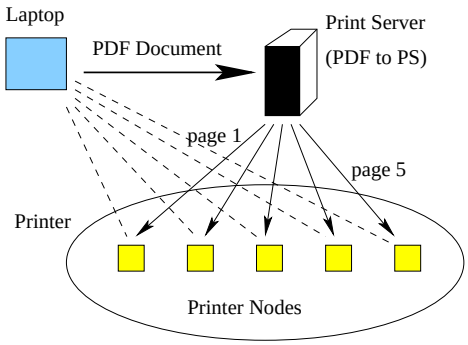


Figure 3.1: A Smart Printing Service

3.1.2 Tasks and Task Graphs

Definition 3.6 A **task** can be described as work executed by a node with a certain expected outcome. The work done by a component of a complex node is considered a *sub-task* of the larger task.

An *atomic* task is an indivisible unit of work that is executed by a simple node. Atomicity is related to the core capability of a device, described through its attributes, and is partially constrained by subjective design choices.

Definition 3.7 A **task graph** is a graph $TG = (V_T, E_T)$ where V_T is the set of *nodes* that need to participate in the task T, and E_T is the set of *edges* denoting data-flow between participating nodes.

Definition 3.8 Instantiation or **Embedding** of a task graph TG on a MANET represented by a graph G is the process of mapping all nodes of TG to nodes in G such that their attributes match. The process also maps edges in TG to paths (single-hop or multi-hop) in G.

An example brings further clarity to the abstractions developed so far. Consider a scenario in which there is a PostScript (PS) printer connected to a computer (print server) running conversion software that can convert Portable Document Format (PDF) files to

printable PS format. The printer node and the computer node each represent *devices* that offer particular *services*. The printer is considered a *specialized device* offering the service of converting PS files into printed pages, while the computer is a *multipurpose device* which has among its many offered services the one service of converting PDF files into PS format. This example is illustrated in Figure 3.1 where the task of printing a PDF document to a single or multiple printers has been logically represented as a task graph.

The printer is a physical device representation of a *simple node* with certain *attributes* (such as print resolution, color capabilities) and it offers the service of converting PS files into printed pages. Analogously, the print server computer plus its conversion software can be viewed as a representation of a *PDF → PS converter node*. By taking these two nodes together we can form a *complex node* that offers a "PDF printing service." Let a *task* be the printing of one PDF document. In this specific case, based on subjective criteria, we define an *atomic task* to be the printing of one page of the document.[1] The entire document can be then printed on a set of available printers as shown in Figure 3.1. The mechanisms of how appropriate physical devices are discovered and selected to perform a sub-task are discussed later in this Chapter and then in more detail in Chapter 5.

Note that in the above scenario, we formed a new service, PDF printing, by combining simpler existing ones. Granted that this example is simplistic, we believe that research that enables such capability in today's MANETs for *arbitrary* device types and quantities is essential for exploiting the networked environment's full potential.

3.1.3 A Taxonomy of Tasks

We broadly classify tasks into the following distinct categories:

Preassigned Tasks Specific devices need to participate in the task. In this case, nodes in the task graph already have physical addresses mapped to them and hence discovery is

[1] We assume that the printer API does not work at the granularity of printing a dot.

not required. We refer to these as *bound* nodes. Therefore, the problem of embedding a task is simply equivalent to finding suitable (not necessarily the shortest) routes between pairs of devices that are directly connected by an edge in the task graph. If the optimization variable is "load" on intermediate forwarding devices instead of delay, algorithms for load balancing should be executed instead of a shortest path algorithm.

Non-preassigned Tasks In this case we have a number of homogeneous or heterogeneous computing devices in the network which provide specific services. Unlike the *preassigned* case, nodes in the task graph are logical entities and do not signify devices with specified physical addresses. In fact, any device that can satisfy the requirements specified in the node's attribute set is a candidate for participating in the task. We, therefore, refer to such tasks as "anycastable." Communication between selected devices need to satisfy the edge attributes as well. Since all nodes in a task graph corresponding to such a task are free to be chosen, we refer to them as *free* nodes. On the contrary, nodes in a preassigned task are referred to as *bound* nodes. Optimization of certain performance measures is desirable during the process of instantiation of task graphs. This is described in more detail in Section 3.4.

Partially *preassigned* tasks have some nodes in their task graph that are bound to pre-specified devices. These *bound* devices have to be selected in the physical network whereas the remaining *free* nodes can be chosen smartly. As in anycastable tasks, the choice of *free* nodes is governed by certain optimization criteria.

Most existing networked distributed applications fall into the *preassigned* category as there is no freedom in the choice of devices and the user decides beforehand which devices participate in the application. We believe that with the advent of pervasive computing, a whole class of *anycastable* tasks will emerge by exploiting the philosophy of loose coupling between services and the devices offering them.

In the context of the smart presentation application, a pocket PDA containing the

presentation slides and a particular overhead display can be *bound* devices but the keyboard, the mouse and the smart storage are *free* devices, instances of which can be smartly chosen from the available network.

3.2 A Data-flow Tuple Representation Model for Distributed Tasks

In this section, we propose a simple tuple-based framework for the high level representation of the logical relationships between different components of a distributed application. The entire application is modeled by a set of tuples each corresponding to a particular data-flow in the application. In other words, each tuple corresponds to a logical unit of data processing that is needed between the distributed components of an application. Every application component is characterized by a *tuple node* with the same semantics as that of a *node* described in Section 3.1. Each unit of data-flow is originated at a certain tuple node and is consumed at one or more terminal tuple nodes (called *sinks*) after being processed and relayed by a set of intermediate tuple nodes. Consider the *smart presentation* application described in Section 1.3. The following data-flows can characterize a sample presentation:

1. Presenter's PDA (U) sends presentation data (e.g., a Powerpoint slide) to Smart Storage (SS) which hosts appropriate presentation software.

2. Keystrokes are originated at a wireless keyboard (K) by the presenter.

3. Mouse commands are originated at a wireless mouse (M) by the presenter.

4. SS receives presentation data, keystrokes and mouse clicks, processes the data and displays them on a projected display (D) and a local screen (LS). SS also extracts and sends the ASCII part of the presentation and some corresponding notes to the user on her PDA screen (U).

To represent such application data-flow between nodes we employ a generalized tuple architecture. If a node of type X receives data from nodes of types A, B and C, and sends the processed data to nodes of types D and E for a certain application flow (e.g., mouse commands or keystrokes or something more application specific), we can represent this data-flow schematically using the following tuple:

$$X : [A, B, C; \{processing\}; D, E]_{tag}$$

Each data-flow can be uniquely identified at any node by its *tag* attribute. We denote by $\{processing\}$ the transformation of the incoming data units from source nodes before they are transmitted to the destination nodes.

Generating Task Graphs from Tuples: Currently, the user node is expected to specify the data-flows in the distributed application as a set of tuples using a standardized language. A Task Graph (TG) representation can be generated from a tuple representation quite easily. Each task graph *node* is derived directly from the corresponding tuple node since it bears one-to-one correspondence with the latter. A task graph *edge* is created between TG nodes X_i and X_j if a data-flow exists from the tuple node corresponding to X_i and the one corresponding to X_j.

The application data-flows for the smart presentation application can be depicted as tuples as shown in Table 3.1 and they translate to the task graph shown in Figure 1.1(a).

Advantages of the Tuple Representation: Having a data-flow tuple representation for a task serves two purposes: (1) It is a natural and structured specification of the data-flows in a task from which a task graph can be derived easily, and (2) after the logical resources specified in the task graph are mapped to physical devices in the MANET, tuples govern the flow of actual application data at each participating device.

Examples of data-flow tuples presented in this section contain only the essential in-

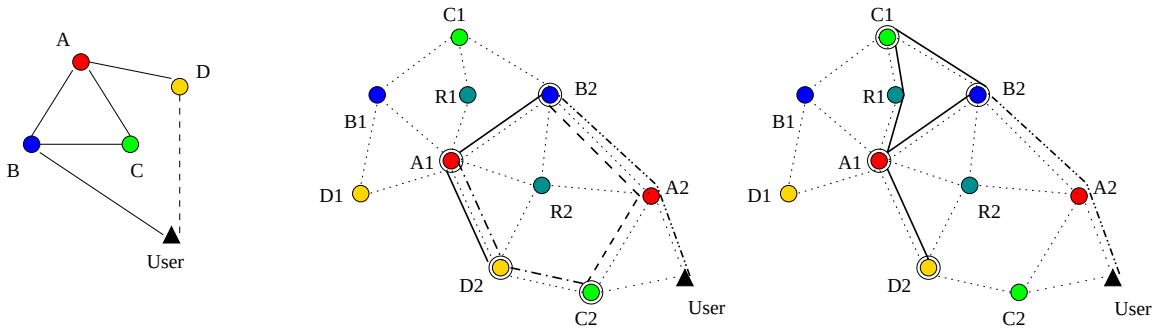
Table 3.1: Data-Flow Tuples for the Smart Presentation Task

ID	Node	Data-flow Tuples
1	U	$[-; SS]_{ppt}\ [SS; -]_{notes}$
2	SS	$[U; LS, D]_{ppt}\ [K; LS, D]_{keys}\ [M; LS, D]_{clicks}\ [U; \{ppt \rightarrow notes\}; U]_{notes}$
3	K	$[-; SS]_{keys}$
4	M	$[-; SS]_{clicks}$
5	LS	$[SS; -]_{ppt,keys,clicks}$
6	D	$[SS; -]_{ppt,keys,clicks}$

formation for data exchange, namely the data source and the data destination, and whether the incoming data needs any processing before it is relayed to another device. In general, the edges in a TG can have attributes such as upper bounds on channel error rates, bandwidth, etc. which reflect the quality-of-service (QoS) needs of a distributed application. These, and requirements such as *proximity* (since devices like keyboard, mouse etc. should be located as near the user as possible) can also be integrated in the TG via the tuple architecture. A direct way of incorporating such requirements and task constraints is by specification of *edge attributes* in the tuple. For example, consider a scenario where a node of type X needs to communicate with another node of type D such that the separation between them is no more than 3 MANET hops and that the average delay over that path does not exceed 10 milliseconds. These two requirements are specified as attributes of the edge $e = (X, D)$ in the corresponding task graph: $e.separation \leqslant 3$ and $e.delay \leqslant 0.01s$. Implementation details of most of these edge attributes are beyond the scope of this research, and are not considered further.

3.3 Embedding Task Graphs onto Networks

The first step in executing a distributed application on a set of specialized devices is to *discover* appropriate devices in the network and to *select* from the ones who responded, the



(a) Task Graph (b) Embedding #1 (c) Embedding #2

Figure 3.2: Example of Task Graph Embedding

devices that are suitable for the execution of the more complex application. Mathematically speaking, embedding a task graph $TG = (V_T, E_T)$ onto a MANET graph $G = (V_G, E_G)$ involves finding a pair of mappings (φ, ψ) such that $\varphi : V_T \rightarrow V_G$ and $\psi : E_T \rightarrow P_G$, where the type or *class* of $v \in V_T$ is the same as that of $\varphi(v)$ and P_G is the set of all source-destination paths in G. Figure 3.2(a) depicts a hypothetical task graph. Figures 3.2(b-c) show a sample network topology with two possible embeddings of TG on it.

The entire process of device discovery, selection of a device from multiple instances of devices in the same category, and the assignment of a physical *device* to a logical *node* in the task graph is referred to as *instantiation*. We also refer to the collective process as *task embedding* or *task-based anycasting* [10].

3.4 Metrics for Performance Evaluation

Definition 3.9 The embedding function (φ, ψ) maps nodes and edges in $TG = (V_T, E_T)$ to *devices* and *paths* in G. **Average (Maximum) Dilation** of an embedding is the average (maximum) length of such paths taken over all edges in TG. Mathematically, if $\|a, b\|_G$ denotes the length of a shortest path between nodes a and b in G, average and maximum

dilation are respectively given by:

$$D_{avg} = \frac{1}{|E_T|} \sum_{e \in E_T} \|\psi(e)\|_G = \frac{1}{|E_T|} \sum_{(x,y) \in E_T} \|\varphi(x), \varphi(y)\|_G \quad (3.1)$$

$$D_{max} = \max_{e \in E_T} \|\psi(e)\|_G = \max_{(x,y) \in E_T} \|\varphi(x), \varphi(y)\|_G \quad (3.2)$$

Average dilation is an important metric since it impacts the throughput between instantiated devices. An embedding with large dilation signifies long paths between directly communicating devices, which is undesirable in MANETs since TCP throughput drops significantly with increase in hop distance [42]. In contrast, an embedding with low dilation results in better task throughput. We consider the weighted version of the metric in Section 3.5 where we formally describe the optimal embedding problem.

Definition 3.10 Instantiation time is a metric which measures the time taken to embed or instantiate all nodes in TG onto G.

Definition 3.11 Re-instantiation time measures the time taken to find a replacement device after an embedding is disrupted owing to node, link, or route failures.

Definition 3.12 Average Effective Throughput, ($AvgEffT$), is the average number of application data units (ADUs) actually received at instantiated data sinks divided by the number of ADUs that were supposed to be received at the intended targets in an ideal situation.[2] Therefore, $0 \leqslant AvgEffT \leqslant 1$. It is a useful metric for measuring the resilience of the protocols to failures.

Definition 3.13 Source-to-sink delay is the latency suffered by an ADU as it funnels itself through various intermediate relay nodes in the instantiated task graph. This metric is useful for measuring application performance during transmission of task data.

The above metrics are useful in the performance evaluation of our embedding algorithms (see Chapter 6). Additional metrics that have not been investigated in this research have been listed in Appendix B.

[2] If a relaying node in the path from source to sink becomes uninstantiated, effective throughput will be affected because some data-flows will be discarded and will not reach the data sinks.

3.5 Optimization Problem Formulation

We formulated the constrained task graph embedding problem as the following optimization problem:

> Let C be a set of principal attributes (or classes) of specialized devices. Let $G = (V_G, E_G)$ be a graph corresponding to a MANET, with the class of each device in V_G belonging to C. Let $TG = (V_T, E_T)$ be a task graph such that the class of each node in V_T belongs to some $S \subseteq C$. Function $w : E_T \rightarrow \mathbb{R}^+$ defines edge weights which could signify application data-flow requirements.
>
> Find mappings $\varphi : V_T \rightarrow V_G$ and $\psi : E_T \rightarrow P_G$, where the *class* of $v \in V_T$ is same as that of $\varphi(v)$ and P_G is the set of all "paths" in the network G, such that the weighted average dilation given by:
>
> $$D_{avg}^{(wt)} = \frac{1}{\sum\limits_{e \in E_T} w(e)} \sum_{e=(x,y) \in E_T} w(e) \, \|\varphi(x), \varphi(y)\|_G \tag{3.3}$$
>
> is minimized, where $\|a, b\|_G$ denotes the shortest path between devices a and b in G.

In the next chapter, we investigate the computational complexity of the general version of the problem where a task graph can have multiple nodes belonging to the same class, and then that of a more specialized version of the problem where all nodes in a task graph belong to distinct classes. We show that the above problem is NP complete in both these situations. However, the problem becomes tractable when the task graph is a tree with nodes belonging to distinct classes, and we give a polynomial time algorithm to solve it (see Section 4.2.3). The solution approach in that section assumes that the user node possesses the knowledge of the entire network topology as well as that about the capabilities of the devices in the network. In Chapter 5, we propose distributed algorithms for embedding, which albeit suboptimal, operate locally and are fast.

Chapter 4

Task Embedding: Theoretical Foundations and New Algorithms

Graph embedding has been extensively studied in the parallel computing literature in the past [19, 56]. However, researchers restricted themselves to cases in which the host network has *regular* topologies such as mesh, hypercube, binary tree, etc. owing to the fact that most parallel computing platforms are comprised of processors connected in a regular fashion leading to regular, tightly-coupled networks. Researchers have established bounds on the mapping problem in some of these restricted settings. In contrast, the application domain considered in this dissertation is characterized by a loosely coupled group of processing elements that can be usually represented by a general graph G instead of a regular topology such as a mesh or a tree. The processing elements are possibly mobile (i.e., G is time varying) and can provide services to other elements in order to complete a certain distributed task at hand.

In this chapter, we investigate several variants of the graph embedding problem for some special cases and establish hardness results for each with respect to minimization of the average dilation metric. The primary motivation behind proving NP-completeness

of decision problems corresponding to the different variants of embedding problems is to establish that these problems are likely to not admit exact polynomial time solutions and one should look for approximate heuristic optimization approaches that have polynomial running time.

Our original contributions in this chapter are threefold: (1) proof of NP-completeness of the Class Constrained Subgraph Isomorphism problem (Section 4.2.1) and of the Class Constrained Task Embedding problem (Section 4.2.2), (2) proposition of a dynamic programming based exact optimal algorithm, TreeEmbed, for the special case in which the task graph has a tree structure (Section 4.2.3) (we also propose the use of TreeEmbed as a heuristic for general task graphs in Section 4.2.4), and (3) proposition of a fast *greedy* heuristic algorithm GreedyEmbed which can be used to embed any general task graph onto a general network graph (Section 4.2.5).

4.1 General Graph Embedding

The most general version of the embedding problem (`GEN-EMBED`) can be stated as follows:

Instance: (TG, G, k)

1. A task graph $TG = (V_T, E_T)$ and a host network $G = (V, E)$. All nodes in the network are identical in capability. The nodes in the task graph are unlabeled which means *any* subtask corresponding to a node in G can be executed on *any* node in the host network.
2. A real number k

Decision problem: Do there exist mappings $\varphi : V_T \rightarrow V$, and $\psi : E_T \rightarrow P_G$, in which P_G is the set of all source-destination (s-d) paths in G, such that the average dilation given by Equation 3.3 is less than k?

Theorem 4.1 (GEN-EMBED) *GEN-EMBED is NP-complete.*

Proof. If $\varphi^* : V_T \rightarrow V$ is one such mapping, the corresponding edge to path mapping ψ^* can be easily calculated as follows: for a given edge $e = (u, v) \in E_T$, $\psi^*(e) = \arg\min_p (p[\varphi^*(u), \varphi^*(v)])$, where $p[u, v]$ is a path between vertices u and v in G. $\psi^*(e)$ is nothing but the shortest path in G between the mapped endpoint vertices of edge e. Given φ^*, ψ^* can be calculated using Warshall-Floyd's all-pairs shortest path algorithm which runs in $\mathcal{O}(|V|^3)$ time [24]. Now, given φ^* and after calculating ψ^*, we can calculate the dilation of the embedding using Equation 3.3 in polynomial time. Hence the problem belongs to NP.

Next we show that `GEN-EMBED` is NP-hard by a polynomial time reduction from a general instance of the Subgraph Isomorphism problem (`SUBISO`) which is long known to be NP-complete [34].

The `SUBISO` problem states that if there are two graphs G_1 and G_2, it is NP-complete to decide if G_1 is isomorphic to a subgraph of G_2. Now from a general instance (G_1, G_2) of `SUBISO`, we create an instance $(G_1, G_2, 1)$ of `GEN-EMBED`. We observe that G_1 is isomorphic to a subgraph of G_2 *if and only if* G_1 can be embedded into G_2 with an average dilation of $k = 1$. If G_1 is indeed isomorphic to a subgraph of G_2, choose the subset S of vertices and T of edges of G_2 that belong to the isomorphic subgraph. Hence, there is a one-to-one mapping between (S, T) and G_1. Since every edge in G_1 is mapped to only an edge in G_2, the dilation is exactly 1. In the other direction, if G_1 can be embedded onto G_2 with average dilation $k = 1$, it means every edge of G_1 has been mapped onto paths of length exactly 1 (only then the average dilation can be exactly 1), and it then follows that G_1 is isomorphic to a subgraph of G_2. Since `GEN-EMBED` is in NP and is NP-hard, it is NP-complete. □

4.2 Restricted Cases of Graph Embedding

In the previous section we proved that the general problem of embedding a task graph onto a host network is NP-complete with respect to the average dilation metric. Now we impose more restrictions on the embedding problem and show that it remains hard even in most restricted cases that are interesting to us.

We consider the simplest in which the attributes of all nodes in the task graph are distinct, (i.e., possess unique labels) and the nodes in the network are labeled too. Multiple nodes in the network can possess the same label. In this scenario, the mapping process is more constrained because a candidate mapping must be *label preserving*. Hence the set of candidate mappings in this case has a lower cardinality than the one mentioned in Section 4.1. However, we prove that in spite of this restriction the embedding problem is NP-complete. In future, we refer to the problem as `CC-EMBED` or *Class Constrained Embedding*.

In proving the NP-completeness of `CC-EMBED`, we first prove that the class constrained version of the Subgraph Isomorphism problem `CC-SUBISO` is NP-complete. In other words, Subgraph Isomorphism is NP-complete even if the vertices of G_1 possess unique labels.

4.2.1 `CC-SUBISO` is NP-complete

The problem is defined as follows:

Instance: Graphs G_1, G_2 and all nodes of G_1 have a *unique* label from the label universe C. Hence, if G_1 has n vertices, $n \leqslant |C|$. G_2 is assumed to have N vertices where $N \geqslant n$. The vertices of G_2 have labels assigned to them from C and multiple vertices can possess the same label.

Decision problem: Is there a subgraph of G_2 which is *isomorphic* to G_1 such that the vertices of the subgraph induced in G_2 by the isomorphism possess the same labels as their

respective vertices in G_1?

Theorem 4.2 (CC-SUBISO) *CC-SUBISO is NP-complete.*

Proof. If $f^* : V_{G_1} \rightarrow V_{G_2}$ is one such isomorphism, there exists a polynomial time algorithm which can verify that the subgraph induced by the vertex set $V^* = \{v | v = f^*(u), u \in V_{G_1}\}$ in G_2 is isomorphic to V_{G_1}, by establishing a one-to-one correspondence between the elements in the respective edge sets. Hence `CC-SUBISO` belongs to the class NP.

Next, we show that this problem is NP-hard by reducing from the `CLIQUE` problem which is known to be NP-complete. Consider a general instance of the `CLIQUE` problem, (G, k), where G is an undirected graph with n vertices and e edges, and k is a positive integer. We transform this instance into a particular instance of the `CC-SUBISO` problem in the following manner:

An instance of the `CC-SUBISO` problem can be specified as (G_1, G_2, c_1, c_2) where G_1 is the guest graph, G_2 is a target or host graph, $c_1 : V_{G_1} \rightarrow C$ is an injective function, and $c_2 : V_{G_2} \rightarrow C$ is a general function.

Polynomial Transformation

1. Define guest graph instance $G_1 = K_k$ where K_k is the clique graph on k vertices.
2. Define the universe of labels $C = \{1, 2, 3, \ldots k\}$.
3. Define labeling function c_1 such that its range is the set C. Hence all the vertices of G_1 have unique labels.
4. Define host graph $G_2 = G \odot C$ where the operation $\odot$ is defined as follows: replace each vertex v of G by a set S_v of k vertices labeled $1, 2, \ldots, k$. We call S_v a "supernode" of v. Vertices in S_v do not have any edges between them. Then replace each edge (u, v) in G by k^2 edges connecting every pair of nodes (x, y) where $x \in S_u$ and $y \in S_v$. Hence G_2 has kn vertices, and ek^2 edges. Note that c_2 is defined automatically. The transformation has been illustrated in Figure 4.1.

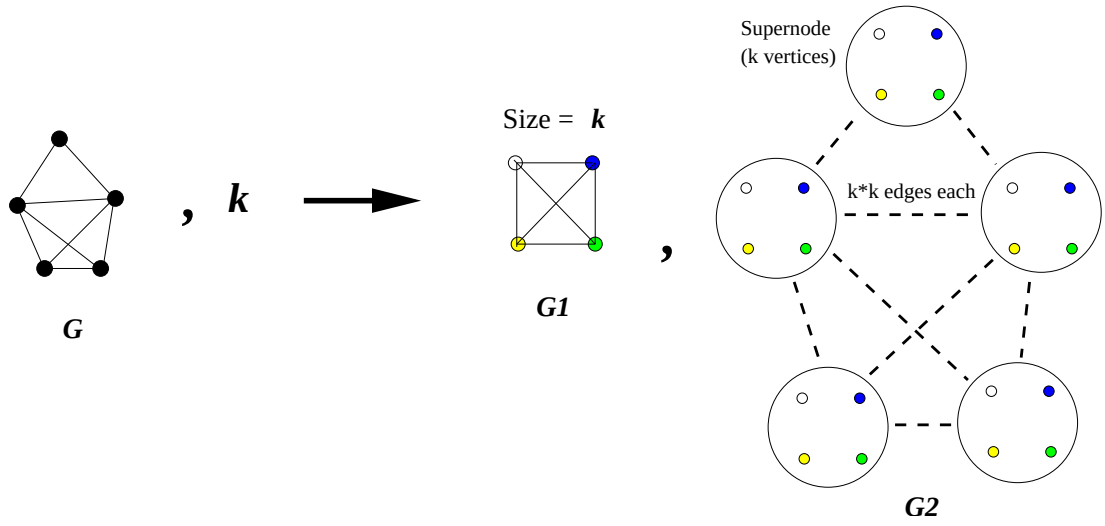


Figure 4.1: Transformation from `CLIQUE` to `CC-SUBISO`

Having defined a polynomial transformation from `CLIQUE` to `CC-SUBISO`, we show that G has a clique of size k if and only if G_1 is isomorphic to a subgraph of G_2 while obeying the labeling constraints.

First consider the forward direction. If G has a clique of size k, let us denote the clique by Q. Now consider the vertices in the set $SQ = \{S_q \mid q \in V_Q\}$ of supernodes in G_2. We can pick a vertex with label 1 from the first supernode, that with label 2 from the second, and so on till a vertex with label k from the last supernode in SQ. Clearly from the construction of G_2, each of these vertices will induce a clique of size k in G_2, and since they have unique labels, that particular induced clique is isomorphic to the input clique graph K_k while preserving the labeling.

Next, consider the reverse direction. If there exists a label preserving isomorphism between $G_1 = K_k$ and a subgraph of G_2. We show that in that case, G must have a clique of size k. Since K_k is isomorphic to a subgraph in G_2 in a label-preserving fashion, not more than one vertex from any supernode in G_2 can be a part of the induced subgraph. It is easy to see why this is the case: suppose two vertices u and v are a part of the subgraph induced by the isomorphism, and they belong to some common supernode. Now by construction of

the supernodes, there is no edge connecting u and v and this contradicts the fact that they are contained in isomorphism image of a clique onto G_2. Hence the k vertices of K_k are mapped onto k vertices with unique labels from different supernodes in G_2. The existence of an isomorphism means all these k vertices in G_2 are connected to each other. In other words, they form a clique which we represent by Q_{G_2}. Now, by construction of G_2, an edge (x, y) (where $x \in S_u$ and $y \in S_v$) in G_2 exists only if one exists in G between the corresponding pre-images of the supernodes, (i.e., u and v). Consider the set of vertices S in G which are the pre-images of the supernodes that contain the vertices in Q_{G_2}. By the previous argument and the nature of the transformation of G to G_2, since Q_{G_2} is a clique, S too must be a clique. Also, its size is k.

Next we show that the transformation happens in polynomial time and the size of the output instance is upper bounded by a polynomial function of the input size. The input instance (G, k) can be encoded in $\frac{1}{2}n(n-1) + \log k$ bits. Since $k \leqslant n$, the input size is $O(n^2)$. The output instance can be encoded in $\frac{1}{2}k(k-1) + k\log k + \frac{1}{2}nk(nk-1) + nk\log k$ bits which is $O(n^4)$. Hence the size of the output instance is quadratically upper bounded by that of the input instance. Also, clearly the transformation takes polynomial amount of time in the size of the input. Since `CLIQUE` is NP-hard, so is `CC-SUBISO`. And, since `CC-SUBISO` is in NP, it is NP-complete too. This concludes our proof. □

4.2.2 `CC-EMBED` is NP-complete

We show that the class constrained version of graph embedding is NP-complete. Formally the problem can be stated as follows:

Instance: Graphs G_1, G_2; All nodes of G_1 have a *unique* label from the label universe C. Hence, if G_1 has n vertices, $n \leqslant |C|$. G_2 is assumed to have N vertices where $N \geqslant n$. The vertices of G_2 have labels assigned to them from C and multiple vertices can possess the same label. ℓ_1, ℓ_2 are the corresponding representations for the labeling function.

Decision problem: Do there exist *label-preserving* mappings $\varphi : V_T \rightarrow V$ and $\psi : E_T \rightarrow P_G$, where P_G is the set of all source-destination (s-d) paths in G, such that the average dilation given by Equation 3.3 is less than k? A label-preserving mapping φ is defined as one which takes a vertex $v \in V_T$ with label (or attribute) ℓ to a vertex $v' \in V$ with a label ℓ.

Theorem 4.3 (CC-EMBED) *CC-EMBED is NP-complete.*

Proof. The proof is very similar to the one given in Section 4.1. If $\varphi^* : V_T \rightarrow V$ is one such mapping, the corresponding edge to path mapping ψ^* can be easily calculated. For a given edge $e = (u, v) \in E_T$, $\psi^*(e) = \arg\min_p (p[\varphi^*(u), \varphi^*(v)])$, where $p[u, v]$ is a path between vertices u and v in G. $\psi^*(e)$ is nothing but the shortest path in G between the mapped endpoint vertices of edge e. Given φ^*, the mapping ψ^* can be calculated using Warshall-Floyd's all-pairs shortest path algorithm as mentioned in Theorem 4.1. Now, given φ^* and after calculating ψ^*, we can calculate the dilation of the embedding using Equation 3.3 in polynomial time. Hence the problem belongs to NP.

Next we show that `CC-EMBED` is NP-hard by a polynomial time reduction from a general instance of the Class Constrained Subgraph Isomorphism problem (`CC-SUBISO`) which by Theorem 4.2 is NP-complete.

The `CC-SUBISO` problem states that given two graphs G_1 and G_2, and labellings $\ell_1 : V_T \rightarrow C, \ell_2 : V \rightarrow C$ for their nodes (labels in G_1 being distinct), it is NP-complete to decide if G_1 is isomorphic to a subgraph of G_2 while obeying the label matching property. Now from a general instance $(G_1, G_2, \ell_1, \ell_2)$ of `CC-SUBISO`, we create an instance $(G_1, G_2, 1, \ell_1, \ell_2)$ of `CC-EMBED`. We observe that G_1 is isomorphic to a subgraph of G_2 in a label-preserving manner *if and only if* G_1 can be embedded into G_2 in a label preserving manner with an average dilation of $k = 1$. If G_1 is indeed isomorphic to a subgraph of G_2 in a label-preserving manner, choose the subset S of vertices and T of edges of G_2 that belong to the isomorphic subgraph. Hence, there is a one-to-one mapping between (S, T) and G_1. Since every edge in G_1 is mapped to only an edge in G_2, the dilation is exactly 1. In the

other direction, if G_1 can be embedded onto G_2 in a label-preserving fashion with average dilation $k = 1$, it means every edge of G_1 has been mapped onto paths of length exactly 1 (only then the average dilation can be exactly 1), and it then follows that G_1 is isomorphic to a subgraph of G_2 while preserving labels. Since `CC-EMBED` is in NP and is NP-hard, it is NP-complete. □

4.2.3 An Optimal Polynomial-time Embedding Algorithm for Tree Task graphs with Distinct Labels

Although the `CC-EMBED` problem is NP-complete with respect to the average dilation metric for the general graphs, there is an interesting special case of a tree which lends itself to an optimal polynomial time solution.

We present below TreeEmbed, an optimal algorithm (with respect to D_{avg}) for embedding a *tree* task graph TG onto a host network G. The running time is polynomial in $|G|$ as well as $|TG|$. The algorithm minimizes searching in the solution space by exploiting the *tree* structure of TG, and is based on the principle of optimality.[1] The algorithm requires that the node executing the algorithm have complete knowledge of the snapshot of the network topology at the given instant of time.

For each node X in TG, algorithm 4.1 seeks to discover the *best* embedding for each child node Z at every instance (x) of X in G. After the best child candidates are known for all instances, the optimal cost embedding φ^* is selected starting at root node U.

The algorithm proceeds by the propagation of a certain value function $v(.)$ from the leaf nodes of TG towards the root node U. The crux of the idea is that the principle of optimality holds because of the tree structure of TG: if a device instance x of node X is selected by its parent and is optimal, then the choice of instance z (of X's child Z)

[1]The Principle of Optimality holds for problems whose structure is such that their optimal solutions contain those for the smaller sub-problems [14].

is optimal too. This greatly reduces the search space for an exact optimal embedding. Moreover, embedding of children nodes can proceed independently of each other because they possess distinct attributes. After carrying out this step for all children of X for each instance x, assign the sum of the calculated minimum values to $v(x)$. Figure 4.2 illustrates the procedure for a task graph of 6 nodes. $\Gamma_B = child(B)$ is the set of children of B in TG. k_j is an instance in G of child k of B in TG.

Theorem 4.4 (TREE-EMBED) TreeEmbed *runs in polynomial time.*

Proof. The running time of the TreeEmbed algorithm can be calculated in the following manner: Assigning levels to nodes in TG (Alg.-4.1::2-3)[2] takes $O(|V_T|)$ time. If the nodes with the same level are pushed onto a stack, then nodes with $L = L_{max}$ will reside at the top and value propagation can occur efficiently. In the worst case, $L_{max} = |V_T| = O(|V_T|)$; although in more balanced trees, $L_{max} = O(\log |V_T|)$.

The embedding function $\varphi_x()$ is calculated for all relevant device instances x in Alg.-4.1::9-15. Suppose there are $|C|$ classes of devices in G with $\frac{|V|}{|C|}$ instances of each class, on average. For every parent instance $x \in V$, each child instance $z \in V$ is considered by the embedding algorithm: the shortest path between x and z is computed (in $O(|V|^2)$ time); the minimization step in Alg.-4.1::11 is performed (in $O(\frac{|V|}{|C|})$ time). Since this process is performed for all edges in TG the time complexity of Alg.-4.1::7-17 is $O(|E_T| \times \frac{|V|}{|C|}(\frac{|V|}{|C|} \times |V|^2 + \frac{|V|}{|C|})) = O(|E_T|\frac{|V|^4}{|C|^2}) = O(|V_T|\frac{|V|^4}{|C|^2})$. Note that the "for loops" in Alg.-4.1::7-8 are subsumed in this calculation. Since $|V|$ is the dominant term, the time complexity is given by the above expression itself. Finally, the optimal embedding (φ^*, ψ^*) is calculated starting from the root node in Alg.-4.1::18-26; the time complexity of this step is also subsumed in the above term.

If we use Warshall-Floyd's all-pairs shortest path algorithm (running time is $O(|V|^3)$ and extraction of shortest path cost is $O(1)$ assuming random access storage) instead of the

[2]In this dissertation, we use the notation Alg.-<Algorithm No.>::<line no.(s)> to refer to specific listings of a particular algorithm.

Algorithm 4.1 TreeEmbed(TG, G, w, c_1, c_2)

Given: Tree Task Graph, $TG = (V_T, E_T)$; $w : E_T \to \mathbb{R}^+$; $c_1 : V_T \to C$),

Host Network Graph $G = (V, E)$; $c_2 : V \to C$);

/* C: attribute universe; c_1, c_2: attribute fns.; c_1 is injective; */

$\forall X \in V_T : X$ is a leaf in TG, $L[X] \leftarrow 0$; /* assign levels to each leaf node */

$\forall X \in V_T : X$ is not a leaf in TG, $L[X] \leftarrow 1 + \max_{Z \in child(X)} L[Z]$; /* and the rest */

for all $(X : L[X] == 0)$ **do**

  $\forall x : (c_2(x) == c_1(X))$, $v(x) \leftarrow 0$; /* assign value to matching instances */

end for

for $(\ell \leftarrow 1; \ell \leqslant L_{max}; \ell \leftarrow \ell + 1)$ **do**

  for all $(X \in V_T : L[X] == \ell)$ **do**

    for all $(x \in V : (c_2(x) == c_1(X))$ **do**

      for all $(Z : Z \in child(X))$ **do**

        $z^* \leftarrow \arg \min_{z \in V \wedge c_2(z) == c_1(Z)} \{v(z) + w_{(X,Z)} \| x, z \|_G\}$;

        $\varphi_x(Z) \leftarrow z^*$; /* best instance of child node Z for x */

        $v(x) \leftarrow v(x) + \{v(z^*) + w_{(X,Z)} \| x, z^* \|_G\}$; /* update value of x */

      end for

    end for

  end for

end for

for $(\ell \leftarrow L_{max}; \ell \geqslant 0; \ell \leftarrow \ell - 1)$ **do**

  $S \leftarrow \{X \,|X \in V_T \wedge L[X] == \ell\}$;

  while $(X \in S \wedge child(X) \neq \phi)$ **do**

    $x \leftarrow \varphi^*(X)$; /* note that $\varphi^*(U) = U$ */

    for all $(Z : Z \in child(X))$ **do**

      $\varphi^*(Z) \leftarrow \varphi_x(Z)$; $\psi^*(X, Z) \leftarrow \| \varphi(X), \varphi(Z) \|_G$; /* optimal embedding */

    end for

  end while

end for

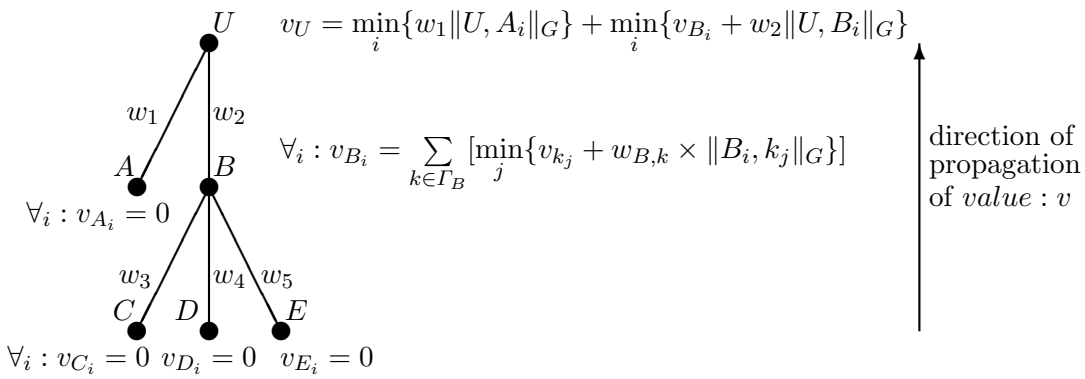


Figure 4.2: Outline of the Exact Optimal Polynomial-Time Algorithm

Dijkstra's single pair shortest path algorithm, then the running time of TreeEmbed is $O(|E_T|\frac{|V|^2}{|C|^2} + |V|^3) = O(|V|^3)$. □

Although being optimal with respect to the average dilation metric, algorithm 4.1 suffers from the following drawbacks: (1) it is a centralized algorithm and does not have a low time complexity, (2) it needs entire topology information at a particular device which may not be realistic for large MANETs, and (3) it is not adaptive to mobility of devices. A distributed version of the aforementioned optimal algorithm is promising in terms of speed and scalability but that too involves frequent broadcasts from every instance of a node in order to compute the optimal embedding. Hence, in Section 4.2.5 we propose a greedy heuristic approach GreedyEmbed which albeit suboptimal, is much simpler, less time consuming, and reasonably efficient in operation. A distributed version of GreedyEmbed is proposed in Chapter 5.

4.2.4 Application of TreeEmbed for Non-tree Task Graphs

If TG is a general graph (and not a tree), then the task embedding problem is much harder since the principle of optimality may not hold in that case. This is because the optimal embedding of every pair of nodes and the edge connecting them in TG cannot be done independently of other edges and nodes in TG, as can be done if TG were a tree. In the

case of a tree TG, as we propagate the *values* from the leaves to the root, the optimal embeddings of each subtree are retained and used later while embedding a node with lower value of L. This is not possible for any general task graph with greater connectivity than a tree.

Since there are $|V_T|$ distinct classes of devices in TG, and $\frac{|V|}{|C|}$ instances of each class in the network G ($|V_T| \leqslant |C|$), the total number of possible embeddings is $(\frac{|V|}{|C|})^{|V_T|}$. Hence, the search space for an optimal brute force algorithm is exponential in $|V_T|$, which is clearly unacceptable for large task graphs.

This combinatorial structure of the problem prompted us to look for an approximate solution, the idea of which is essentially the following: find a minimum weight spanning tree, $ST_{min}(TG)$ and then embed that in G using the TreeEmbed algorithm described in Section 4.2.3. The edges of TG that are not in $ST_{min}(TG)$ are considered by the embedding algorithm; in the worst case they can be mapped to very long paths and thus yield suboptimal results without a low approximation ratio. If the task graph is unweighted, we use the Breadth First Search tree $BFST_{TG}$ instead. In Appendix A, we derive an upper bound for the approximation factor α for this heuristic algorithm which may not be tight. This upper bound is the product of the diameter of TG and the average dilation of $BFST_{TG}$ in G. We then derive the approximation factors for several task graphs with special structures such as a cycle, a mesh and a complete graph. α is low for the cycle but can be high in the worst case for the other two cases.

A better solution may involve partitioning TG into smaller components which can be embedded into G *independent* of each other. Both finding such components and then embedding them onto G are likely to have high time complexity. Hence, we trade off complexity for optimality and look for greedy heuristic solutions which are simple and distributed in nature. We describe a centralized version of one such greedy approach in the next section.

Algorithm 4.2 GREEDYEMBED(TG, G, c_1, c_2)

1: **Given:** Task Graph, $TG = (V_T, E_T)$; $c_1 : V_T \to C$;
Host Network Graph $G = (V, E)$; $c_2 : V \to C$.
/* C: attribute universe; c_1, c_2: attribute fns. */
2: $U \leftarrow$ User Node in G; $U_T \leftarrow$ User Node in TG;
3: $TG[U_T].address \leftarrow U$; /* instantiate U_T in TG with U */
4: $visited[U_T] \leftarrow$ TRUE;
5: $instantiated[U] \leftarrow$ TRUE;
6: queue $TGQ \leftarrow$ **nil**, $GQ \leftarrow$ **nil**; /* create and initialize FIFO queues */
7: $TGQ.append(U_T)$; $GQ.append(U)$; /* push U_T, U into resp. queues */
8: **while** ($TGQ.empty() ==$ FALSE) **do**
9: $v \leftarrow TGQ.pop()$; /* get and remove the element from the front */
10: $w \leftarrow GQ.pop()$; /* get and remove the corresponding instance */
11: $visited[v] \leftarrow$ TRUE; /* mark the TG node as visited */
12: **for all** ($p \in$ neighbors(v, TG)) **do**
13: /* loop until finding the first non-visited neighbor of v */
14: **if** ($visited[p] ==$ TRUE) **then**
15: **continue**;
16: **end if**
17: /* find nearest available instance of node p in G starting at w */
18: $nearest \leftarrow$ FINDNEAREST$(G, w, c_1(p))$;
19: $TG[p].address \leftarrow nearest$; /* instantiate p in TG with $nearest$ */
20: $visited[p] \leftarrow$ TRUE;
21: $instantiated[nearest] \leftarrow$ TRUE;
22: **end for**
23: $TGQ.append(p)$; $GQ.append(nearest)$; /* do this to continue BFS */
24: **end while**

4.2.5 A Greedy Algorithm for Embedding

In Section 4.2.3, we proposed an algorithm which finds an optimal task embedding if the task graph is a tree. Although the algorithm is optimal for trees, it is not optimal for non-tree task graphs and it suffers from large time complexity even though it is polynomial. The principal reason for this is that all devices in the network G are considered as candidates for embedding and the dynamic programming algorithm chooses the best subset among them systematically. Moreover, the algorithm may often fail to run in polynomial time if a node of particular type occurs more than once in TG. Due to these reasons, we felt the need for a simple algorithm which albeit suboptimal (even for trees) has lower time complexity and works for the case where all node types in TG are not distinct.

We propose Algorithm GreedyEmbed (4.2) which is a *greedy* algorithm and it starts the search for candidate devices from the user node U itself. This embedding algorithm operates by means of a Breadth First Search (BFS) starting from the user node (root) in the task graph. At every step of the BFS process, a hitherto unvisited TG node is instantiated greedily from the current location (in G) of search. In other words, the nearest candidate in G which matches the requested type is selected. Ties are broken arbitrarily and there is no lookahead. Now, finding the nearest instance of p in G starting at device w (Alg.-4.2::18) involves a partial BFS through G rooted at w. This process takes $O(|V| + |E|)$ time in the worst case and is repeated for every node in TG. This guarantees the shortest path between w and *nearest* and results in the total time complexity of $O((|V|+|E|)|V_T|)$ which is significantly lower than the worst case complexity of TreeEmbed (Algorithm 4.1). This is primarily because unlike TreeEmbed, GreedyEmbed does not consider all devices in G as candidates for instantiation. Also, the average case complexity of GreedyEmbed maybe lower than $O((|V| + |E|)|V_T|)$ because searching for the nearest suitable instance of a TG node may not require a complete traversal of G.

GreedyEmbed also possesses a few clear advantages over TreeEmbed in its functionality and implementation. Unlike the latter, GreedyEmbed can handle the case in

which multiple nodes in TG possess the same attributes. Moreover, distributed implementations of GreedyEmbed are facilitated easily due to the nature of breadth first search. We propose a distributed approach based on these principles in the next chapter.

Chapter 5

Distributed Task Embedding on Mobile ad hoc Networks

In Chapter 4 we presented a theoretical foundation of the task embedding problem in the context of task-attribute based resource discovery on irregular MANET topologies. We presented an exact optimal algorithm TreeEmbed that executes in polynomial time for tree task graphs with distinct node attributes, and showed how it can be utilized heuristically even for general graphs. We also presented a faster, greedy heuristic algorithm GreedyEmbed whose solution search space is much more restricted than that of TreeEmbed. Both these algorithms require knowledge of the entire network topology and of attributes of all devices at the user node. Such centralized solutions are often preferable when there is an existing wired or wireless infrastructure with a fixed base station node or a directory server, devices are less mobile and more connected with each other, and the probability of node or link failure is low. However, in dynamic, mobile distributed environments such as MANETs, it can be hard for a device to track and react to the changes in topology between other participating devices using centralized algorithms and reactive MANET routing protocols. Moreover, no single device in the network may possess adequate computational power or energy to continually execute such algorithms.

It is also difficult to guarantee that a device acting as a centralized controller will always be connected to the rest of the network. When a disconnection or a partition occurs in the network, a new centralized controller, where the algorithm runs, may have to be elected, and the device must have to compute a new embedding. Although it can be easier to find close-to-optimal solutions with respect to particular performance metrics using a centralized approach than a distributed one, the latter is more robust and adaptive to mobility because there is no single point of failure. Hence, we claim that localized distributed algorithms are better suited for task embedding in dynamic MANET environments, and propose a distributed version of GreedyEmbed to achieve the same. Distributed GreedyEmbed uses local search in a distributed fashion for TG based resource discovery while heuristically optimizing the average dilation metric. It is also very responsive to disconnections during instantiation as well as task execution and facilitates local recovery.

A pertinent design question is where the protocol developed in this chapter should be implemented within the layers of the networking stack. Since our approach assumes the presence of multihop routing support in the MANET, we advocate that resource discovery and selection should reside in an intermediate layer (called the *TG-layer*) between the application and network layers. However, the requirement of a clean separation between these layers can be relaxed for better protocol and application performance. Additional design and implementation issues are addressed in Chapter 7.

5.1 A Distributed Algorithm for Instantiation of Heterogeneous Task Graphs

In this section we present a distributed algorithm for the instantiation of nodes in a given task graph. We also refer to this as anycasting of the corresponding task [10]. Initially, we consider the heterogeneous case in which each node in the task graph belongs to a distinct class (possesses distinct attributes). Later in Section 5.2, we discuss the modifications

required in the protocol to support the homogeneous and the "mixed" cases as well.

The principal goal of the distributed GreedyEmbed algorithm is to produce an embedding of a given TG onto a MANET with the objective of optimizing average dilation (defined in Section 3.4). In this chapter, each device is assumed to provide a single type of service, and all nodes in TG are assumed to be *simple*. The proposed concept can be extended to incorporate the case in which multiple types of services are provided by one particular device. The presence of a MANET routing protocol such as DSR [46], AODV [61], or OLSR [44] etc. is assumed for facilitating point to point communication between two devices. Also, the presence of a reliable transport protocol is essential for the transmission of certain important protocol control messages and data. In this work, we use TCP as the transport protocol. A UDP based scheme with acknowledgments, timeouts and retransmissions can also be used in small networks but in MANETs where congestion is likely to be a norm rather than an exception, the congestion control semantics of TCP are important. Later in this chapter and then in Chapter 7 we describe some challenges that researchers face in the design of a reliable transport protocol for MANETs.

From a given data-flow tuple description (as illustrated in Section 3.2), the corresponding task graph TG is derived; TG is then submitted by the user node's application (*taskapp*) layer to its TG-layer which then begins the embedding (or instantiation) process. This occurs on-demand and no proactive exchange of device state is assumed in the MANET.

5.1.1 Operations of the Distributed Version of GreedyEmbed

A distributed version of Algorithm 4.2 is presented in this section. All devices in the network execute a copy of the same algorithm. Every device in the system exists in a state s such that $s \in S_1 \times S_2$, where

$$S_1 = \{\texttt{COORDINATOR, NON\_COORDINATOR}\}, \text{ and}$$

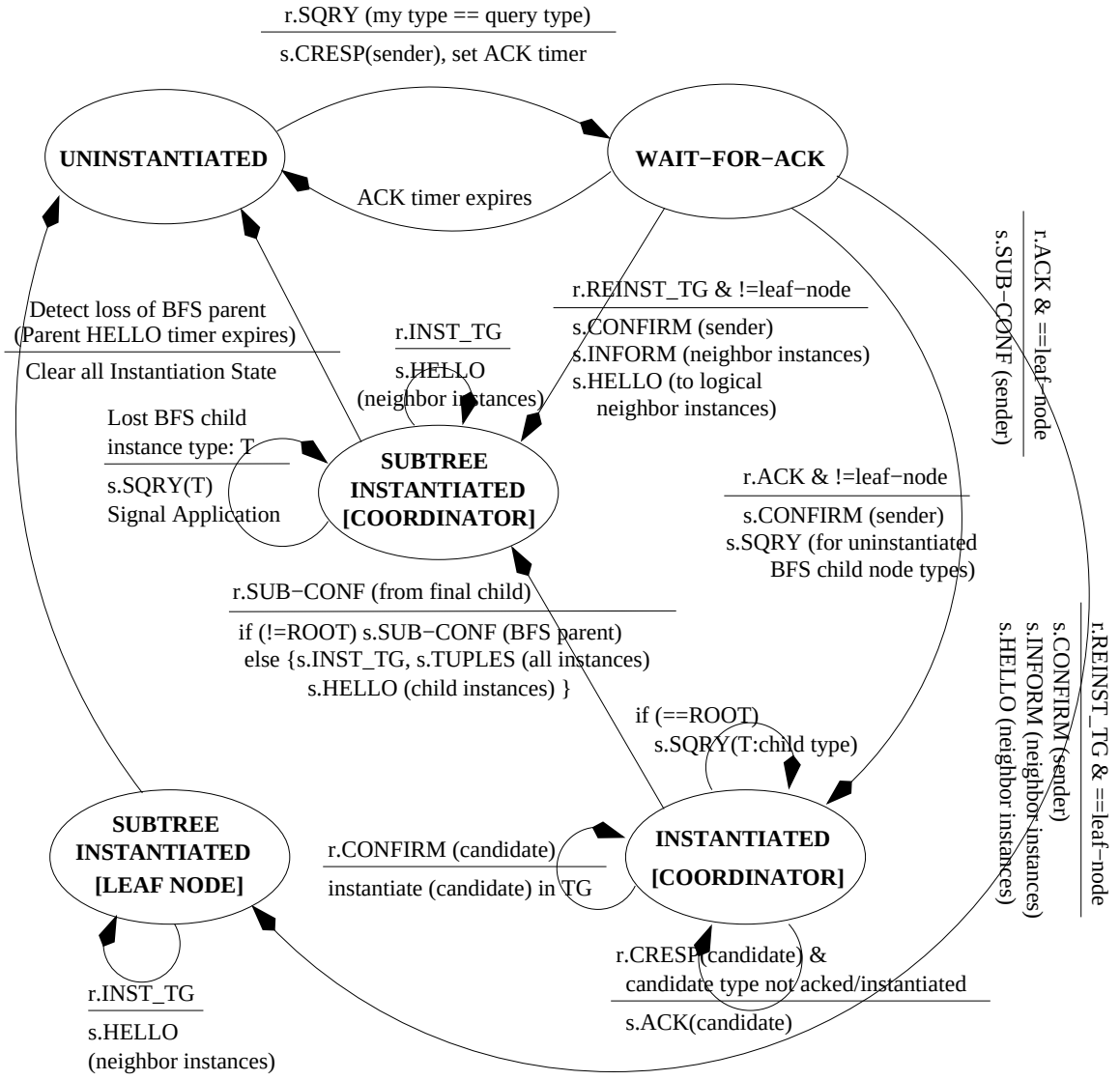


Figure 5.1: Finite State Machine Representation of the Salient Characteristics of the Instantiation Protocol: (`r.<PKT>` represents a packet reception event and `s.<PKT>` represents a packet transmission event. Data-flow tuple transmissions are not shown.)

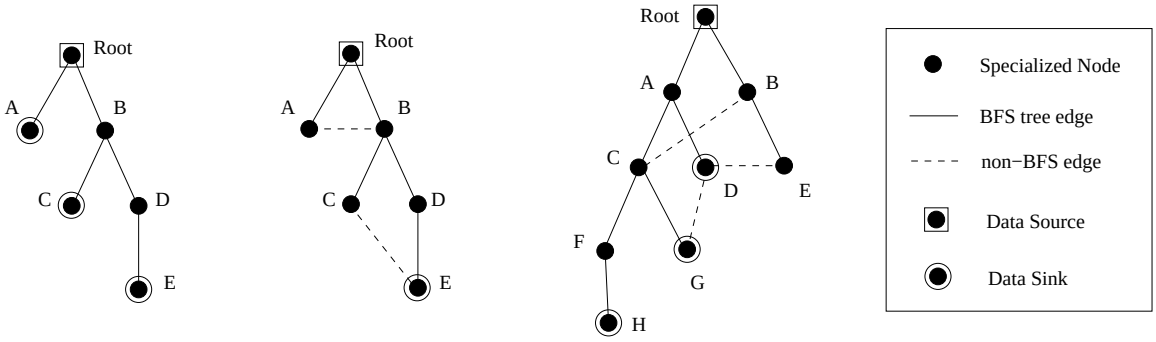


Figure 5.2: Task Graphs: (a) Tree, (b,c) Non-Tree Graphs

$S_2 = \{$`UNINSTANTIATED, WAIT_FOR_ACK, INSTANTIATED, SUBTREE_INSTANTIATED`$\}$.

A device plays the role of a "Coordinator" when it coordinates the process of instantiation of a subset of the nodes in TG. Only five of the eight possible states in $S_1 \times S_2$ are necessary in the protocol; these are depicted in the Finite State Machine diagram in Figure 5.1. All devices except the user device U start in the (`NON_COORDINATOR, UNINSTANTIATED`) state. U starts execution in the (`COORDINATOR, INSTANTIATED`) state which means that it is instantiated itself and is playing the role of a coordinator. It also means that TG has not been instantiated yet, since in that case U would be in the (`COORDINATOR, SUBTREE_INSTANTIATED`) state.

The embedding process begins at U with a distributed search which proceeds through the MANET (represented by G) hand-in-hand with a *breadth-first search* (BFS) through TG. Figure 5.2 depicts three examples of task graphs with their BFS and non-BFS edges. We call the spanning tree on TG induced by BFS and rooted at U, a BFS-tree of TG ($BFST_{TG}$). In this discussion, we assume here that all edges of TG have equal weight. If the weights are unequal, a Minimum Weight Spanning Tree, $ST_{min}(TG)$ is computed and its edges are used for performing the embedding instead of the BFS-tree. The rest of the steps are similar to those applied for embedding a BFS-tree with corresponding changes in nomenclature. In our algorithm, a coordinator device is responsible for achieving the instantiation of nodes that are its children in $BFST_{TG}$. In other words, each chosen instance

of a TG node in the network has exactly one BFS parent that acts as its coordinator. An instance of a leaf node always remains in `NON_COORDINATOR` state.

We propose a *greedy* solution to keep the average dilation of the embedding low. As mentioned in Section 3.4, the dilation metric plays an important role in affecting application performance. Hence, we focus on achieving a task embedding with low dilation. The greedy mapping algorithm begins from U by progressively mapping the nodes of $BFST_{TG}$ to nearest devices in G and the edges of $BFST_{TG}$ to shortest paths in G. Instantiation of any pair of nodes $X, Y \in V_T$ cannot affect each other if X is not a parent of Y in $BFST_{TG}$, or vice versa. Hence, the search can proceed in a distributed manner along the branches of $BFST_{TG}$. Note that unlike the optimal algorithm proposed in Section 4.2.3, this algorithm greedily instantiates nodes in TG and searches only the local space around a device for instantiating a child node in TG.

The various packet types used for the execution of the distributed instantiation protocol are summarized in Table 5.1. The task graph is encapsulated and sent as control data during the instantiation process. As the instantiation process progresses, the selected set of devices communicate reliably with one another using the underlying TCP transport layer which sits on top of the MANET routing protocol layer. TCP is used to exchange control and data packets because packet losses due to route errors are very common in MANETs [20]. As mentioned before, it is possible to use reliable UDP (with acknowledgments, timeouts and packet retransmissions) for control messaging but its impact on network congestion would then be unknown for large networks. We utilize a similar transport during the design of a proof-of-concept prototype as described in Chapter 7. The proposed instantiation protocol is independent of the underlying routing protocol; in fact, it can be executed on top of a variety of routing protocols such as DSR [46], AODV [61] or OLSR [44].

Table 5.1: Packet Types used in the Distributed Embedding Protocol

Packet Name	**Function in Protocol**
S_QRY	contains the names of required resources (*broadcast*)
C_RESP	response from a candidate device (*UDP unicast*)
ACK	from a coordinator device to a selected candidate
CONFIRM	confirmation from candidate to coordinator
SUBTREE_CONF	device confirms instantiation of subtree to its coordinator
TUPLES	instantiated tuples from parent to child node
INST_TG	instantiated TG information to children
REINST_TG	re-instantiated TG information to children
REJECT	sent in response to an ACK by an already instantiated node (to reject instantiation offer)
PREINST_HELLO	sent from instantiated parent to child before the entire instantiation process is completed
PREINST_HELLO_ACK	sent as immediate response to a PREINST_HELLO
PATCH_ACK	patch a subtree to handle failures during instantiation
TWOHOP_CONFIRM	inform about grandchildren during instantiation
PATCH_CONFIRM	confirm to parent after a subtree is patched for handling failures during instantiation phase
HELLO	exchanged between instantiated neighbors in TG
HELLO_ACK	sent as immediate response to a HELLO
NEIGHBOR_INFORM	encapsulate and send logical neighbor table to logical neighbor nodes in the instantiated TG
INFORM_MAIN_COORD	inform root node about a re-instantiation
ASK_MAIN_COORD	inquire root node about loss of non-BFS child
TASK_DATA	task application data

Choice of MANET Routing Protocol Unless the traffic load is high and rich (source-destination pairs are widely distributed across the network) and devices are highly mobile, reactive protocols such as DSR [46] and AODV [61] tend to outperform their proactive counterparts such as OLSR [45]. Under the scenarios described above, the protocol performance is poor for all protocols in general. In this work, although we simulate high mobility, our traffic patterns are generally localized owing to greedy instantiation (this will become clear soon). Hence, we use an on-demand routing protocol (DSR, specifically) as a protocol of choice for our simulation experiments. However, we realize that proactive protocols are more suitable for facilitating task-aware routing since device state can be piggybacked onto periodic link-state flood updates and can thus be exchanged easily between devices. We leave this as a topic for future research (see Chapter 9 for more details).

Details of the TG-Embedding Protocol The salient steps of the distributed version of the GreedyEmbed algorithm are illustrated in Figure 5.3 by means of a time-based message diagram for a given TG. First, TG nodes which are neighbors of the user node U are visited in parallel, and suitable instances of these nodes are discovered in the network. Node U issues broadcast search queries that are encapsulated in `S_QRY` packets for each neighbor node in the TG. For example, in Figure 5.3, U queries for devices with same attributes as those of nodes A and B in TG. The broadcast is controlled by executing it at the TG-layer rather than at the IP layer. The time-to-live (TTL) value in the `S_QRY` packet is set equal to the expected diameter of the network. Essentially, a device in the network broadcasts an `S_QRY` packet to all its one-hop neighbors that examine the state of the packet and decide whether to rebroadcast it or not. If a device decides to rebroadcast an `S_QRY` packet, it decrements the TTL value by 1 before rebroadcasting it to its neighbors. Performing the broadcast at the TG-layer instead of the IP layer yields the following advantages: (1) no special broadcast storm suppression mechanisms are necessary at the IP layer since intelligent techniques can be used for suppression of duplicates for mitigating the problem [68], and (2) a device may choose not to rebroadcast the `S_QRY` packet if it matches the query itself, even if the TTL

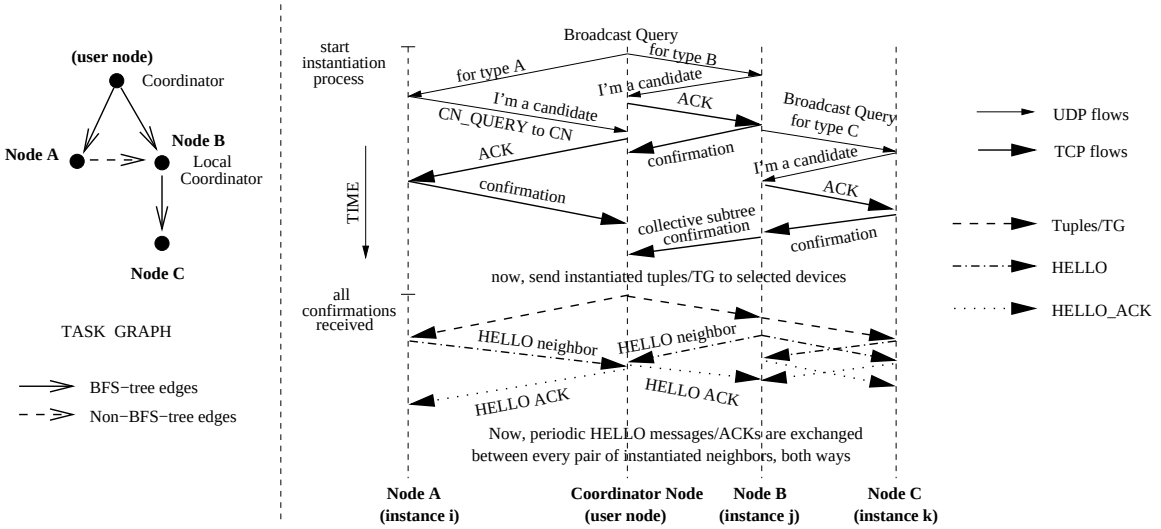


Figure 5.3: Dynamics of the Distributed Embedding Scheme

value is positive (This can be done to save bandwidth).

The `S_QRY` packet includes the principal attributes of nodes A and B. It is rebroadcast by the TG-layer of a device D if it does not match the search query type, the TTL value in the packet has not reached zero, and if D has not seen the same broadcast before.[1] Note that a query can include multiple secondary attributes of a requested node in addition to its principal attribute; for example, a query can be for `[device=printer; resolution=1200dpi; duplex=yes]`. A device which possesses capabilities that not only match the principal attribute `device=printer` but also match the other attributes mentioned in the query would respond to the query. In the following discussion, $X_{<number>}$ uniquely represents a device that matches the principal attribute of a certain TG node X.

When a free instance B_j receives an `S_QRY` packet, it sends a candidate response (`C_RESP`) packet to its coordinator (BFS parent, U) expressing its willingness to participate in the task. B_j then changes its state to `WAIT_FOR_ACK` as it waits for an acknowledgment to arrive from its coordinator. U checks the instantiation state of the TG node B and

[1] An `S_QRY` packet is identified by a broadcast sequence number that is incremented by its originator when the latter performs a new broadcast

sends an acknowledgment (ACK) to B_j if node B has not yet been instantiated in its TG. The coordinator device is responsible for accepting and rejecting responses from several candidate devices in the network before the instantiation of node B is completed. For example, suppose that two instances of node B, namely B_i and B_j, respond to U's query. If U receives a C_RESP from B_j earlier, it can decide to instantiate node B with device B_j. In that case, when the C_RESP from B_i arrives later, U ignores it since node B has already been instantiated.

Selection of a candidate device from a set of respondents can be performed on the basis of several policies. For the purpose of most discussions in this dissertation, we select the *first* candidate device that replies to an S_QRY for TG node B to become node B's instance at U. This is a simple method of selecting a nearby device in an attempt to minimize average dilation. More sophisticated policies can be used for the selection process; for example, one can attempt to maximize the energy efficiency and remaining battery lifetime of a candidate. For that purpose, the candidates need to encapsulate their current attributes in the C_RESP packet. The coordinator then determines its selection based on these attributes. Another instantiation goal may be to minimize the overlap of routes between any pair of communicating instances of TG nodes in order to reduce bottlenecks or hot-spots in the network.

Along with an ACK, U sends a copy of TG listing the node types that are in the subtree of TG rooted at node B (such as node C). U does not send explicit negative acknowledgments to all other instances of node B. Instead, the responding candidates (B_i and B_j) start respective timers as soon as they send a C_RESP packet to U. If a candidate's timer expires before it receives an ACK, it transits to the UNINSTANTIATED state again. This is more scalable from a network traffic point of view than U sending explicit negative acknowledgments to all the *unselected* respondents (such as B_i), albeit at the cost of tying them up in the WAIT_FOR_ACK state for a longer period of time. An acknowledged candidate B_j changes its state to (COORDINATOR, INSTANTIATED) and confirms its role to U by sending

a CONFIRM packet, thus completing a three-way handshake. B_j then assumes the role of a coordinator since there are uninstantiated nodes rooted at node B in TG, and it takes the responsibility of instantiating these nodes. For achieving this, B_j broadcasts an S_QRY packet containing the attributes of all uninstantiated children nodes and the instantiation thus proceeds further. Thus, B_j acts as a *local coordinator* for all nodes in the subtree of $BFST_{TG}$ that is rooted at B.

Instantiation of node B is not complete until the CONFIRM packet arrives at U from B_j. If a device is an instance of a leaf node in $BFST_{TG}$ it sends a CONFIRM packet with the SUBTREE_CONF flag *set* since it does not have to propagate the search further. This is illustrated in Figure 5.3 where C_k (which is an instance of a leaf node in TG) sends a subtree confirmation to its BFS parent and coordinator, B_j. The instantiation of node C at B_j is complete at this stage.

After receiving subtree confirmations from all selected child node instances, the parent node sends a subtree confirmation (SUBTREE_CONF) to its BFS parent (local coordinator) along with a partially instantiated task graph; in the running example, B_j sends a SUBTREE_CONF packet to U after receiving the same from C_k. Simultaneously, it also transits to the (COORDINATOR, SUBTREE_INSTANTIATED) state. On the other hand, a leaf node instance such as A_i, upon instantiation transits to the (NON_COORDINATOR, SUBTREE_INSTANTIATED) state after sending the SUBTREE_CONF upstream.

When U receives SUBTREE_CONF packets for all its children nodes in TG, it concludes the instantiation process. At this stage of the protocol, U is aware of the instances of all nodes in TG that will participate in the task. It then sends the instantiated TG and data-flow tuples to the chosen devices (node instances) so that each device knows the addresses of instances of its neighboring nodes in TG. Instead of sending the tuples separately to each instantiated device, U merely sends them to its children who then relay it downstream (along the edges of $BFST_{TG}$) to their children and so on. This reuses the routes and TCP connections that have already been established during the earlier stages of the protocol,

and thus helps to keep in check new traffic that could result from new route discovery and TCP handshaking.

An important point to be noted here is that the root coordinator U must send the instantiated TG to certain devices whose corresponding TG nodes have incoming or outgoing non-BFS edges incident upon them. For example, because the search process was carried out along the BFS tree edges of TG in Figure 5.3, B_j and A_i will not know about each other's existence unless U tells each about the other. Ideally, for any two given nodes connected by a non-BFS edge in TG, a node which is their nearest common ancestor in $BFST_{TG}$, has knowledge of their instances. However, for simplicity we delegate the responsibility of disseminating such information on the root U. For tree TGs, the above situation cannot arise, and hence U need not disseminate the instantiated TG information at all. The nature of the instantiation process ensures that each instantiated device keeps all the information that is essential to it for task data-flow, i.e. the addresses of all its parent and children nodes. Regardless of the nature of TG, each instantiated device maintains slightly more information than just parent and child addresses to be able to perform local repair after suffering disconnections. We address this issue in detail in Section 5.3.

Salient details of state transitions and control packet exchanges in our protocol are illustrated in Figure 5.1 by means of a finite state machine (FSM) diagram. It should be noted that packets arriving at a device in a state different from the intended state (in sender's view) are discarded; such events are not shown explicitly in the FSM. The user devices are best suited for acting as *root* coordinators because they usually originate the application data flows, and even under conditions of mobility, always remain near the user. Since the instantiation process is distributed among nodes in $BFST_{TG}$, the root coordinator is not overwhelmed with control packet transmissions and receptions as would be the case in a centralized approach proposed elsewhere [50].

MANET multicast protocols can play a role in this algorithm, especially for the dissemination of TG information from the coordinator after the completion of instantiation.

However, assuming the presence of multicast support in all MANETs may not be realistic; therefore, our protocol is not built with this dependency.

After the exchange of instantiation information, the user application can start data transmission. The flow of data is governed by the instances of the corresponding data-flow tuples, and in the ideal situation, all data originating at the source should reach the instances of the sink nodes in TG (A_i and C_k in the example in Figure 5.3) after being relayed by the intermediate devices (B_j). In reality, application data may not reach the intended sinks due to disconnections. We propose mechanisms to recover from such situations in Section 5.3.

5.2 Instantiation of Non-Heterogeneous Task Graphs

In the previous section, we presented the basic instantiation protocol that is used when TG is comprised of nodes with distinct attributes. However, some steps of that protocol pose problems when there is a large number of homogeneous TG nodes. We illustrate the situation by a simple example. Consider a homogeneous task graph represented as a line graph with 4 nodes with the user node at the root: U–X–X–X. The user node queries for its adjacent node X by flooding the network with an `S_QRY` packet with a high TTL value. The request can propagate far even if the user node has received `C_RESP` packets from many candidates. Although this problem existed in the heterogeneous case as well, it is aggravated in the homogeneous case in the following manner: if the MANET has many instances of resource X, all of them can respond to the `S_QRY` and can cause significant redundant traffic in the network.

Another problem associated with the protocol steps defined in the previous section can be illustrated as follows: all devices which respond to a query enter a `WAIT_FOR_ACK` state for time W just after sending a `C_RESP` packet. In that state, they do not respond to future queries until the W timer expires and they re-enter the `UNINSTANTIATED` state.

Hence, after U instantiates node X to device x_i, the latter queries for node X again but gets no response from any candidate for a while. This is because the other instances of node X have been *bound* unnecessarily for time W by the earlier query from U. This phenomenon can significantly increase the total instantiation time (for the entire TG) even in a static network.

The first problem can be solved by adopting expanding ring search. We use this by starting with TTL = 1; the TTL is then incremented systematically until at least one device responds to the query. This increment is performed only after waiting for a fixed delay for a response to come back. In the simulation studies performed in Section 6.2, we increment the TTL by one after waiting for τ seconds. More aggressive schemes such as multiplicative increments are also possible.

The second problem can be solved by a more fundamental modification of the protocol. To this end, we deduced that making the instantiation phase triggered by packet receptions and not getting "bound" in the `WAIT_FOR_ACK` state for a significant amount of time can help reduce instantiation time as well as the average dilation D_{avg} of the embedding. However, the benefits arrive at the expense of extra protocol traffic. When the `WAIT_FOR_ACK` state is removed, a device X_i stays in the `UNINSTANTIATED` state even after responding to a query. It is also enabled to respond to multiple queries from different sources while remaining in that state until it receives an `ACK` from a particular source. Immediately after this event, it transits to the `INSTANTIATED` or the `SUBTREE_INSTANTIATED` state depending on whether it assumes the role of a non-leaf node or a leaf node in the task graph, respectively. It then sends a `CONFIRM` or a `SUBTREE_CONF` packet to the parent accordingly. Note that X_i had also responded to some other search queries before, hence it could receive `ACK` packets from some of these other devices. Since X_i can assume the role of only one TG node in our model, and it already has done that, it responds to those particular sources with explicit `REJECT` packets. This essentially *rejects* the instantiation request from the source and instructs it to continue instantiation. The pseudo-code for

the revised instantiation process is shown in Algorithm 5.1 (for a coordinator device) and Algorithm 5.2 (for a non-coordinator device) and associated functions 5.3–5.9.

Function 5.9 (QueryForNeighborInstances) is called by a coordinator device for instantiation of nodes in the subtree of $BFST_{TG}$ for which it is root. If there are multiple children nodes in $BFST_{TG}$ with same attributes, only *one* `S_QRY` packet is broadcast since multiple candidates will hear and respond to the query.

Function 5.3 (HandleTTLTimeout) is called whenever a TTL timer which had been scheduled after an `S_QRY` was issued, expires. A TTL timer can expire at a device only when there exists a child node in $BFST_{TG}$ for which no candidate has responded yet before the timeout τ. For example, if node S has two children of type X and if only one `C_RESP` arrives at S_k (an instance of S) within timeout period τ, then the corresponding TTL timer is not canceled but is allowed to expire after τ. At that instant, S_k broadcasts another query for node X with a higher TTL (expanding ring search), and the TTL timer is rescheduled. Only when both children of type X have been responded for does S_k cancel the TTL timer.

A candidate device responds to an `S_QRY` packet with a `C_RESP` only when it is in the `UNINSTANTIATED` state. Otherwise it merely rebroadcasts the query. As shown in function 5.4, a candidate device records the time at which it responds to a certain `S_QRY`, and it does not respond to subsequent queries from the same source that arrive within `ACK_PERIOD` (which is a small multiple of τ). This step is taken for allowing the expanding ring search to spread effectively without duplicating network traffic. We illustrate this with an example. Consider the earlier example of S having two children of type X in its $BFST_{TG}$. Suppose S has one instance x_1 of X within one hop and another one x_2 which is within two hops. x_1 responds to the first query with TTL = 1 but after τ, S determines that it has only received one response (which it has `ACK`-ed), and broadcasts another query for X with TTL = 2. Now, x_1 which has not received the `ACK` yet (perhaps due to TCP delays) receives this second query within `ACK_PERIOD` of the first query. So it merely rebroadcasts the query to

the second hop. Had x_1 responded to the second query, it would have duplicated network traffic. This mechanism can be particularly helpful for the instantiation of large TGs in dense networks.

The handling of a `C_RESP` packet at a coordinator device is shown in function 5.5 (HandleCandidateResponse). If any uninstantiated and un-`ACK`ed children nodes of type *dt* (principal attribute of respondent) remain, the coordinator sends an `ACK` to the respondent. If not (this means that the coordinator has already sent an `ACK` for this node), then the respondent's address is cached in a queue *RespQ* for future use. This is useful because with a certain probability depending on the network topology and the distribution of resource types in the network, the coordinator may get a `REJECT` from a candidate device that has already been instantiated by another coordinator. In such a situation, the former coordinator has to continue instantiation of that particular node. Function 5.7 (HandleReject) shows how a coordinator handles this. If *RespQ* is empty, then the coordinator has to perform a broadcast with an appropriately large TTL in order to get more responses from candidates. But if *RespQ* is not empty, the coordinator can send `ACK`s to the candidate devices in *RespQ* one by one. In the worst case, all the devices in *RespQ* may have been already instantiated by other coordinators but the probability of that occurrence is likely to be low. Although we have not mentioned this in the pseudo-code, one can keep the size of *RespQ* upper bounded by a value characteristic of the distribution of resource types in the network. Doing that will prevent the coordinator from sending a series of unnecessary `ACK`s to instantiated devices.

The final few steps of the instantiation process are the same as the ones mentioned in Section 5.1. If an uninstantiated candidate device receives an `ACK` from a coordinator, it extracts the task graph encapsulated in the packet into a local variable TG. After instantiating the appropriate node, it computes $BFST_{TG}$ and determines whether it needs to act in the capacity of a leaf node or a non-leaf node. If the former is true, then it sends a `SUBTREE_CONF` to its parent coordinator, thus confirming its instantiation. However, if the

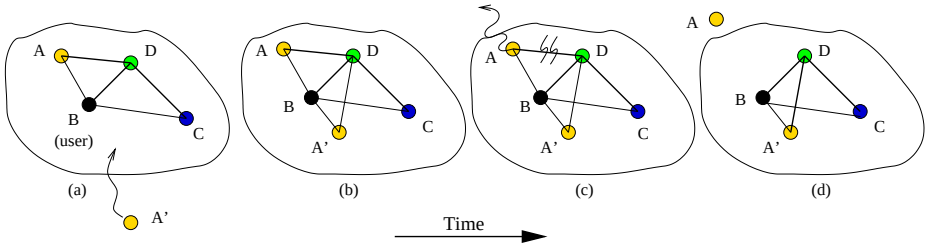


Figure 5.4: An Example of Re-Instantiation

latter is true, then it needs to continue the distributed instantiation process further. After confirming its instantiation to the coordinator by means of a `CONFIRM` packet, the candidate makes itself a coordinator by transiting to the `(COORDINATOR, INSTANTIATED)` state. It then continues the instantiation process by querying for its uninstantiated children nodes. Function 5.6 (HandleAck) shows the pseudo-code for the above steps.

Any coordinator device after receiving a `SUBTREE_CONFIRM` checks if all its BFS children have sent such confirmation; in that case, it sends one `SUBTREE_CONFIRM` to its parent coordinator. When the root coordinator (i.e., the user device) receives the final `SUBTREE_CONFIRM` packet, it deems the instantiation process to be complete. At this instant, the user node has successfully mapped each node in TG to a physical device in the network using the distributed algorithm. The user application can then begin.

5.3 Handling Device Mobility

In this section we present how our instantiation protocol reacts to the mobility of devices. If the devices in the network are highly mobile during the lifetime of a distributed application running on the network, the network topology and previously established connections can change, and this can disrupt the application either due to unreachability of devices or due to frequent route changes. Therefore, in the presence of mobility, it is no longer sufficient to permanently select specific devices and appoint them to execute the application – continual monitoring must be performed for detecting disruptions, and replacement devices must be

Algorithm 5.1 CoordinatorDeviceInstantiation()

```
1: mystate ← (COORDINATOR, INSTANTIATED);
2: QueryForNeighborInstances();                                  /* function 5.9 */
3: while (1) do
4:    WaitForEvent();            /* returns at the occurrence of an event */
5:    if (T_A.expired()) then
6:       HandleTTLTimeout(T_A);                                  /* function 5.3 */
7:    else if (C_RESP arrived from a_i) then
8:       HandleCandidateResponse(a_i.addr, a_i.type);            /* function 5.5 */
9:    else if (CONFIRM arrived from a_i) then
10:      HandleConfirmation(a_i.addr, a_i.type, a_i.tgid);       /* function 5.8 */
11:   else if (REJECT arrived from a_i) then
12:      HandleReject(a_i.addr, a_i.type, a_i.tgid);             /* function 5.7 */
13:   end if
14: end while
```

Algorithm 5.2 NonCoordinatorDeviceInstantiation()

```
1: mystate ← (NON_COORDINATOR, UNINSTANTIATED);
2: while (1) do
3:    WaitForEvent();            /* returns at the occurrence of an event */
4:    if (S_QRY(qdt) arrived from c_i) then
5:       HandleSearchQuery(c_i.addr, qdt);                       /* function 5.4 */
6:    else if (ACK arrived from c_i) then
7:       HandleAck(c_i.addr, dt, tgid);                          /* function 5.6 */
8:    end if
9: end while
```

function 5.3 HandleTTLTimeout(TTLTimer T)

```
1: T.ttl← f(T.ttl);                          /* f(x) = x + 1 or 2 * x etc.  */
2: Broadcast(S_QRY,T.dt,T.ttl);     /* query for type:T.dt with higher TTL */
3: T.resched(τ);                                  /* reschedule TTL timer */
```

function 5.4 HANDLESEARCHQUERY(**address** c, **type** Qdt)

```
1: if (mytype == Qdt) then
2:   if (time_now − lastresponded[c] ⩽ ACK_PERIOD) then
3:     REBROADCASTPKT(); return;                          /* if TTL permits */
4:   end if
5:   SEND_UDP_PKT(C_RESP, c, mytype);                    /* respond to c over UDP */
6:   lastresponded[c] ← time_now;                        /* suppresses duplicate C_RESPs to c */
7: else
8:   REBROADCASTPKT();                                   /* rebroadcast query if no match */
9: end if
```

function 5.5 HANDLECANDIDATERESPONSE(**address** a, **type** dt)

```
1: RespQ(dt).append(a);                                  /* cache respondent's address for future */
2: for all (X : X is an uninstantiated BFS child in TG and X_dt == dt) do
3:   if (acked[X_tgid] == 0) then
4:     SEND_PKT(ACK, a, X_tgid);                         /* send an ACK to the respondent */
5:     acked[X_tgid] ← a;                                /* mark corresponding TGID as acked */
6:     RespQ.pop();                                      /* pop a from RespQ since it has been acked */
7:     if (all children of type dt have been responded for) then
8:       T_dt.CANCEL();                                  /* cancel corresponding TTL timer */
9:     end if
10:    break;                                            /* break out of for loop */
11:  end if
12: end for
```

function 5.6 HANDLEACK(**address** c, **type** dt, **tgid** $tgid$)

```
1: if (mystate == (NON_COORDINATOR, UNINSTANTIATED)) then
2:   lastresponded ← nil;
3:   TG ← EXTRACTTASKGRAPH();            /* for downstream instantiation */
4:   Instantiate myself in TG for tgid;
5:   COMPUTEBFSTREE(TG);      /* compute BFST_TG and my position in it */
6:   if (!ISLEAFNODE(me, TG)) then
7:     SEND_PKT(CONFIRM, c, tgid);                 /* If I am not a leaf node */
8:     mystate ← (COORDINATOR, INSTANTIATED);          /* become COORDINATOR */
9:     QUERYFORNEIGHBORINSTANCES();          /* downstream instantiation */
10:  else
11:    SEND_PKT(SUBTREE_CONF, c, tgid);                  /* I am a leaf node */
12:    mystate ← (NON_COORDINATOR, SUBTREE_INSTANTIATED);
13:  end if
14: else if (mystate == (COORDINATOR, INSTANTIATED)) then
15:  SEND_PKT(REJECT, c, tgid);        /* already instantiated, hence REJECT */
16: end if
```

function 5.7 HANDLEREJECT(**address** a, **type** dt, **tgid** $tgid$)

```
1: acked[tgid] ← 0;                                          /* unmark this TGID */
2: a' ← RespQ(dt).pop();              /* use addresses of earlier respondents */
3: if (a' ≠ nil) then
4:   SEND_PKT(ACK, a', tgid);            /* send ACK pkt to the new candidate */
5:   acked[tgid] ← a';                                  /* mark this TGID as acked */
6: else
7:   BROADCAST(S_QRY, dt, TTL+1);          /* query for type:dt with TTL+1 */
8:   T_dt.RESCHED(τ);                                  /* reschedule TTL timer */
9: end if
```

function 5.8 HANDLECONFIRMATION(**address** a, **type** dt, **tgid** $tgid$)

1: **if** (`CONFIRM` packet) **then**
2: Instantiate $tgid$ in TG with a;
3: **else if** (`SUBTREE_CONF` packet) **then**
4: `/* extract downstream instances and instantiate in` TG `*/`
5: $TG_{inst} \leftarrow TG_{inst} \cup$ EXTRACTSUBTREEFROMPKT();
6: **if** (all BFS children have confirmed) **then**
7: $mystate \leftarrow$ (`COORDINATOR`, `SUBTREE_INSTANTIATED`);
8: **if** ($me \neq$ ROOT(TG)) **then**
9: `/* encapsulate instantiated subtree rooted at` me `in` TG_{inst} `and send subtree confirmation upstream to my BFS parent */`
10: $S_{buf} \leftarrow$ ENCAPSULATEINSTSUBTREE(me);
11: SEND_PKT(`SUBTREE_CONF`, BFS_PARENT, S_{buf});
12: **else**
13: `/* Instantiation of` TG `has been completed; */`
14: Disseminate Instantiated `TUPLES` to instantiated BFS children;
15: **end if**
16: **end if**
17: **end if**
18: Start exchanging `PREINST_HELLO` or `HELLO` messages with a;

function 5.9 QUERYFORNEIGHBORINSTANCES()

1: TTL$\leftarrow$ 1;
2: **for all** ($X : X$ is an uninstantiated BFS child in TG) **do**
3: BROADCAST(`S_QRY`,X,TTL); `/* query for type:`X `with` TTL `= 1 */`
4: queue $RespQ(X) \leftarrow$ **nil**;
5: T_X.SCHED(τ); `/* schedule a` TTL `timer` T_X `for` τ `seconds */`
6: **end for**

selected for resuming the application. Figure 5.4 illustrates the process of re-instantiation in the case of disconnections caused by mobility of devices. The task graph in that example is $V_T = \{user, \alpha, \gamma, \delta\}; E_T = \{(user, \delta); (\delta, \alpha); (\delta, \gamma)\}$; B is the user device, A and A' are instances of α, C is an instance of γ, and D of δ.

Disruptions can occur either during the instantiation phase or in the post-instantiation phase. In the latter case, all nodes in TG were mapped to a set of physical devices but a subset of those devices may have been disconnected from the rest of the instantiated set. In this scenario, the onus of rediscovering new candidate devices and remapping relevant pieces of the distributed application falls onto the currently instantiated devices. The situation is similar if disconnections occur before the instantiation has been completed, only the number of instantiated nodes is smaller.

The necessity of self-organizing instantiation protocols that adapt to changes in network topology at any stage of operation cannot be overemphasized. Due to the unique nature of MANETs, device mobility is beyond the user's control in most situations and can indeed cause disruption of service. Hence our foremost goal is to design robust protocols that react to such events and attempt recovery. Another important design goal is to eliminate the involvement of a centralized controller as far as possible while not unduly complicating the recovery protocol. A centralized recovery approach has been proposed by us elsewhere [50].

5.3.1 Detection of Disconnections

Mobility of devices can cause network partitions or disconnections resulting in instantiated devices that may no longer be able to communicate if their interconnection paths are broken. In such situations, replacement devices must be instantiated in lieu of their disconnected counterparts. The necessary first step in this direction is the *detection* of which device(s) is/are disconnected from which other device(s).

We propose a lightweight, soft-state exchange protocol for detecting disconnections

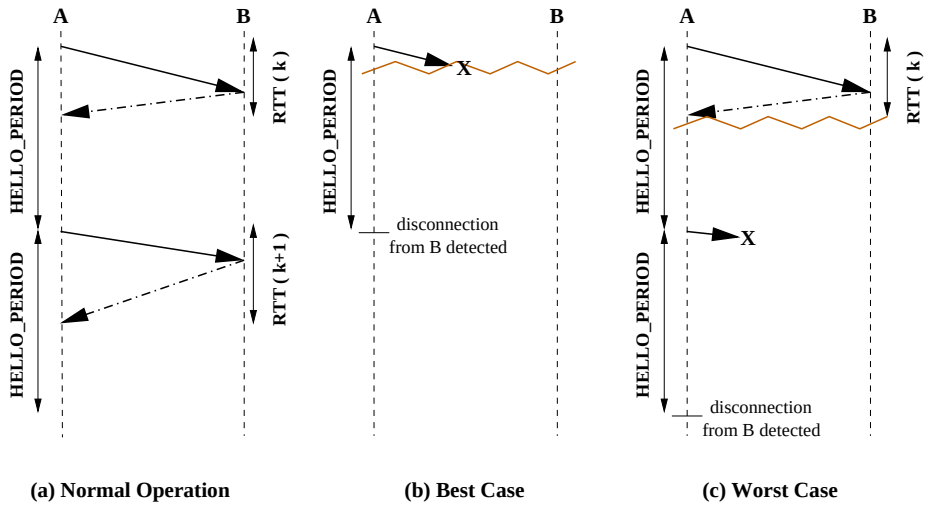


Figure 5.5: Detection of Disconnections: A and B are Parent–Child Instances

in an instantiated task graph. The protocol requires each instantiated device to send *periodic* `HELLO` messages (with period T) to its logical neighbor instances in TG, which reply with a `HELLO_ACK`. This is illustrated in Figure 5.6. Note that all instantiated devices are in either of the two states when they start exchanging `HELLO` messages: (`COORDINATOR, SUBTREE_INSTANTIATED`) or (`NON_COORDINATOR, SUBTREE_INSTANTIATED`).

Each instantiated device keeps track of its BFS parent and BFS children. For the TG in Figure 5.2(c), the instance of node C keeps track of the instance of node A (its BFS parent) as well as of the instances of nodes F and G (BFS children). If a BFS parent device stops hearing from one of its BFS children,[2] it *uninstantiates* this child from its TG and initiates search for a replacement of the same type. The child meanwhile stops hearing `HELLO_ACK`s from the parent (assuming bidirectional links), and *uninstantiates* itself at the end of two `HELLO_PERIOD`s. This is illustrated in Figure 5.6. If the `HELLO_PERIOD` timer is set to H at every instantiated device, on average, disconnections will be detected after $\frac{3}{2}H$ time units, approximately. Function 5.10 (HandleDisconnects) lists the pseudo-code for detecting disconnections from instantiated neighbor devices.

[2]The parent concludes this if it does not get a `HELLO_ACK` from that child before the expiry of its `HELLO_PERIOD` timer.

Theorem 5.1 (Average Detection Time) *The average time to detect a disconnection due to a persistent partition is* $\frac{3}{2}H - r$ *where* H *is the* `HELLO_PERIOD` *and* r *is the average time for the round-trip exchange of* `HELLO` *messages.*

Proof. Figure 5.5 illustrates the three cases of normal operation without any disconnection, the best case, and the worst case for detecting disconnections respectively. Suppose that a device A wants to detect if its association with device B has been disrupted. A then sends a `HELLO` packet to B which then immediately responds with a `HELLO_ACK`. Suppose r is the average time for this exchange to occur. r is dependent upon the number of hops A and B are from each other and on the dynamics of the underlying routing as well as transport protocols. As shown in Figure 5.5(b), if the actual disconnection occurs just after the `HELLO` packet is transmitted, then it will not reach B and hence A can detect the disconnection in H. However, if the disconnection happens just after A receives a `HELLO_ACK` packet, then it will not be able to detect the disconnection at the end of H. In fact it will take an additional H to detect the disconnection if the partition still exists. Now, if the disconnection happens at time t starting from the beginning of the `HELLO_PERIOD` in consideration, the time to detect the disconnection using the aforementioned scheme is given by:

$$T_{detect}(t) = \begin{cases} H - t & : \quad 0 \leqslant t < r \\ 2H - t & : \quad r \leqslant t < H \end{cases} \tag{5.1}$$

If the disconnection time t is uniformly distributed over the interval $[0, H]$, the p.d.f. of the disconnection time is given by: $f(t) = \frac{1}{H}, 0 \leqslant t \leqslant H$. Then the mean time to detect the disconnection can be easily calculated as follows:

$$\begin{aligned} E[T_{detect}] &= \int_0^H T_{detect}(t)\, f(t)\, dt \\ &= \frac{1}{H}\left\{\int_0^r (H - t)\, dt + \int_r^H (2H - t)\, dt\right\} \\ &= \frac{1}{H}\left\{\frac{3}{2}H^2 - Hr\right\} = \frac{3}{2}H - r \end{aligned} \tag{5.2}$$

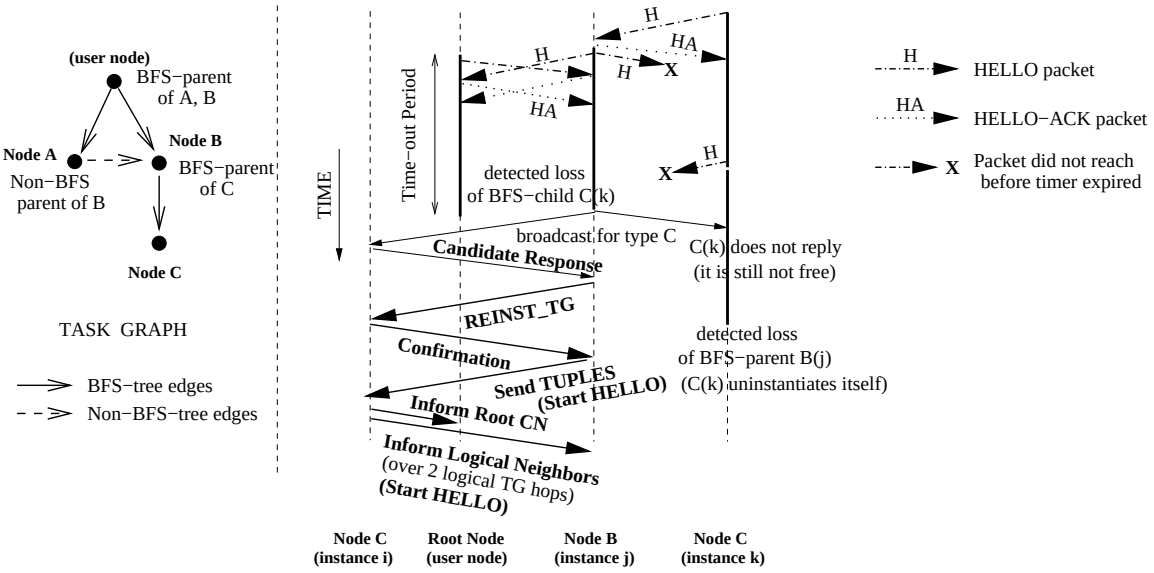


Figure 5.6: Re-instantiation of Task Graph Nodes

However, we note that this is only true for *persistent* partitions. In other words, if a partition heals during the detection process (due to node mobility), calculation of $E[T_{detect}]$ is much more difficult and not considered here. In other words, Equation 5.2 gives an upper bound unless the partition gets worse. □

Mobility of devices can also result in lengthening or shortening of routes between device instances because the underlying MANET routing protocol can adapt to the changes in network topology due to mobility. In an ideal scenario, if there is no disconnection or network partition, the application should proceed without disruption. But such ideal conditions may not hold in reality. Route failures can trigger route discovery which along with TCP re-transmissions (after timeouts) can sometimes take several seconds to complete. Hence, this can result in `HELLO_ACK`s not returning within H. This in turn causes a device X that is expecting a `HELLO_ACK` from device Y to conclude that a disconnection has happened even when X and Y are in fact reachable from one another.

Recently, researchers have recognized problems with TCP behavior in MANETs. Holland and Vaidya [42] found that TCP throughput drops significantly when nodes move,

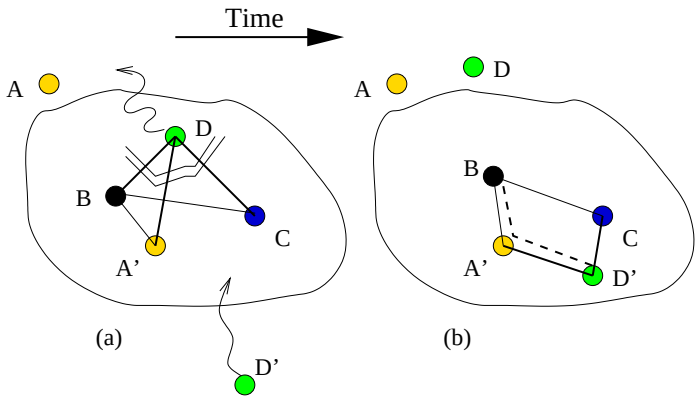


Figure 5.7: An Example of TG-patching

due to TCP's inability to recognize the difference between link or route failures and congestion. Fu et al. discovered that even in static ad hoc networks, packet losses due to channel contention affects TCP performance significantly [33]. Some solutions to the first phenomenon have been proposed in the literature that are based on explicit notification of route errors to TCP [22]. However, the above endeavors are orthogonal to our goals, and we do not attempt to alter TCP or the MANET routing protocols (nor their default timer settings); we instead build our protocols simply on top of these protocols. Hence, if a device does not receive a `HELLO_ACK` from its neighbor in H, we deem the neighbors to be disconnected from each other. A reasonable value of H is one which is not so low to cause significant control overhead, and not so high such that disconnections are not quickly detected. In our simulations, we experimented with $H = 7$ seconds and $H = 14$ seconds (both greater that the default TCP re-transmission timer) and found them to be suitable for the mobility patterns that we investigated.

5.3.2 Re-instantiation and Bookkeeping Algorithms

A disconnection due to mobility or device failure can occur at two different stages of protocol execution: (1) during the instantiation phase, and (2) after the instantiation has been completed. Although the two above scenarios are conceptually the same, there are subtle

differences between the two from a protocol standpoint. In this section, we introduce the basic techniques for achieving re-instantiation once instantiation has been completed. The principal goal in this endeavor is to retain most of the instantiation below the point of failure and re-instantiate the task graph minimally, whenever possible. We refer to this "local repair" process as *TG-patching* and advocate it instead of the re-instantiation of the entire task graph below the point of failure. TG-patching has been illustrated with a simple example in Figure 5.7 continuing the example shown in Figure 5.4.

If a BFS parent device detects a disconnection with its child at the expiry of its `HELLO` timer, it issues a broadcast search query for the same type of child. The re-instantiation process then proceeds much like the earlier instantiation process. This has been illustrated with a time based message diagram in Figure 5.6. Also, the root coordinator device is not involved in this process, and the BFS parent device acts like a *local* coordinator. However, after the completion of re-instantiation, the newly re-instantiated device informs the root coordinator about the event directly with an `INFORM_MAIN_COORD` packet. The reasons for doing this will become clear later in this section.

Our protocol addresses the problems of state maintenance after disconnections, and attempts to solve them locally, i.e., without involving the coordinator node. Each instance of a certain node X in TG is aware of the addresses of the instances of the parents (both BFS and non-BFS), the children, the children's parents, and the children's children of X in TG. The devices in the last two categories are referred to as siblings. Essentially, this is 2-hop logical neighborhood information save the information about the addresses of the grandparents of X in TG, which is unnecessary.

Definition 5.1 (Logical Neighborhood) If X is a node in a task graph TG, i.e., $X \in V_{TG}$, its neighborhood with respect to the BFS ordering of TG starting from the root is defined as follows:

$$TG\_ngb(X) \stackrel{\triangle}{=} X \cup parents(X) \cup children(X)$$

If x is an instance of a node X, the logical neighborhood of x is defined as:

$$logical\_ngb(x) \stackrel{\triangle}{=} \{z : z \in \varphi(Z), Z \in TG\_ngb(X)\}$$

where φ is the embedding function as defined in Chapter 3. Essentially it is the set of devices in the network which are instances of nodes in the neighborhood of X.

Definition 5.2 (2-hop Logical Neighborhood) If X is a node in a task graph TG, i.e., $X \in V_{TG}$, its 2-hop logical neighborhood is defined as:

$$2hop\_TG\_ngb(X) \stackrel{\triangle}{=} TG\_ngb(X) \cup children(children(X)) \cup parents(children(X))$$

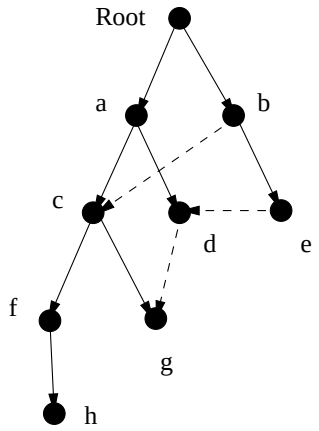
If x is an instance of a node X, the 2-hop logical neighborhood of x is defined as:

$$2hop\_logical\_ngb(x) \stackrel{\triangle}{=} \{z : z \in \varphi(Z), Z \in 2hop\_TG\_ngb(X)\}$$

where φ is the embedding function as defined in Chapter 3.

Figure 5.8 depicts the 2-hop logical neighborhood information that each instantiated device is aware of at a certain instant of time. With a little thought, one can see that information about this portion of TG is sufficient to handle single node disruptions in the instantiated task graph. We explain this further with an example. After finding a new replacement device for its child node, a BFS parent re-instantiates it in its own copy of TG, and sends a portion of the re-instantiated TG to the new child device. Figure 5.9 illustrates how 2-hop logical neighborhood information helps in local recovery. $a, b, \ldots, h$ are initial device instances of their respective node types, and g' is a replacement device with same capabilities as g.

Function 5.11 (InformLogicalNeighbors) encapsulates the *logical_ngb* information at a device and sends it to the other devices in its logical neighborhood. The originator of the logical neighbor table update encodes $round = 1$ into the `NEIGHBOR_INFORM` packet before calling InformLogicalNeighbors(). When a device in the logical neighborhood receives this packet (as shown in Function 5.12 (LogicalNeighborUpdate), it updates its



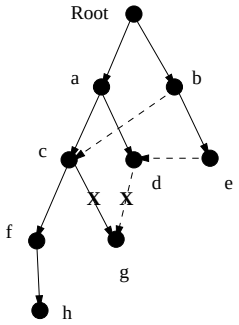
Node	Parents	Children	Grandchildren	Siblings
Root	–	a,b	c,d,e	–
a	Root	c,d	f,g	b
b	Root	{c},e	d,f,g	a
c	a,{b}	f,g	h	d
d	a,{e}	{g}	–	c
f	c	h	–	–
g	c,{d}	–	–	–
h	f	–	–	–

Figure 5.8: Logical Neighbor Table Information

logical_ngb, increments *round* to 2 and sends the updated logical neighbor table to the other devices in *logical_ngb*. When devices receive a `NEIGHBOR_INFORM` packet with $round = 2$, they update their $2hop\_logical\_ngb$ structure accordingly. At the end of this exchange, nodes in the logical neighborhood of a device possess a consistent view of other instantiated devices in the neighborhood. This exchange of NEIGHBOR_INFORM messages occurs over TCP because these can traverse multiple hops and their loss can cause inconsistencies and therefore delays in the recovery process. Also, any additional disconnections *during* this exchange can cause inconsistency and that will be detected by `HELLO` messaging at a later time.

Lemma 5.1 (Recovery from Single Disconnects) *If there is a persistent single device disconnection, maintenance of 2-hop logical neighborhood information at each instantiated device is sufficient to accomplish re-instantiation of the task graph locally (TG-patching), and to make it resilient to single device disconnections in future.*

Proof. Since every device has exactly one BFS parent, upon the disconnection of a single device s from its parents and children, $p = parent_{bfs}(s)$ is the one which detects the disconnection and attempts to rediscover a replacement device for the corresponding task graph node. After p discovers and instantiates a new device s', the latter needs to know about the

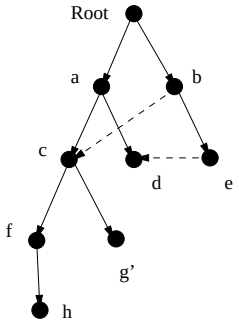


g becomes disconnected from **c** and **d**

c detects disconnection (after HELLO timeout)

c searches for a new instance of **g**

d does not take any active step (non-BFS parent)



c discovers and re-instantiates **g'**

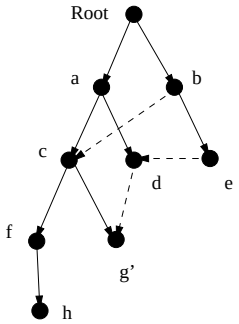
g' does not know about **d** yet

c sends the 2-logical-hop portion of its TG to **g'**

g' knows about **d** now

d does not know about **g'** yet

g' 1-logical-hop broadcasts its logical neighbor table (lgNT)



d knows about **g'** now

c knows that **g'** knows about **d**

c and **d** 1-logical-hop broadcast their lgNT's

a,**b**, and **e** come to know about **g'**

(This is useful if **c** or **d** suffer disconnection later)

Figure 5.9: Bookkeeping using 2-hop Logical Neighborhood Information

function 5.10 HANDLEDISCONNECTS(list[**node**] *parent*, list[**node**] *child*)

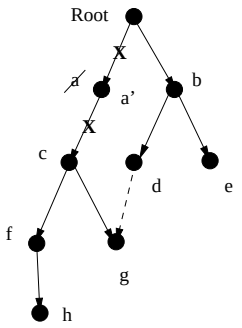
1: **if** ($reachable[parent_{bfs}.addr] == 0$) **then**
2: Uninstantiate myself; `/* I have been disconnected from my parent */`
3: **else**
4: **if** ($\forall i : reachable[parent_{nonbfs}[i].addr] == 0$) **then**
5: Do nothing; `/* I am the child; Non-BFS parent will react to this */`
6: **end if**
7: **if** ($\forall j : reachable[child_{bfs}[j].addr] == 0$) **then**
8: Uninstantiate $child[j]$ in TG and deactivate corresponding data-flow tuples;
9: REDISCOVER($child[j].dt, child[j].tgid$); `/* re-instantiate BFS child */`
10: **end if**
11: **if** ($\forall k : reachable[child_{nonbfs}[k].addr] == 0$) **then**
12: Deactivate corresponding data-flow tuples;
13: Ask Root Coordinator device about the newer instance of $child[k]$;
14: **end if**
15: **end if**

function 5.11 INFORMLOGICALNEIGHBORS(**int** *round*)

1: SENDPKT(`NEIGHBOR_INFORM`, $logical\_ngb(me)$, *round*);

function 5.12 LOGICALNEIGHBORUPDATE(**int** *round*)

1: RECEIVEPKT(); `/* receive a logical neighborhood update */`
2: **if** ($round == 1$) **then**
3: Update $logical\_ngb(me)$; `/* update my logical neighborhood */`
4: INFORMLOGICALNEIGHBORS($round + 1$); `/* propagate logical ngb. info */`
5: **else if** ($round == 2$) **then**
6: Update $2hop\_logical\_ngb(me)$ according to rules in Definition 5.2;
7: **end if**



a becomes disconnected from the rest of the network

Root detects disconnection from **a** (after `HELLO` timeout)

Root searches for and instantiates **a'**

c detects disconnection from **a** (after `HELLO` timeout)

But **c** waits for another `HELLO_PERIOD` before uninstantiating

Root sends **a'** patching information (2-hop lgNT)

a' patches **c** and hence the subtree rooted at it

Figure 5.10: TG-patching an Instantiated Task Graph

children and other parents that its predecessor s used to have. Since p keeps partial 2-hop logical neighborhood information, it is aware of $children(s)$, and $parents_{nonbfs}(s)$. Hence s' will be able to get that information readily from p after the re-instantiation is complete. As a result, s' will be capable of recovering successfully in future from disconnections of its children, if any.

Also, since s' will get non-bfs parent information as well from p, it can inform these devices about its new existence. These devices can update their 2-hop logical neighborhood tables upon receiving updates from s' and can then resume sending application data to s'. Therefore, this 2-hop logical neighborhood information facilitates the execution of the entire book-keeping process in a cooperative manner. Also, this process is local in a TG sense. Figure 5.10 illustrates the steps in the process of TG-patching. □

5.3.3 Handling Multiple Disconnections

Now we present how our re-instantiation schemes attempt to recover from multiple simultaneous node disconnections that can occur in an instantiated task graph. Even though the disconnections happen simultaneously, they may be detected at different times owing to different start and finish times of `HELLO_PERIOD`s at different devices.

Definition 5.3 (Independent Disconnects) If instantiated devices $x_1, x_2, \ldots, x_k$ (cor-

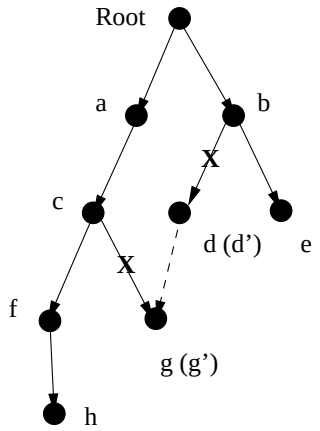
responding to nodes $X_1, X_2, \ldots, X_k \in V_{TG}$) become disconnected from their respective BFS parents simultaneously, and all subtrees in $BFST_{TG}$ rooted at X_i's are disjoint, we categorize this as an independent multiple disconnect.

Definition 5.4 (Affecting Disconnects) If instantiated devices $x_1, x_2, \ldots, x_k$ (instances of nodes $X_1, X_2, \ldots, X_k \in V_{TG}$) become disconnected from their respective BFS parents simultaneously, and $\exists i, j : X_i$ is an ancestor of X_j in $BFST_{TG}$, we categorize this situation as an affecting multiple disconnect.

Lemma 5.2 (Recovery from Independent Disconnects) *In case of independent disconnects, recovery can be performed as if there are independent single disconnects.*

Proof. Consider any two (of k) disconnected nodes X and Y in TG which have been categorized as independent. Then the subtrees rooted at them respectively (in $BFST_{TG}$) are disjoint. Therefore, either they have the same BFS parent or the subtrees rooted at their BFS parents are disjoint too. In either case, the process of recovery at $parent_{bfs}(X)$ is not affected by that at $parent_{bfs}(Y)$. Hence this can be treated as a case of independent single disconnects, and can be solved by TG-patching.

However, there is one type of disconnection that cannot be adequately handled by our re-instantiation technique even if the disconnection is *independent* in nature: a network partition such that two nodes connected by a non-BFS edge get disconnected from their BFS parents; (e.g., in Figure 5.11, if d and g get disconnected from b and c respectively, in roughly the same `HELLO` period). b finds d', and c finds g' but b tells d' about g and c tells g' about d (from their 2-hop logical neighbor tables). Due to the staleness of 2-hop logical neighborhood information, a deadlocked situation occurs in which d' and g' do not come to know about each other. To solve this problem, we adopt an approach where a non-BFS parent (d' here) asks for the root coordinator's help if it loses its non-BFS child (g'). Since the coordinator is kept informed by newly instantiated nodes (by an `INFORM_MAIN_COORD` message), situations like above can be avoided. We illustrate this process further in Figure 5.12.



b detects disconnection from **d** (after `HELLO` timeout)

b searches for and instantiates **d'**

b tells **d'** about non-BFS child **g** (*stale*)

c detects disconnection from **g** (after `HELLO` timeout)

c searches for and instantiates **g'**

c tells **g'** about non-BFS parent **d** (*stale*)

Hence, **d'** and **g'** do not come to know of each other!

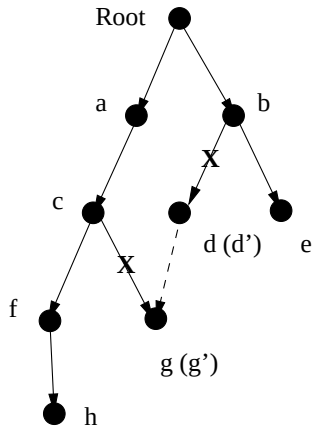
Figure 5.11: Effect of Double Disconnect on a Non-BFS Edge

Note that if the coordinator is unreachable from this node instance, it is due to a network partition alone (since we do not have any timers for such messages), and in this case, a component of the instantiated network is disconnected from the rest (containing the coordinator). In such situations, repair occurs as soon as the `HELLO` timers expire at BFS-parent node instances. □

Definition 5.5 (Burst Disconnects) When two of the affecting nodes whose instances become disconnected from each other are in a parent–child relationship, we refer to the situation as a burst disconnect. All other types of affecting disconnects are categorized as non-burst.

Lemma 5.3 (Recovery from Non-Burst Affecting Disconnects) *Recovery from an affecting disconnect which is devoid of bursts is possible by TG-patching.*

Proof. Figure 5.13(a) illustrates this scenario and how the TG-patching algorithm can succeed in the recovery. TG-patching is suitable for recovery because in such a situation, two affecting disconnects are separated by at least 2 logical TG-hops (as a and c). Hence there is no parent device x such that $2hop\_logical\_ngb(x)$ contains more than one affecting disconnects. Therefore parents of all disconnected devices can detect these disconnects, re-instantiate their children suitably (subject to availability of resources), and finally TG-patch the remaining devices in the 2-hop logical neighborhood to the newly re-instantiated



Events at Root Coordinator:

(A) : Re-instantiation information from **g'** (at time t_A)

(B) : Inquiry from **d'** about its non-BFS child (at time t_B)

(A) occurs before (B)	(B) occurs before (A)
informs **d'** about **g'** (more common case)	buffers the request (up to t_A) then informs **d'** about **g'**

Figure 5.12: Handling Double Disconnect with Help from Root Coordinator

devices independent of each other. Recovery can thus be achieved. □

Lemma 5.4 (Recovery from Burst Disconnects) *Recovery from a burst disconnect is not possible by TG-patching but occurs eventually.*

Proof. Figure 5.13(b) illustrates this scenario and how the recovery algorithm proceeds. Consider two devices a and c (instances of nodes A and C) in the disconnected burst. Let a be the BFS parent of c. $parent_{bfs}(a)$ will detect that a has been disconnected, and it will re-instantiate node A with another device a'. However no instantiated node other than children of c can detect the latter's disconnection. Since in our re-instantiation model, only parent devices take corrective action, a replacement for c can only be chosen by a'. More specifically, after re-instantiation, a' receives from $parent_{bfs}(a)$ the information about $2hop\_logical\_ngb(parent_{bfs}(a))$. Since, a' is not aware of the disconnection of c from the instantiated tree, it attempts and fails to TG-patch c to itself. When it receives no `HELLO_ACK` back from c, it detects that the latter has been disconnected and then it must initiate the instantiation process starting at node C. Since a' only receives the 2-hop logical neighborhood of its BFS parent, it is not aware of the instances f, g. a' would have been aware of these after receiving a `NEIGHBOR_INFORM` message from c had the latter not been disconnected.

Meanwhile, devices in the instantiated subtree rooted at f will not receive `HELLO` messages from their parents and will uninstantiate themselves eventually. In other words,

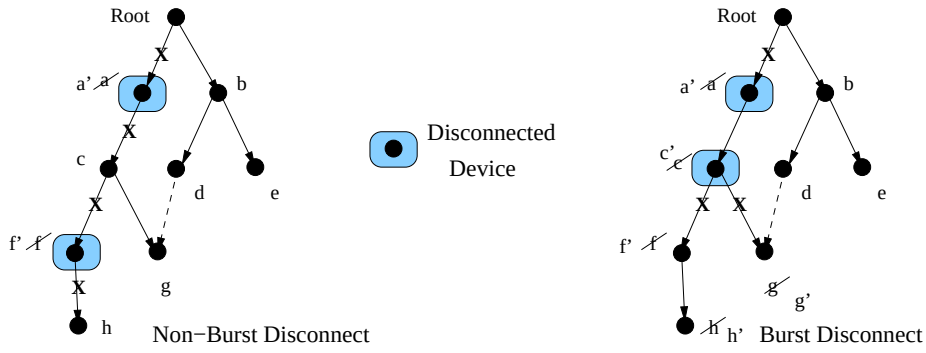


Figure 5.13: Types of Affecting Disconnects

in this type of a situation, the entire subtree downstream of the burst disconnect will be re-instantiated. □

Choosing to store only the 2-hop logical neighborhood information was a design decision from the standpoint of recovery from disconnections in most situations. However a more verbose protocol which exchanges more than 2 logical hops of instantiation information can be applied for more rapid recovery from burst disconnects. We leave this for future investigation.

Theorem 5.2 (Re-Instantiation) *In event of persistent disconnections of instantiated devices participating in a task, the re-instantiation algorithm described in this section will facilitate eventual recovery from these disconnects.*

Proof. From Lemmas 5.1, 5.2, 5.3 and 5.4, we see that the re-instantiation protocol is able to recover from both single and several types of multiple disconnects which are of a *persistent* nature. □

If the disconnects are *transient* in nature, the detection algorithm occasionally fails to detect them with `HELLO` messaging. This can be advantageous in many situations as the disconnects are *hidden* from the re-instantiation protocol. Re-instantiation causes extra overhead at the TG-layer (due to additional messaging and delays) as well as at the TaskApp layer (due to retransmission of ADUs); hence its rapid occurrence should be

avoided. But, in scenarios when a transient disconnect is detected, it can result in unnecessary re-instantiations. In fact in many cases, the re-instantiation algorithm may end up selecting the same device that was only briefly disconnected. Because of the above reasons, the value of H in the disconnection detection protocol is of significant importance.

5.3.4 Handling Disconnections During Instantiation

As we remarked earlier, there are a few differences in the techniques used to recover from disconnections during the instantiation phase and afterward, although the broad principles are the same. The main difference is the following: during the instantiation phase, the instantiated devices are not aware of their 2-hop logical neighborhoods until subtree confirmations trickle back upstream. Hence, if a device gets disconnected from its child at such a stage, although it can rediscover a new child, it may have to discard the instantiation that has already progressed downstream.

The recovery protocol is simpler in this situation because instantiation proceeds only along the edges of $BFST_{TG}$. Hence, unlike the post-instantiation recovery process described in Section 5.3.2, no instantiated device is even aware of its non-BFS parents or children. Therefore, a device tracks the reachability of its BFS parent and BFS children alone after becoming instantiated. This tracking is achieved by a periodic exchange of `PREINST_HELLO` messages which is a process similar to the periodic exchange of `HELLO` messages; the only difference being that the parent devices alone send these messages to their instantiated children who reply with a `PREINST_HELLO_ACK` message.

Before sending a `PREINST_HELLO` message, the parent device (say x) schedules a timer which is canceled at the instant a `PREINST_HELLO_ACK` reply returns from the relevant child instance (say y). If the above timer expires, then the parent device uninstantiates the child and searches for a suitable replacement. The child device, on the other hand, starts a timer as soon as it receives an `ACK` from the parent, and waits for a `PREINST_HELLO`. If the latter doesn't arrive before the timer expires, the child uninstantiates itself. Typically, the

child timer is scheduled for twice the duration of the parent's timer so as not to suffer from frequent uninstantiations.

After a replacement y' is discovered, the parent device x attempts to *patch* its grandchildren instances (if they have been instantiated) with this newly discovered device. Note that there are no 2-hop logical neighborhood exchanges that have occurred until this stage, hence x would know about its grandchildren (y's children z, w) only if y had informed x about them. This is done by means of a `TWHOHOP_CONFIRM` message: as soon as y gets a `CONFIRM` from its child z, it forwards that information upstream to x as a `TWHOHOP_CONFIRM`.

After x and y' have the usual instantiation handshake, y' obtains information about z and w from x, and it sends `PATCH_ACK` messages to z and w. These devices then respond to this by `PATCH_CONFIRM` messages which in turn contain the addresses of their child instances. This is done so that y' learns its 2-hop logical neighborhood and becomes more resilient to future disconnects, if any. Even if z or w were in `UNINSTANTIATED` state, they would respond to the `PATCH_ACK`, for otherwise, y' would have to find replacements for z and w.

5.3.5 Impact of Disconnections on the Application Layer

The application (TaskApp) layer of every participating device keeps up-to-date (in-out) data-flow tuple information for parent and child devices. If the running task gets disrupted by disconnection of some participating devices, then it is the responsibility of the BFS-parent device to transfer the application state to the newly instantiated replacement device, and then resume the application data-flow. Meanwhile data packets reaching old device instances are dropped by those devices which would be in the `UNINSTANTIATED` state after being disconnected. Our average effective throughput metric ($AvgEffT$) attempts to capture the effectiveness of our recovery algorithm by measuring the fraction of the application data that reaches the currently instantiated data sinks from the source. A higher layer buffer management scheme at the originator of the data-flow can increase the reliability of task completion. We discuss this further in Chapter 7.

Chapter 6

Performance Evaluation

In this chapter we present detailed results of the simulation of our protocols under different operative scenarios. We simulate the algorithms (and protocols) described in Section 5 using the popular public domain network simulator *ns-2* (from ISI and UC Berkeley) [3] for several different scenarios. We consider a standard mobile ad hoc network which obeys the IEEE 802.11 standard [26]. Direct Sequence Spread Spectrum (DSSS) with a single data-rate of 2 Mbps (the default configuration in ns-2) is simulated at the physical layer of each radio node. The medium access control (MAC) is configured in the Independent Basic Service Set (IBSS) mode and the Distributed Coordination Function (DCF) mode. These result in the formation of a multihop ad hoc network. All simulations are conducted for flat two dimensional rectangular surfaces. The Ground Reflection (Two-Ray) model is used to model radio propagation (path loss) [36]. The path loss exponent of this model is 4 which means that the power of the received signal falls as a fourth power of the distance between a transmitter and a receiver.

All devices are simulated to run the Dynamic Source Routing (DSR) protocol [46] which is currently a representative on-demand routing protocol for MANETs. For simulation purposes, each device is assumed to perform a certain service which it makes available to the remainder of the devices in the network. A remote device is assumed to be able to

Table 6.1: Simulation Parameters (Heterogeneous Case)

Simulation Parameter	**Value**
Number of Devices, N	$50, 100$
Simulation Area, A	1500m×300m, 1500m×600m
Transmission Radius, Tx	250m
Mobility Model	Random Waypoint ([20])
MANET Routing Protocol	Dynamic Source Routing (DSR)
#Classes of Devices, $\|C\|$	12 (classes assigned with uniform probability $\frac{1}{\|C\|}$)
Size of TG: $(\|V_T\|, \|E_T\|)$	$Tree\ (6,5); TG_1\ (6,7); TG_2\ (9,11)$ (See Figure 5.2)
Simulation Period, $t_{last}^{ADU} - t_{first}^{ADU}$	400s
MaxSpeed, v_{max}	1, 5, 10, 15, 20 m/s
MinSpeed, v_{min}	0 m/s
Pause Time, PT	0, 100, 200, 300, 400, 500, 600s
Data Traffic Pattern (at U)	Constant Bit Rate (CBR), Exponential
Mean ADU size, period	2500 bytes, 1.0s
Cross Traffic Pattern (if applicable)	10 random (source,destination) pairs exchange 512 byte packets after every 1s interval

broadcast a query for a particular service with the name/type of that service encapsulated in the query packet. The service-providing device upon receiving a query Q can respond to the sender of Q if it matches the criteria specified in Q. These responses are sent using UDP over a well known port. All other types of packets are sent using TCP.[1] Simulator code and relevant software are available via reference [63].

First, we present the results for the heterogeneous case in which all nodes in each Task Graph possess distinct labels. In other words, this scenario represents a task where each subtask needs a different type of device to be executed upon. The MANET too consists

[1]TCP Reno implementations in ns-2 have been used for simulation purposes including the simulation of the connection establishment phase.

of several instances of each type of device. Later on in Section 6.2, we present results for the homogeneous case in which all nodes in the task graph possess identical labels and so do all nodes in the MANET.

6.1 Simulation Results for the Heterogeneous Case

The simulation parameters for this case are summarized in Table 6.1. We show simulation results for task-graphs in Figure 5.2. We refer to them as $Tree$ TG, Non-Tree TG_1 and Non-Tree TG_2 respectively. The instantiation process begins at a time when steady state is reached in the network with respect to MaxSpeed and Pause Time parameters, and almost immediately, a data source at the *user/root* node starts sending data to the data sinks using the four different traffic load patterns specified in Table 6.1. The results are averaged over seven different sets of mobility scenarios for each case. Devices which are not part of the current instantiated TG do not relay data packets, and such packets are not buffered. In other words, if a device which was part of a TG becomes disconnected while there is a packet in transit, the packet is lost.

Dilation First we analyze the constant mobility scenarios for different simulation parameters. We evaluate the quality of embedding using the average dilation metric. For every mobility scenario, dilation is measured initially after completion of instantiation and subsequently after every re-instantiation event. These values are then averaged over the simulation time period to yield one value. We observe from Figure 6.1 that average dilation for the embedding scheme does not vary greatly with speed; in fact d_{avg} lies between 1.25 and 2 hops for all three task graphs at all different values of MaxSpeed. This means that the average number of physical hops between two instantiated nodes in TG is low and remains approximately constant under mobility. This is because of the approximately uniform spatial distribution of device categories and the reasonable abundance of devices of each category in the network (5 to 13 of each type).

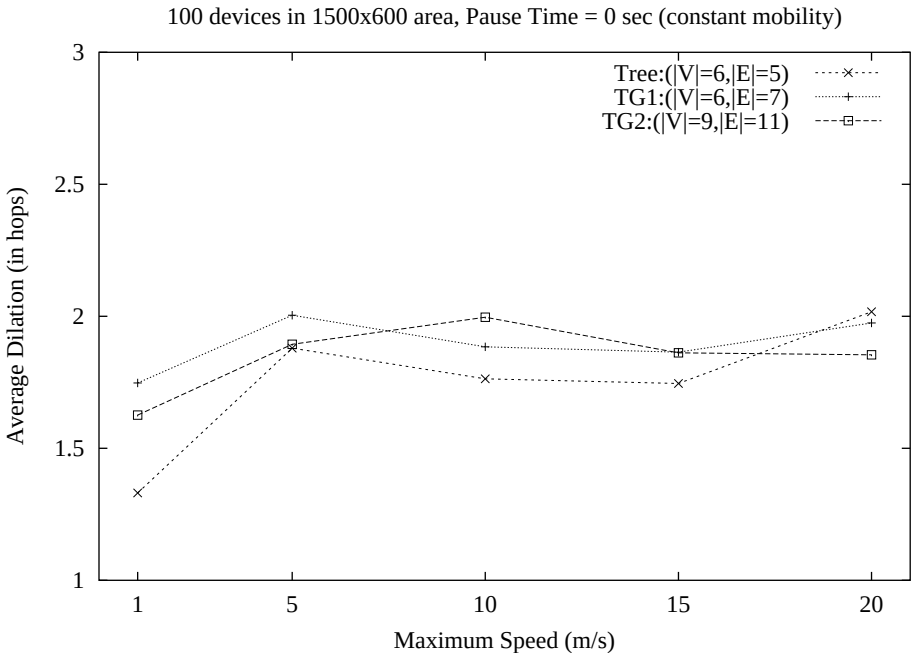


Figure 6.1: Average Dilation vs. Speed

However, we do observe that d_{avg} increases when the maximum speed is increased above 1 m/s. The principal reason for this is the following: at MaxSpeed=1 m/s, re-instantiations are rare and the d_{avg} does not deviate significantly from its value after initial instantiation. On the contrary, at greater speeds, re-instantiation events occur more frequently because of logical neighbor instances either having moved far away from each other or having been disconnected by a network partition. Either of these events disrupts the usual smooth exchange of `HELLO` message resulting in re-instantiations. Owing to the uniform distribution of device categories in space, the re-instantiation process will find another device with similar attributes within its vicinity. Although that keeps the contribution of the new path length towards d_{avg} low, the hop distances between existing instances along other TG edges are likely to have increased over time (although not high enough to cause re-instantiations along these edges). Increased hop distances cause d_{avg} to increase at higher speeds, on the whole.

Another observation from Figure 6.1 is that at lower speeds, d_{avg} is lower for $Tree$

than for TG_1. This is obvious because, our heuristic algorithm attempts to minimize the hop count only along the BFS-tree edges of a task graph both during instantiation as well as re-instantiation; since TG_1 has extra non-BFS edges, the minimization does not occur along those edges; thus the latter has a higher dilation. The above reasoning does not hold at high rates of mobility as all instantiated paths break more often and device category distribution is spatio-temporally more uniform in the neighborhood of a device. Hence, non-BFS-tree edges are likely to be mapped onto paths with similar lengths as BFS-tree edges quite often, and that causes d_{avg} values to be similar for both TG_1 and TG_2.

Instantiation Time Table 6.2 compares the times taken for instantiating (embedding) each TG on the MANET. We depict the minimum, maximum, and median times for each TG for three different maximum speeds. We show the median instantiation time instead of the average instantiation time since the time samples are skewed. Generally, instantiation times for TG_2 exceed those for TG_1 and $Tree$ since the former is a larger task graph and it needs exchange of packets between a larger number of devices during instantiation. Some samples are much greater than the rest owing to the dynamics of TCP executed over the DSR routing protocol during the instantiation process. After the `C_RESP` packet reaches a coordinator node, it sends `ACK`s encapsulated in TCP packets since they have a higher probability of getting lost if sent using an unreliable transport protocol. All subsequent communication (except `S_QRY` and `C_RESP` packets) also uses TCP. Now, if for some reason a route error occurs while a TCP transmission has not completed, TCP attempts redelivery only after waiting for a period of time even if a new route is rediscovered immediately by DSR. This period can be as large as 6 seconds (default initial retransmission timeout value in the ns-2 implementation of TCP) if no prior communication has occurred between the two communicating devices. If a route error occurs shortly after two devices have communicated using TCP but before another TCP transmission is completed, the retransmission timer is set based on the round trip time estimate between those two devices and hence it is usually less than 6 seconds. Hence, we see instantiation time samples larger than 6 seconds

Table 6.2: Instantiation Time (in seconds)

	$N = 100$ **devices**			$N = 50$ **devices**		
Task Graph	**min**	**max**	**median**	**min**	**max**	**median**
$Tree$ (1m/s)	0.795	6.561	1.435	0.278	0.538	0.433
TG_1 (1m/s)	0.810	6.819	1.399	0.213	0.416	0.319
TG_2 (1m/s)	2.170	7.957	6.674	0.623	6.832	2.170
$Tree$ (5m/s)	0.670	6.111	1.728	0.313	0.416	0.409
TG_1 (5m/s)	0.536	7.708	6.278	0.312	0.486	0.383
TG_2 (5m/s)	1.742	9.537	7.827	0.576	13.137	6.616
$Tree$ (10m/s)	0.643	1.438	1.216	0.232	6.315	0.551
TG_1 (10m/s)	0.842	6.694	1.530	0.321	6.153	0.387
TG_2 (10m/s)	3.337	9.168	7.275	0.529	7.515	0.614
$Tree$ (15m/s)	0.749	4.039	1.062	0.331	0.421	0.353
TG_1 (15m/s)	0.446	6.511	0.909	0.313	0.548	0.417
TG_2 (15m/s)	1.520	4.090	3.241	0.718	13.072	7.310
$Tree$ (20m/s)	0.651	2.062	1.088	0.308	6.224	0.517
TG_1 (20m/s)	0.717	4.022	1.484	0.309	6.320	0.409
TG_2 (20m/s)	1.361	7.674	5.262	0.472	6.247	0.637

on several occasions. If TCP is augmented with explicit feedback mechanisms (briefly mentioned in Section 5.3), then instantiation times can be reduced significantly.

Also, we observe no monotonic pattern as a result of increasing mobility of devices. This can be attributed to the uniform spatial distribution of device categories in all mobility patterns as well as large variability in TCP timers during the multiple handshaking steps of the instantiation process. For the $N = 50$ scenario with the same device density, we observe that instantiation times are in the same range as in the 100 device network. The median times are generally lower for the $N = 50$ case since in a smaller network, broadcast traffic consumes lower bandwidth and also a lesser number of devices (approximately half) are

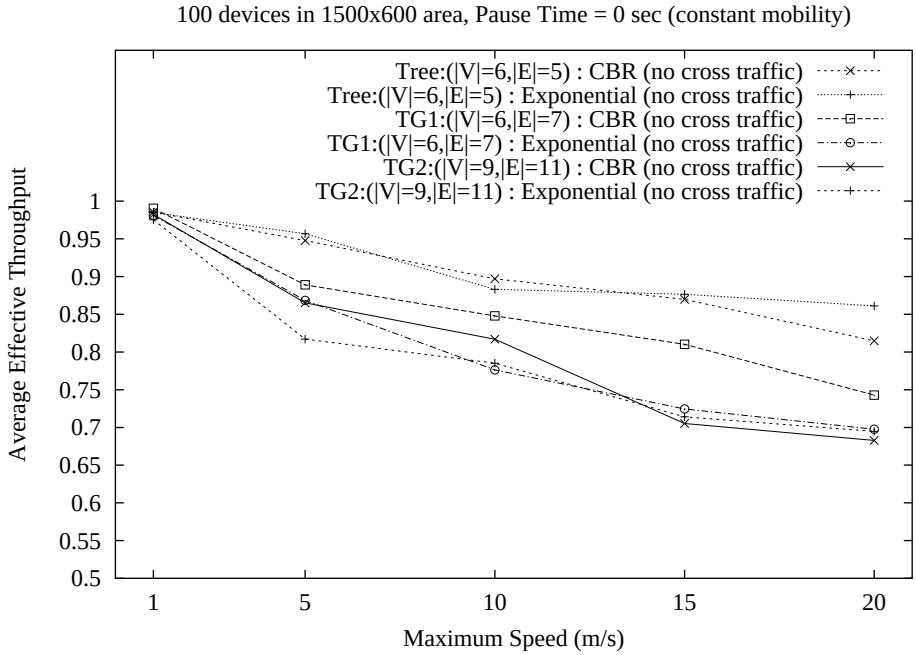


Figure 6.2: Average Effective Throughput: Variation of v_{max}, $N = 100$

likely to respond to `S_QRY` packets, thus resulting in lower queuing delays for other packets in the network.

Effective Throughput After the instantiation process is completed, we begin data transmission from the user node (source) to the various sinks shown in Figure 5.2 according to particular tuple specifications. In $Tree$ TG, instances of A, C and E get one flow each. In TG_1, the instance of E gets 4 data-flows in all, through instances of various relay nodes. In TG_2, instances of D and H get one flow each and the instance of G gets 4 flows. We normalize and then plot $AvgEffT$ for all three TGs for both the 100 and 50 device MANETs in Figures 6.2 and 6.3 respectively. We generate task data traffic using two different patterns: periodic constant bit rate (CBR) bursts and bursts with exponentially distributed sizes after exponentially distributed inter-arrival times (resulting in Poisson distributed bursts). The mean burst sizes and inter-arrival times are kept constant for both cases. A maximum aggregate throughput of 300 Kbps can be reached for the TG_2 scenario assuming simultaneous transmission at all instantiated devices in accordance with the underlying tuple

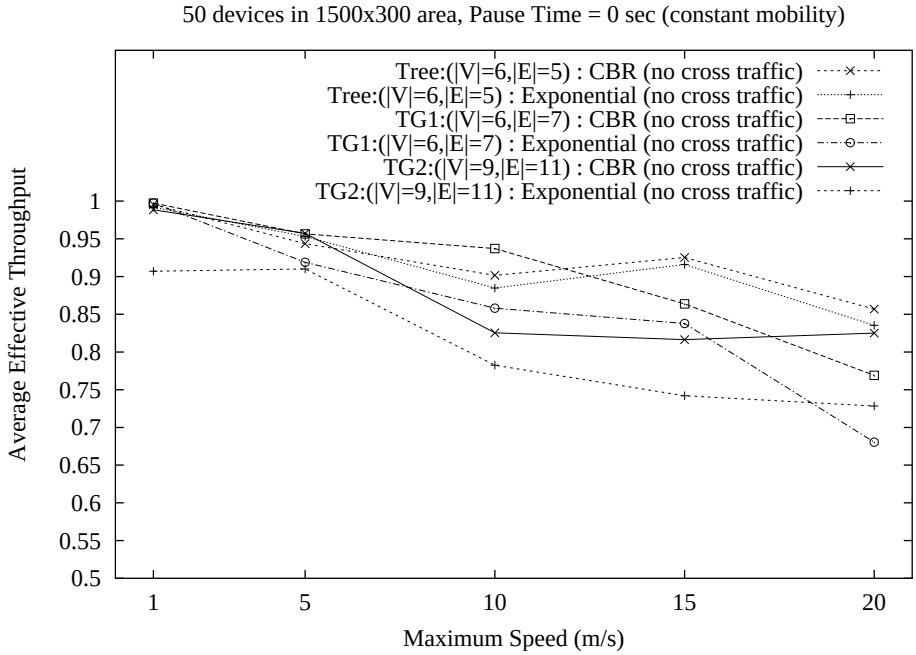


Figure 6.3: Average Effective Throughput: Variation of v_{max}, $N = 50$

architecture.

From Figures 6.2 and 6.3 we observe that at low speeds, $AvgEffT$ is almost perfect (close to 1.0). We can also observe that in general, $AvgEffT$ drops with increase in the maximum speed of devices for most situations. This is to be expected since higher speeds generally result in more re-instantiations and that results in more ADUs not reaching their intended destinations. However, $AvgEffT$ rarely drops below 70% in the simulated scenarios even under heavy mobility. This demonstrates that our protocols adapt fairly well to mobility and are able to recover from disruptions in task data flow. We make two more observations from these two figures: (1) exponential traffic pattern occasionally results in a lower throughput than the CBR traffic pattern in scenarios involving non-tree task graphs, and (2) TG_1 usually yields lower throughput than $Tree$ TG.

Exponentially distributed data generation times can occasionally result in large periods without much network activity, and this causes the on-demand routing protocols to lose routes to destinations. More route errors cause TCP to back off more often and some-

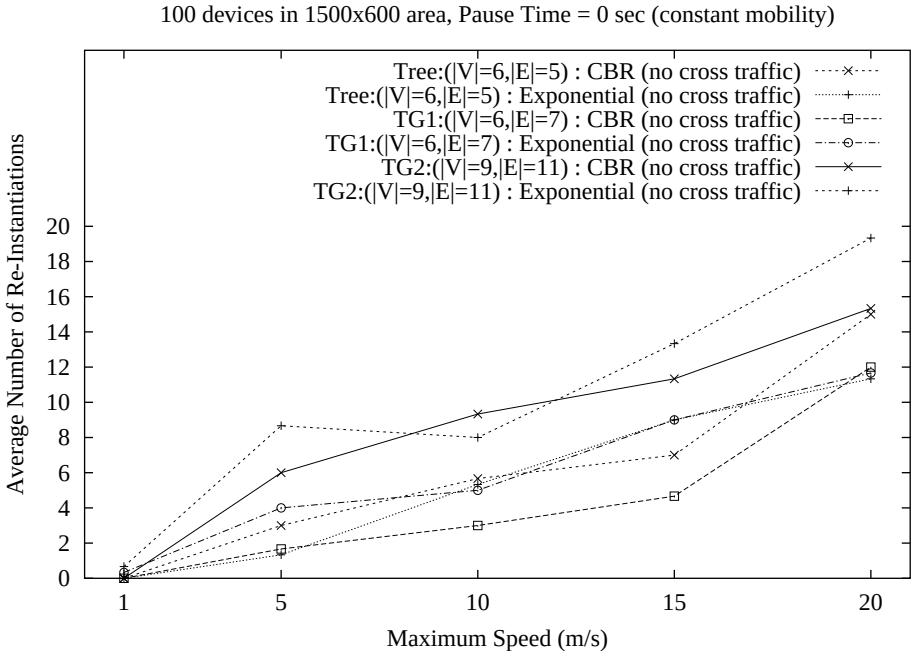


Figure 6.4: Average Number of Re-instantiations

times results in re-instantiation because of reasons explained earlier. Loss of throughput is greater in the case of non-tree TGs than $Tree$ TG because recovery from the loss of a non-BFS child takes more time than that of a BFS child as explained in Chapter 5. On the contrary, in the case of CBR traffic, periodic generation of packets keeps routes fresh.

Number and Time of Re-instantiations In Figure 6.4, we plot the average number of re-instantiations underwent during the entire simulation time (400s) for the 100 device case. Results are quite similar for the 50 device scenario and have been omitted for brevity. The rate of change in network topology increases with speed causing more network partitions and route errors. These events in turn prevent `HELLO` packets from arriving in time, and thus triggering more re-instantiations. Since packets caught in transit during the re-instantiation process are dropped (as mentioned earlier, we do not consider application layer buffering in this work), $AvgEffT$ is directly affected by re-instantiations.

Although $Tree$ TG is a sub-graph of TG_1, for the CBR data case, TG_1 suffers less

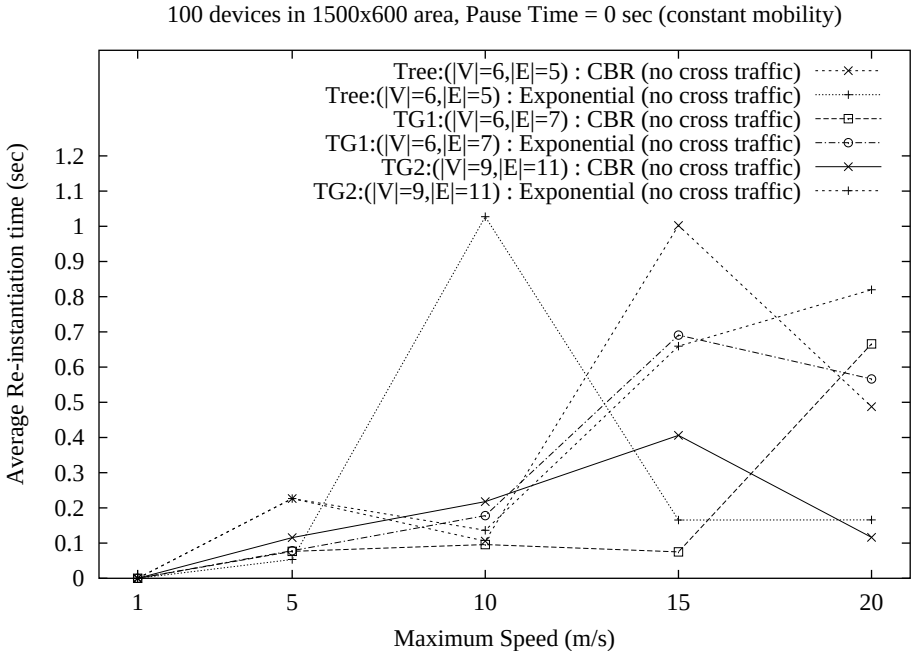


Figure 6.5: Average Re-instantiation Time

re-instantiations because data flow along the non-BFS edges of TG_1 results in the presence of more valid alternate routes (or parts of them). Hence, when a route error happens along a BFS edge (the primary cause of re-instantiations) of TG_1, often these alternate routes come to the rescue before the `HELLO` timer expires, thus reducing the rate of re-instantiations. TG-2 generally suffers more re-instantiations since it is a larger graph with more depth.

In spite of $Tree$ TG having more re-instantiations than TG_1, it experiences better $AvgEffT$ than TG_1. This is because the data-flow tuples of TG_1 (as well as TG_2) involve data flows along non-BFS edges in the graph. Also, the set of re-instantiation events is only a subset of the set of all possible disruptions. When a non-BFS parent loses a child instance momentarily due to partitions or `HELLO` timeouts, a re-instantiation will not be triggered since that is the responsibility of the BFS parent of the child instance. Hence, the throughput is affected until a new instance is found by a BFS parent and the non-BFS parent is informed of this event by a 1-logical-hop broadcast (as shown in Figure 5.9), or a route to the old instance is restored. Also, $Tree$ TG has sinks at all depths unlike

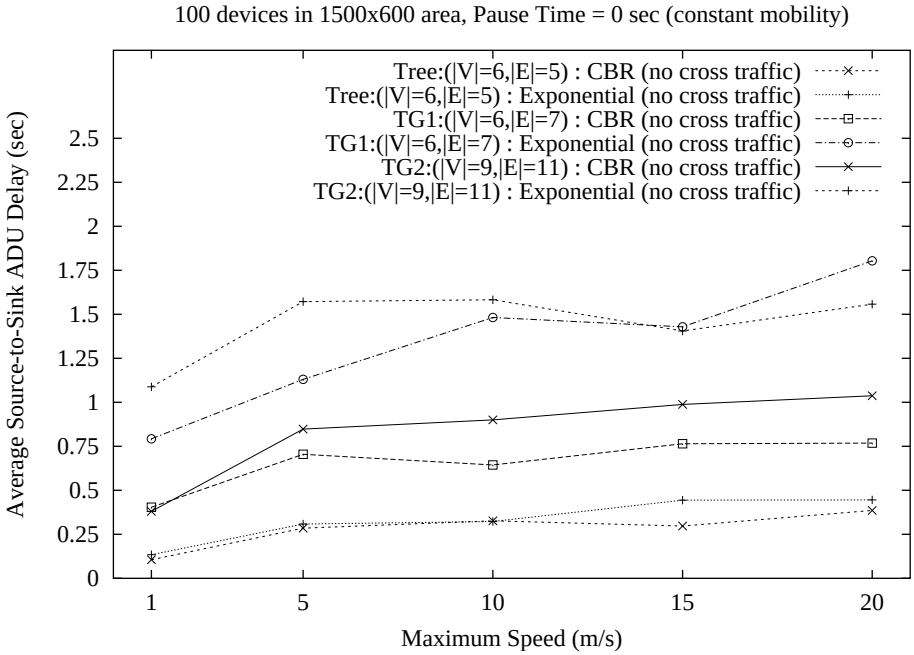


Figure 6.6: Average Source-to-Sink ADU Delay

TG_1 – hence the latter's effective throughput suffers more from a re-instantiation of an intermediate relay node. Exponential traffic generally affects re-instantiations more than CBR traffic especially for the non-tree graphs as explained before. The result of that is slightly lower throughput in the respective cases.

Figure 6.5 shows the variation of times taken to re-instantiate a TG node, i.e., the times taken to discover a new replacement for a disconnected device that can participate in the task. This time is measured from the time when the rediscovery broadcast is sent out until the time instant when a confirmation is received from the new candidate (this involves 2 round-trip handshaking steps including the broadcast). Our re-instantiation protocol is able to find a new device nearby within 1 second. In fact, in most cases, these times are only a few hundred milliseconds. Local network effects are dominant factors in the determination of this metric at higher speeds, hence there is little correlation between the values in such cases.

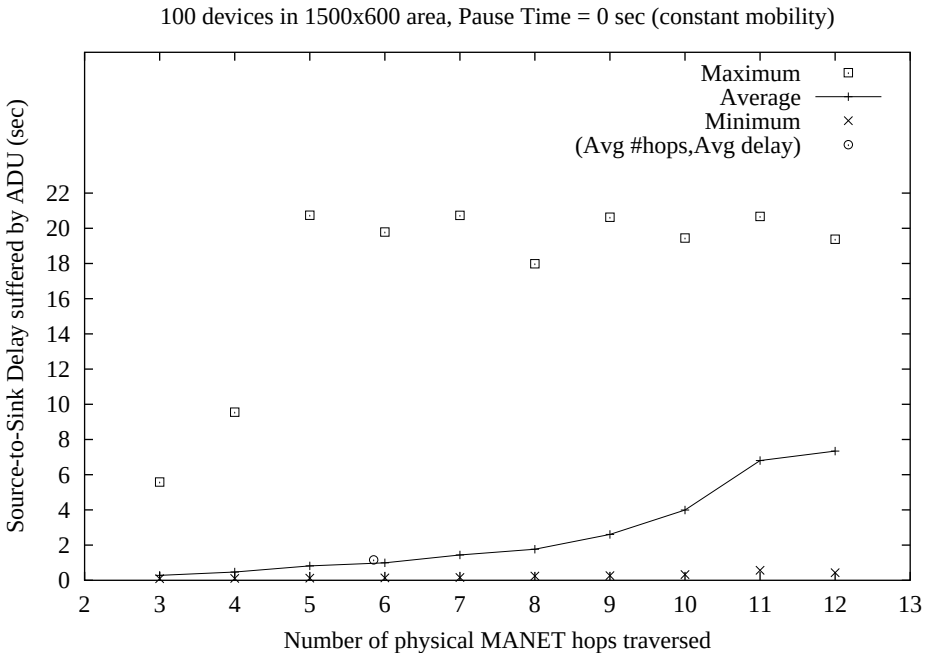


Figure 6.7: Source-to-Sink ADU Delay vs. Hops (for TG_2; $v_{max} = 10$ m/s)

Average Source to Sink Delay We now analyze the effect of varying mobility (speed) on the source to sink delays suffered by application data units. The average delay values for all TGs are plotted in Figure 6.6. We observe very low delays for $Tree$ TG (less than 0.5s) at all speeds for both CBR and Exponential traffic patterns although delays increase slightly with MaxSpeed. However, we observe greater delays for the non-tree TGs; TG_2 has higher delays because it is a larger graph and has sinks at greater depths in the task graph than the other two graphs. TG_1 has greater delay than $Tree$ TG because all its sinks are at the greatest possible depth in the graph unlike the latter.

We also observe that exponential traffic suffers greater delay than CBR traffic in almost all cases. This is due to several reasons: first, although the mean burst size of exponential traffic is the same as that of CBR, larger ADUs suffer greater delays in a mobile network since they are subjected to route changes and hence TCP delays for a longer period of time. These delays are not compensated adequately by the low delays experienced by small ADUs. Secondly, occasionally the time between ADU transmissions is much larger

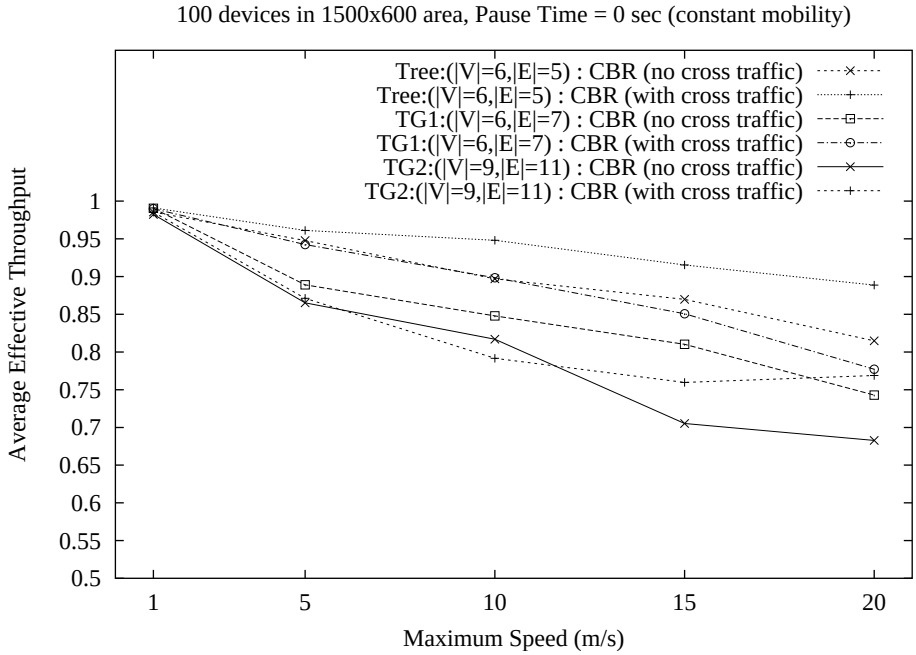


Figure 6.8: Effect of Cross Traffic: Average Effective Throughput

than the mean (1s), and that results in a greater probability of route changes between successive ADU transmissions. Also, if two large bursts are generated in quick succession while re-instantiation is happening somewhere in the network or DSR is discovering new routes, the probability of experiencing greater delays increases.

Only average delay values are shown in Figure 6.6. To illustrate how delay varies for a particular scenario as a function of MANET hops, we plot a subset of the delay values for a representative scenario (TG_2 ; MaxSpeed = 10m/s ; Pause Time = 0s ; CBR traffic) in Figure 6.7. We see that the delay expectedly increases with the number of MANET hops traversed by the ADU. Although the maximum delays are very high (above 20s), the average delays are reasonable. On the same graph we see that an ADU on average traverses less than 6 MANET hops and experiences a delay of approximately 1 second. Average delay increases linearly up to 9 hops (at a rate of about 250ms per hop) but then increases non-linearly for a greater number of hops. We examine additional interesting properties of the ADU delay distribution later in this section.

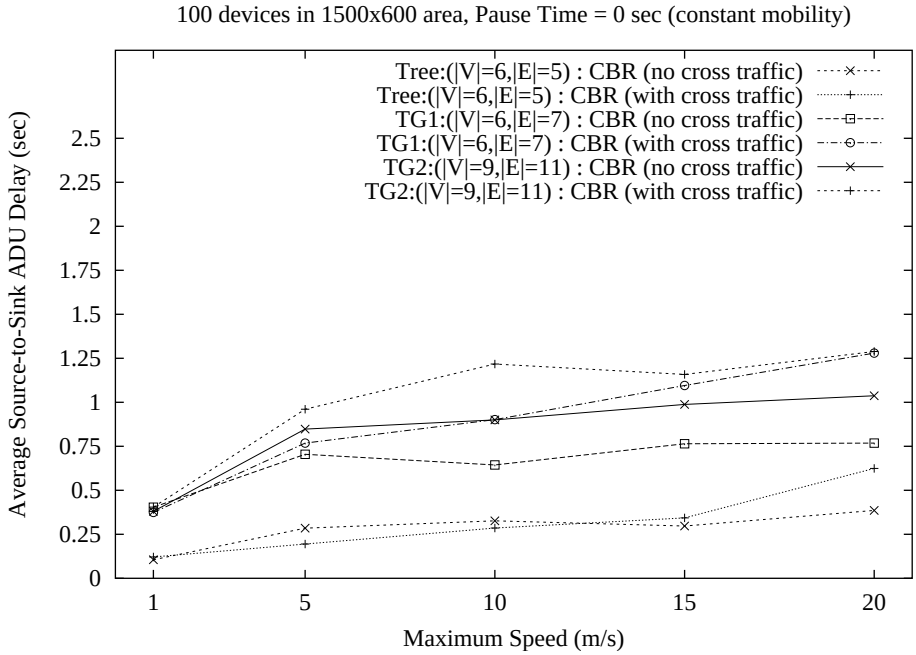


Figure 6.9: Effect of Cross Traffic: Average Source-to-sink Delay

Since the spatial device density is the same for both $N = 50$ and $N = 100$ device scenarios, all metrics except instantiation time exhibit similar behavior in both cases.

Effect of Cross Traffic In our analysis so far, we have assumed that there is no additional traffic in the MANET except task data. In this section we examine the effect of other traffic in the network on the performance of our protocols. We choose 10 source-destination device pairs at random from the 100 devices, and each source sends a 512 byte UDP data packet to the destination every second. We report the effect of such cross traffic on our throughput and delay metrics in Figures 6.8 and 6.9 respectively. We observe from Figure 6.8 that introduction of cross traffic actually improves the effective throughput in the simulated scenarios. This is not surprising because the use of an on-demand underlying routing protocol like DSR causes routes along all paths which are not currently in use to become stale. This results in lower throughput at higher speeds when route errors and partitions cause re-instantiations. Introduction of cross traffic keeps alternative routes and route caches fresh at various nodes in the network and this results in quick recovery from re-

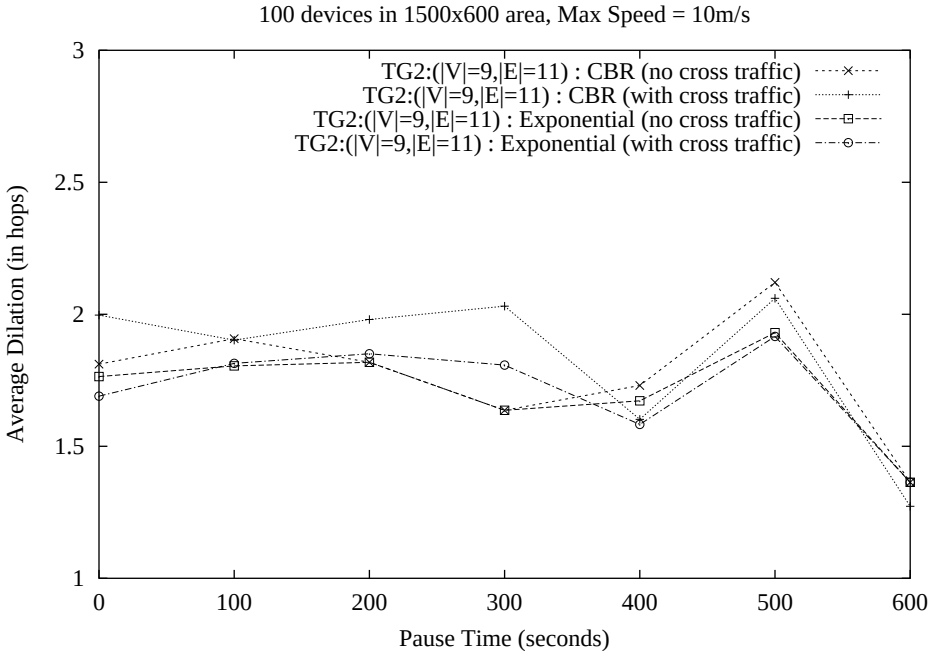


Figure 6.10: Effect of Varying PT on Dilation at $v_{max} = 10$m/s for TG_2

instantiations or even lower number of re-instantiations (graph not shown) due to `HELLO` message timeouts. We see as much as 15% increase in effective throughput for the TG_2 case.

However, one pays a price due to the flow of cross traffic in the network. Figure 6.9 shows the average source-to-sink delays suffered by an ADU for the simulated scenarios. We observe that introduction of cross traffic has resulted in slightly greater delays especially for the non-tree task graphs and for $Tree$ TG at high speeds. This is because as mobility increases, more route errors occur and a significant number of control packets are injected into the network by DSR. This results in task packets getting delayed in interface queues since control data packets related to routing get preference over regular application data packets. The most significant increases in delay are seen for TG_1 – from 750ms to above 1.25s at 20 m/s speed, primarily because all its data sinks lie far away from the data source.

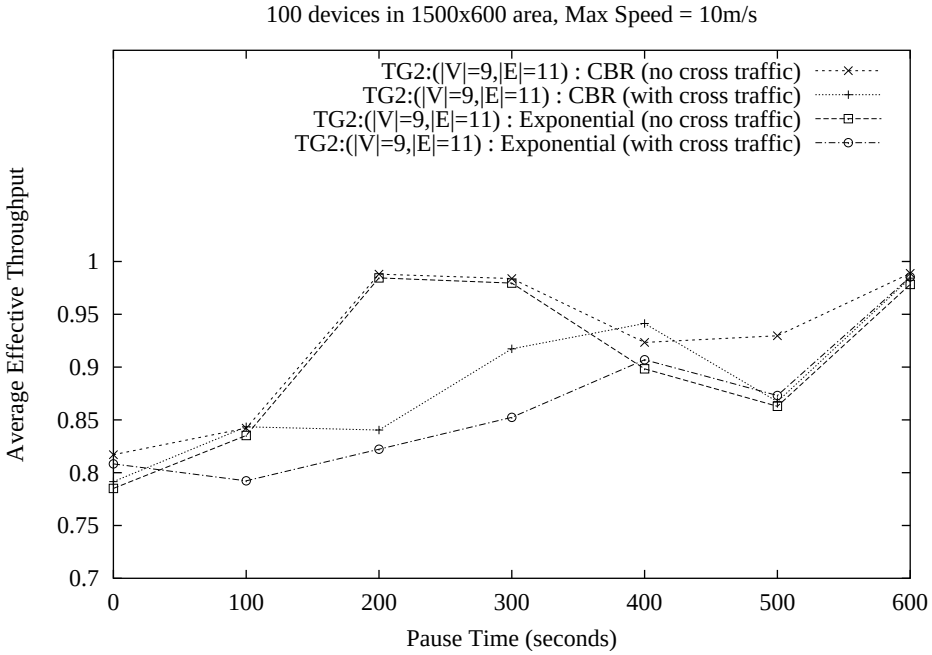


Figure 6.11: Effect of Varying PT on $AvgEffT$ at $v_{max} = 10$m/s for TG_2

Effect of Variation in Pause Times Now we investigate the effect of variation in pause time while keeping MaxSpeed constant at 10 m/s. The pause time parameter PT was varied from 0s, i.e., constant mobility (already considered earlier) to 600s (completely static) at steps of 100s. Note that the scenario generator used pauses *all* nodes for PT seconds before they start moving. Hence, for accuracy in simulations, we began the experiments and data collection at time $t > k \times PT$ where $k > 1$ and is sufficiently large such that a steady state that is characteristic of the movement pattern (in terms of fraction of nodes moving) was reached. In order to realize this we created mobility scenarios much greater in duration than 600s for $PT = 200, 300, 400, 500$. $PT = 0$ and $PT = 600$ are special cases and were handled by scenarios of duration 600s. Task execution protocols were run for 400 seconds from $t = 1100s$ to $t = 1500$s for the former cases and from $t = 200$s to $t = 600$s for $PT = 0, 100, 600$s.

The results are shown in Figures 6.10, 6.11, and 6.12 for TG_2 for all possible traffic patterns. We observe from Figure 6.10 that average dilation is minimum ($d_{avg} \approx 1.4$) for

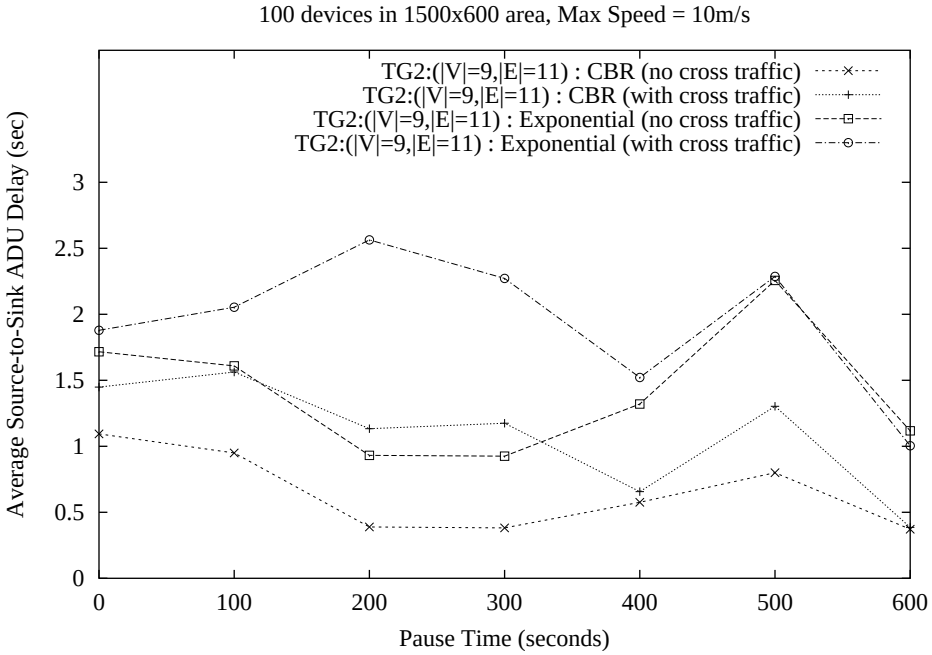


Figure 6.12: Effect of Varying PT on ADU Delay at $v_{max} = 10$m/s for TG_2

the static scenario whereas it is more or less unaffected by PT (lies in the 1.6–2.1 range). This is expected as the spatial uniformity of devices is the dominant factor in determining this metric at all pause times.

From Figure 6.11 we observe that our protocols manage to yield almost perfect effective throughput in the static scenario. This is because once the devices are instantiated, no re-instantiations occur due to mobility, and hence the sink node instances hardly lose any ADUs. Moreover, $AvgEffT$ does not drop below 0.8 even for the constant mobility scenario. This value hovers in the 0.8–1.0 range for all the intermediate values of PT. Monotonically increasing patterns that are observed in the constant mobility and varying MaxSpeed case (Figures 6.3 and 6.2) are not observed here because the rate of route changes between any pair of instantiated devices may not decrease monotonically with decrease in PT if most devices in that part of the network are static.[2] However, we believe that if a much larger number of scenario and random device pattern samples are taken, we are likely

[2]This could explain the extremely high throughput for $PT = 200$ and $PT = 300$; very similar average delay values in Figure 6.12 point to the same conclusion.

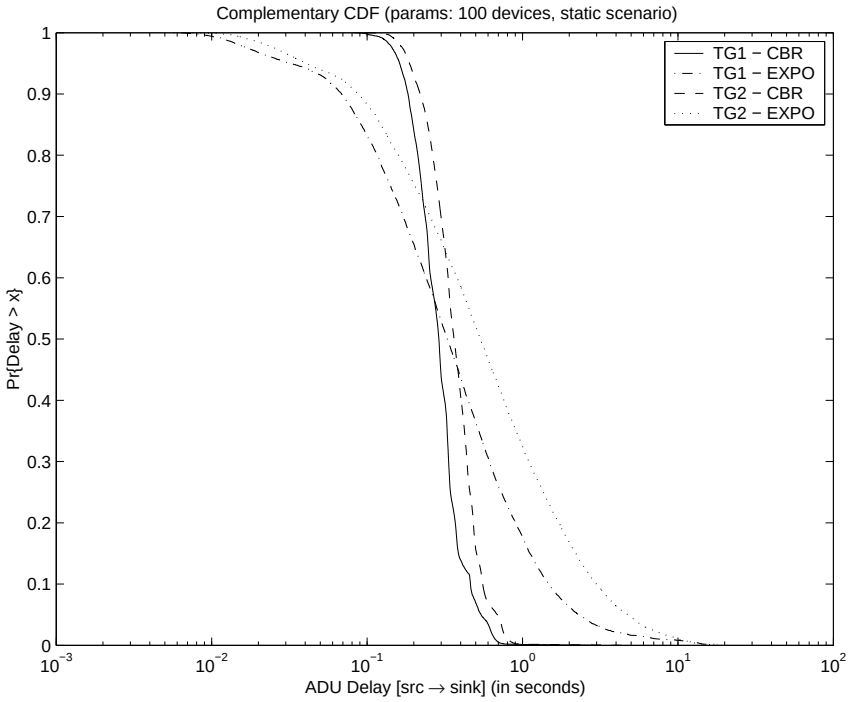


Figure 6.13: Empirical Cumulative Distribution Function of ADU Delay ($PT = 600$s)

to observe a monotonic behavior.

We also observe that exponential traffic results in lower throughput and higher average delays (both with and without extra cross traffic) than CBR traffic. Reasoning for this is similar to the one mentioned in an earlier discussion (on effect of variation of MaxSpeed), and we do not repeat that here. However, we do observe that cross traffic is not able to improve the effective throughput in these scenarios. In fact cross traffic begins to help only at greater MaxSpeed (15 and 20 m/s) for the TG_2 case (see Figure 6.8), i.e., when route errors increase significantly.

Cumulative ADU Delay Distributions So far we have investigated only the average delays experienced by ADUs. We now examine the nature of the delay distributions that occur as a result of sending task data using CBR and Exponential traffic patterns. Figures 6.13 and 6.14 show the empirical cumulative probability distributions (CDF) of ADU delay

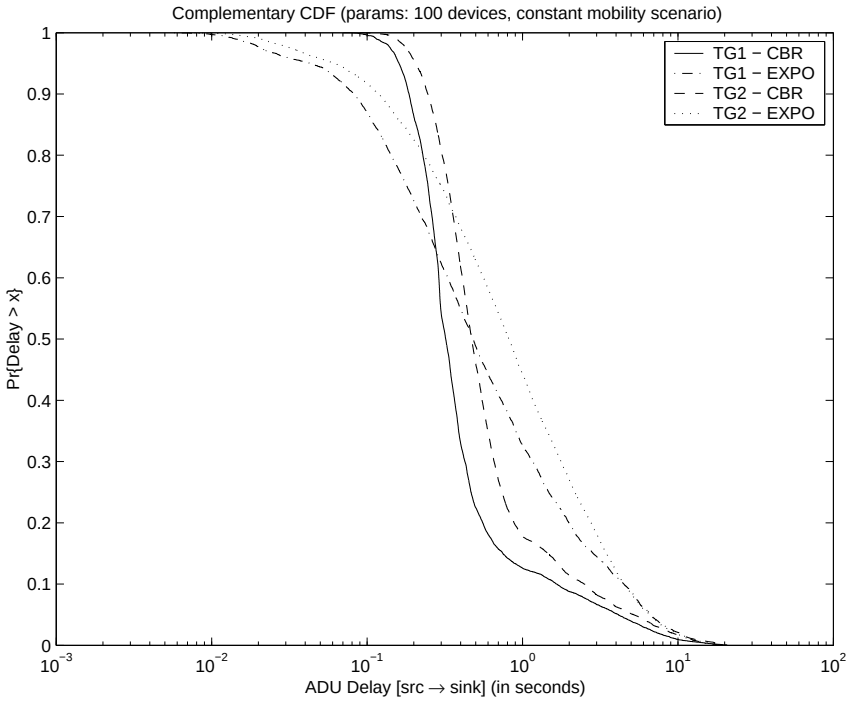


Figure 6.14: Empirical Cumulative Distribution Function of ADU Delay ($PT = 0$s)

samples. A logarithmic scale is used for the delay samples in order to differentiate between delays at lower and higher ends more effectively. In Figure 6.13, delays for the static case are plotted. We observe that CBR delay values span a much smaller range than their exponential counterparts. The structure of the task graph does not affect that of the CDF curves. That is primarily because the distribution of sinks in both TG_1 and TG_2 have a common aspect which is a dominant factor in the determination of ADU delays; two sinks each in TG_1 are 3 and 4 logical hops away from the source, respectively. Similarly, in TG_2, four sinks are 3 logical hops away and two sinks are 4 logical hops away from the data source.

CDF curves of delays in the constant mobility scenario are plotted in Figure 6.14. We can easily see that although the shapes of the curves are similar at lower values of delay, they become much flatter and somewhat heavy tailed at larger values for both TGs and traffic patterns. These samples correspond to ADUs which had to experience delays due to

route errors and expiry of TCP timers. In this dissertation, we do not attempt to investigate the exact statistical nature of the distribution, and leave that as a topic of future research.

6.2 Simulation Results for the Homogeneous Case

In this section, we investigate the scenario in which the nodes in the task graph are homogeneous, i.e., they do not have distinct attributes. First, we analyze the performance of the distributed version of the GreedyEmbed algorithm for embedding trees of different sizes and depths on static random topologies with respect to several metrics proposed in Section 3.4. Secondly, we present the results of performance analysis of the GreedyEmbed algorithm on static random topologies with varying spatial node density and investigate factors which limit the possibility of instantiation. Finally, we investigate and analyze the behavior of the distributed protocols proposed in Chapter 5 under varying physical parameters such as mobility and spatial node density (as we have done in Section 6.1).

Definition 6.1 (k-ary tree) A k-ary tree is one which is rooted and each of its non-root nodes has exactly one parent and between 0 and k children. A complete k-ary tree with ℓ levels has $N = 1 + k + k^2 + \ldots + k^{\ell-1} = \frac{k^\ell - 1}{k-1}$ nodes. If the number of nodes is less than N, the nodes are added to the tree from left to right. In other words, nodes are not added to a parent if there exists another parent with non-zero but less than k children. A k-ary tree task graph considered in this analysis consists of nodes with identical attributes.

6.2.1 Effect of Variation in Size and Depth of Task Graphs in Static Scenarios

We conducted a set of simulations for static ad hoc networks with $N = 100$ devices distributed uniformly (randomly) in a $A = 1000 \times 1000 m^2$ area. We simulated the instantiation of task graphs of several shapes and sizes on the above topology. Specifically, we chose k-ary trees as task graphs with $k = 2, 3, 4,$ and 5.

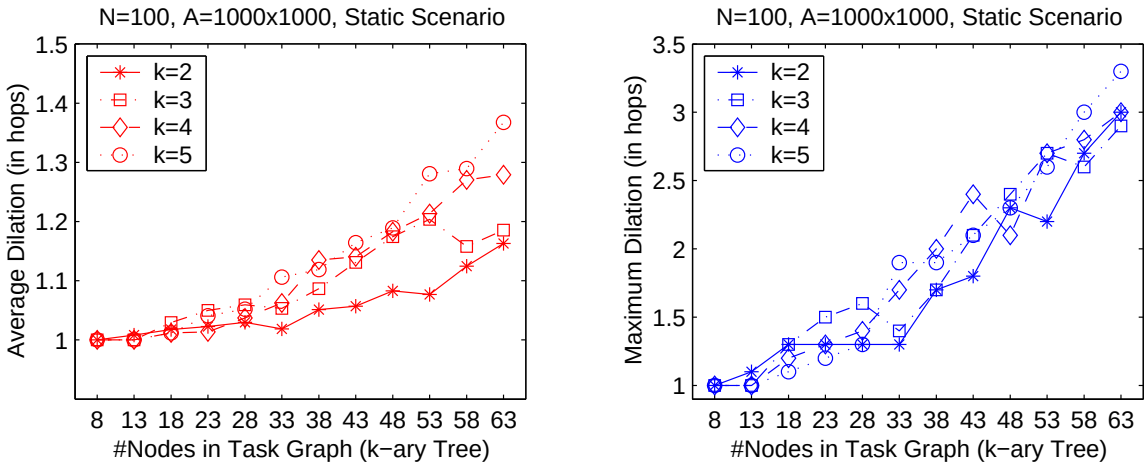


Figure 6.15: Dilation after Instantiation for *k-ary* Tree TGs (static scenario)

k-ary trees were generated for sizes up to $|V_T| = 63$ nodes for $k \in \{2, 3, 4, 5\}$. The metrics studied in this analysis were dilation (both average and maximum), instantiation time, and control overhead at the TG-layer (broadcast-, unicast-, and byte-overhead). Since no mobility was simulated in this analysis, there were no disconnections and hence no re-instantiations. For each set of parameters, 10 simulation runs were conducted with random seeds.

Dilation Figure 6.15 shows the values (averaged over 10 simulation runs) of both average and maximum dilation. We observe that the average dilation is close to 1 (optimal) for smaller trees for all values of k. However, for larger trees, the increase in dilation is greater for $k \in \{3, 4, 5\}$ than for $k = 2$, although the dilation itself is quite reasonable (< 1.4) The maximum dilation on the other hand increases at a greater rate as the tree size increases for all values of k. This phenomenon can be explained from the greedy nature of the instantiation protocol: suppose a parent wants to instantiate its k children. It will attempt to select all of them from among the devices in the nearest vicinity (single hop). For larger sized TGs, each of these k selected child nodes will attempt to instantiate k more children. Now, for higher values of k, the probability of overlap between the k sets of candidates (of intermediate nodes) is higher than for the lower values of k. Hence, there is more

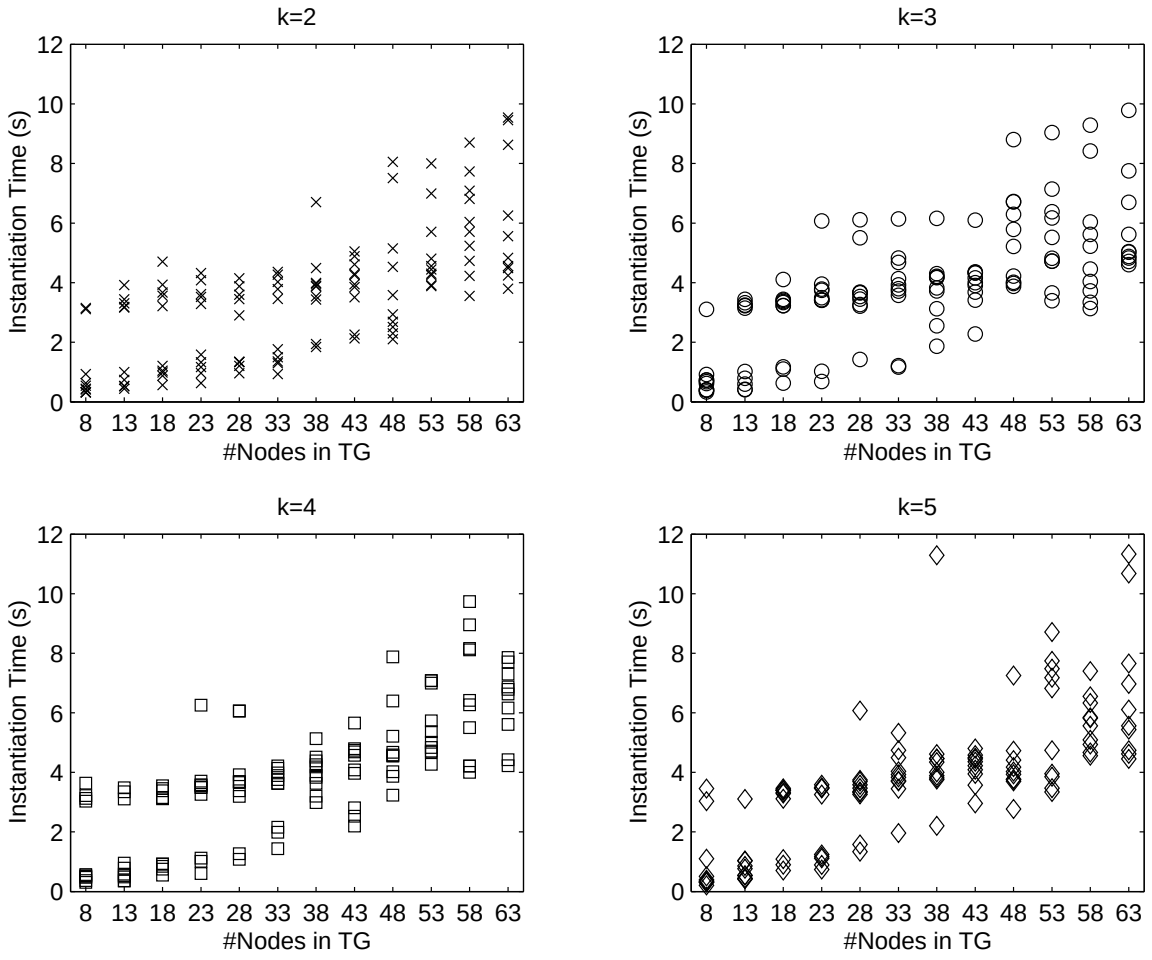


Figure 6.16: Instantiation Time for *k-ary* tree TGs (static scenario)

competition for resources in the MANET; for high k, a device is often not be able to find all its child instances within 1 MANET hop and has to increase its TTL scope instead and rebroadcast an `S_QRY`. Therefore, the average dilation value increases. For lower k, the above phenomenon does not occur and thus the average dilation increases only slightly as N increases.

We can observe that the maximum dilation is less than 3.5 hops on average. This means a TG-edge is mapped onto a path in the network that is less than 4 hops in length (although if averaged over all edges this value goes up to 1.4 hops only). The maximum dilation value shows an increasing trend but does not exhibit significantly different behaviors

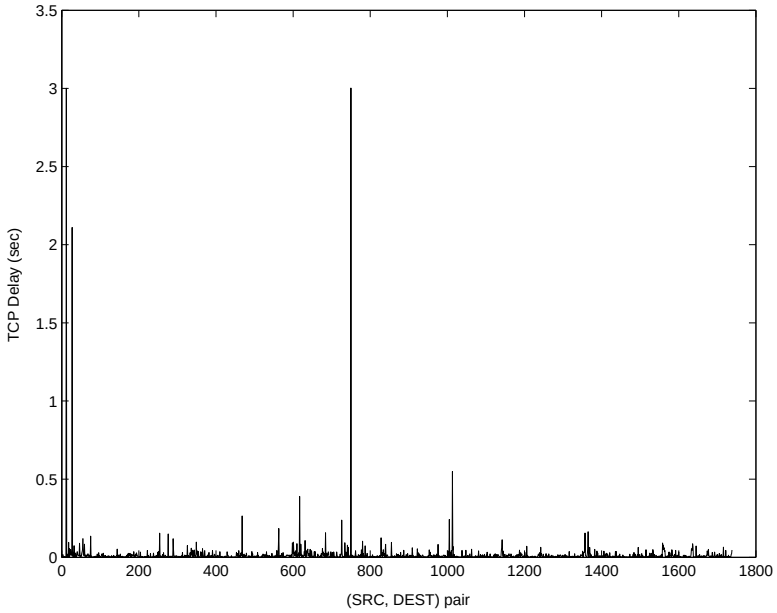


Figure 6.17: TCP Delays during Instantiation of a 63 node Binary Tree TG

for different values of k.

Instantiation Time In Figure 6.16, we plot the time taken to instantiate a *k-ary* tree for each of the 10 simulation runs. We observe a clear bimodal behavior for lower sizes of TG for all values of k: for one bunch of simulation runs, the instantiation times are closely distributed below 1 second, whereas for other runs, the times are above the 3 second mark. This is because of the TCP timeouts that happen occasionally stalling the instantiation process.[3]

The principal reason behind such 3 second TCP timeouts is the loss of SYN packets during the establishment of connection. Figure 6.17 shows the delays suffered by TCP packets for a sample simulation run for a 63 node binary tree. These losses happen in a

[3]For the simulation results reported Section 6.1, we used a default TCP timer value of 6s which was the default ns-2 value. However, for the simulations in this section we used a default value of 3s which is more in agreement with the recommendations in RFC 2988.

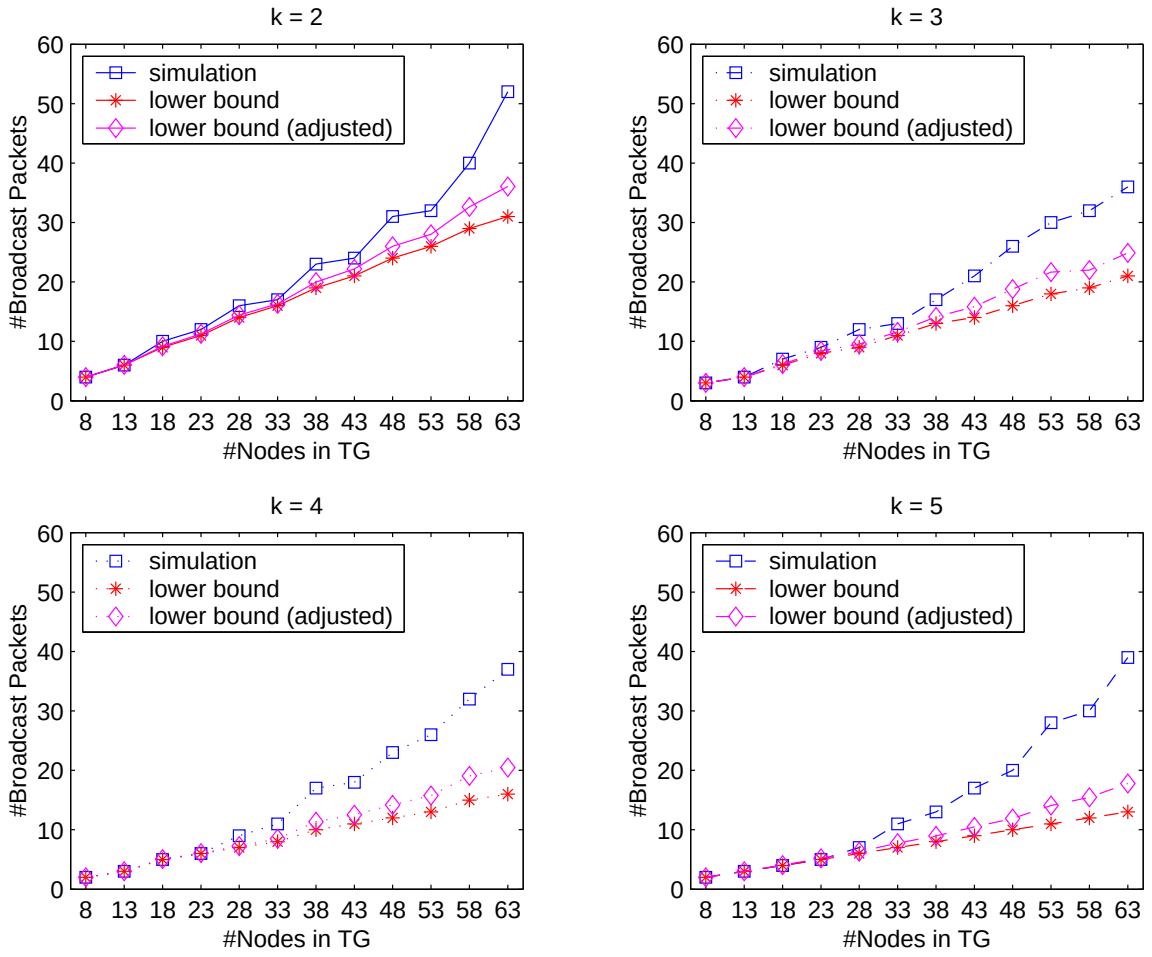


Figure 6.18: Broadcast Overhead for Instantiation of *k-ary* Tree TGs (static)

static ad hoc network due to collisions which are in turn caused by channel contention. This has been pointed out by other researchers as well [33]. However, for larger sizes of k-ary trees, the bimodal behavior is not observed any more as there are a larger number of TCP connections open. This and the presence of more instantiation and routing traffic in the network increases the probability of TCP timeouts. All of these symptoms point at the need for a better design of a reliable transport protocol for MANETs. We do not investigate that in this research further.

Protocol Control Overhead Here we analyze the control traffic overhead at the TG-layer due to our protocol. The overhead comprises of that due to both broadcast and unicast traffic. First we count the total number of broadcast packets that needed to be sent by the root and by other instantiated devices during the instantiation phase. Figure 6.18 shows the plots for each $k \in \{2, 3, 4, 5\}$. The lower bound on the number of necessary broadcasts is equal to the number of non-leaf nodes in TG since each non-leaf node must perform a broadcast for its child nodes. The minimum number of broadcasts that are needed to be originated is given in terms of the embedding function $(\varphi, \psi) : T \to G$ by this expression:

$$min_{BCAST} = \sum_{v \in V_T} \max_{\substack{w \in V_T: \\ (v,w) \in E_T}} \{\|\psi(v, w)\|_G\} \tag{6.1}$$

For a k-ary tree with $n = |V_T|$ nodes, the number of non-leaf nodes is given by $\lfloor \frac{n+k-2}{k} \rfloor$ (this is the number of terms in the summation in Equation 6.1), and hence a lower bound for the minimum number of initial broadcasts necessary for the distributed instantiation protocol is given by:

$$min_{BCAST} \geqslant \left\lfloor \frac{n + k - 2}{k} \right\rfloor \tag{6.2}$$

This is a bound linear in n that can be met only when an embedding with perfect dilation ($= 1$) can be found in the network. In this situation, no rebroadcasts are necessary. From Figure 6.18 we observe that the results agree with the dilation curves in Figure 6.15. For low k, since the dilation is close to 1, the number of broadcasts is only slightly above the lower bound and that too for larger TGs. For higher k, the two curves diverge at smaller n, since TTL timers expire more often and broadcasts with greater TTL scope are necessary for discovery and instantiation of devices. Figure 6.19 which shows curves for forwarded broadcast packets corroborates this analysis. If the average dilation of an embedding is d_{avg}, the number of broadcasts that were needed to be originated can be approximated by:

$$n_{BCAST} \approx \left\lfloor \frac{n + k - 2}{k} \right\rfloor \times d_{avg}. \tag{6.3}$$

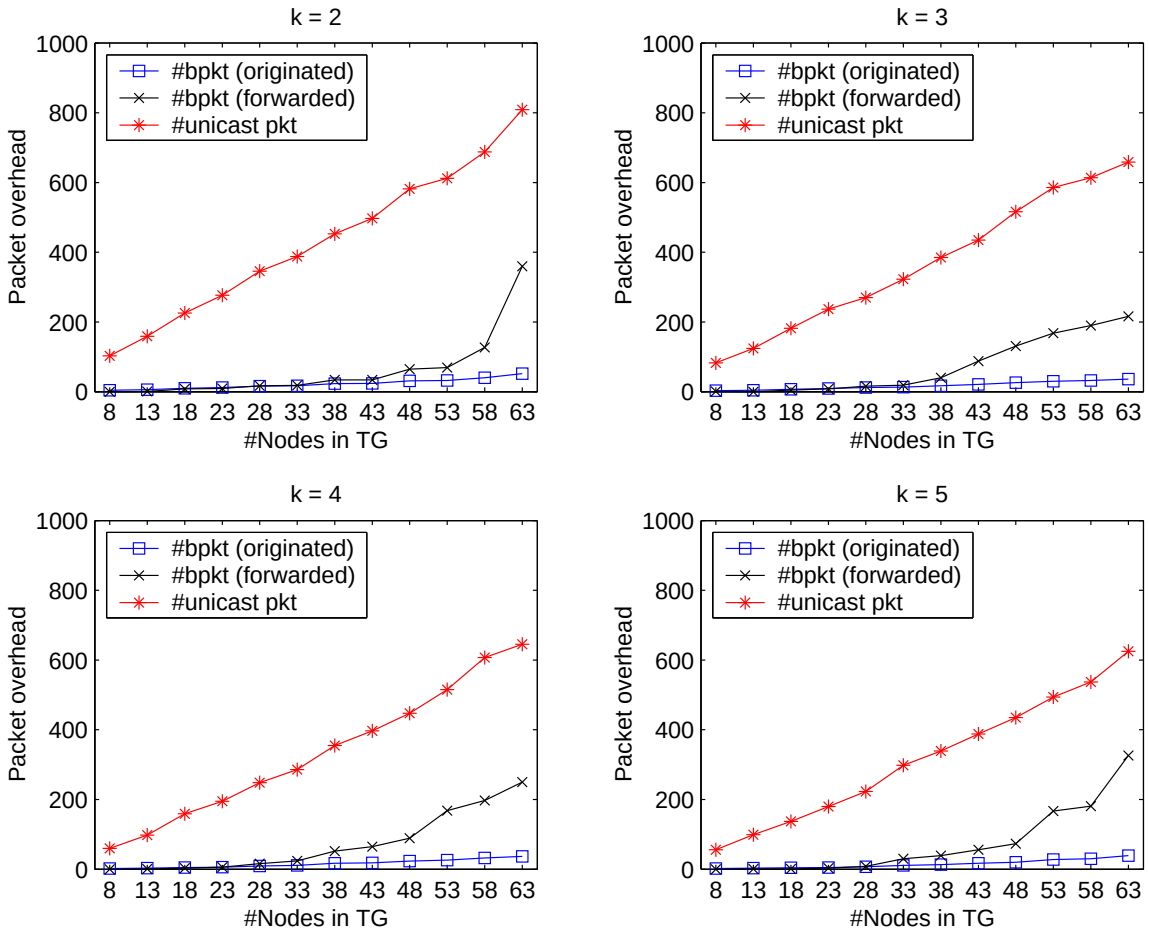


Figure 6.19: Control Traffic Overhead at TG-layer for *k-ary* Tree (static)

We plot this quantity for each value of k in Figure 6.18 and observe that it lies between the lower bound and the actual simulation result. The reason why this does not exactly coincide with the simulation results is that d_{avg} alone is not enough to accurately account for the number of broadcasts; the entire sequence of lengths of network paths onto which the TG edges were mapped determines the exact number of broadcasts as shown in Equation 6.1.

The control overhead due to unicast packets (both UDP *C_RESP* packets and TCP packets such as `ACK, CONFIRM, SUBTREE_CONF, PREINST_HELLO`) are plotted alongside the broadcast overhead curves in Figure 6.19 and on separate axes in Figure 6.20. We observe

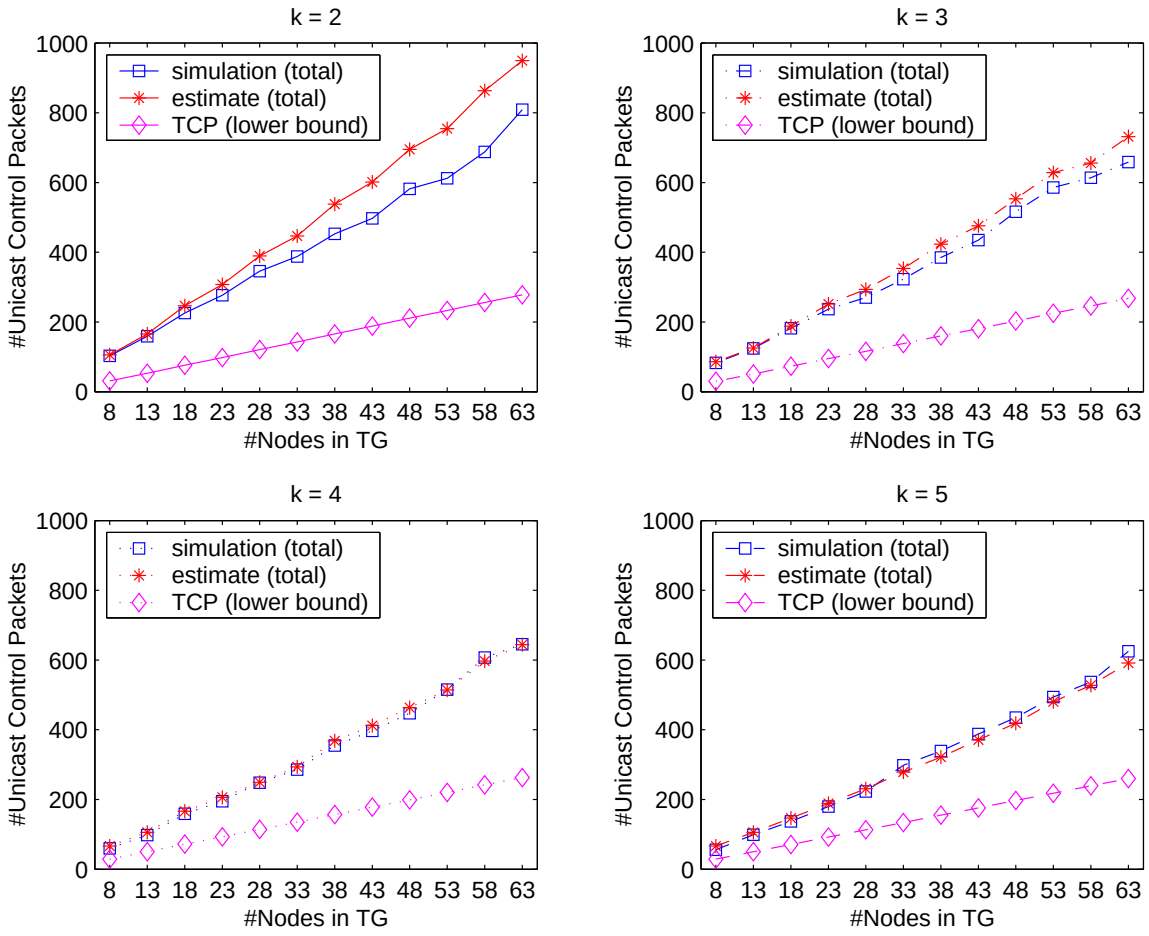


Figure 6.20: Unicast Control Overhead for Instantiation of *k-ary* Tree TGs (static)

that the control overhead is almost linear in n (size of TG) for all values of k. For the sake of comparison, we plotted the lower bound for the TCP control overhead and a rough estimate of the total unicast control overhead. The lower bound for the TCP packet overhead can be calculated as a function of n and k as follows:

$$\begin{aligned} n_{\texttt{TCP}} &= n_{\texttt{ACK}} + n_{\texttt{CONFIRM}} + n_{\texttt{SUBTREE\_CONF}} + n_{\texttt{PREINST\_HELLO}} + n_{\texttt{PREINST\_HELLO\_ACK}} \\ &\geqslant (n-1) + (n-1) + (\left\lfloor \frac{n+k-2}{k} \right\rfloor - 1) + (n-1) + (n-1) \\ &= 4n + \left\lfloor \frac{n+k-2}{k} \right\rfloor - 5 \end{aligned} \tag{6.4}$$

Note that `REJECT` packets do not feature in the lower bound calculation as in the best

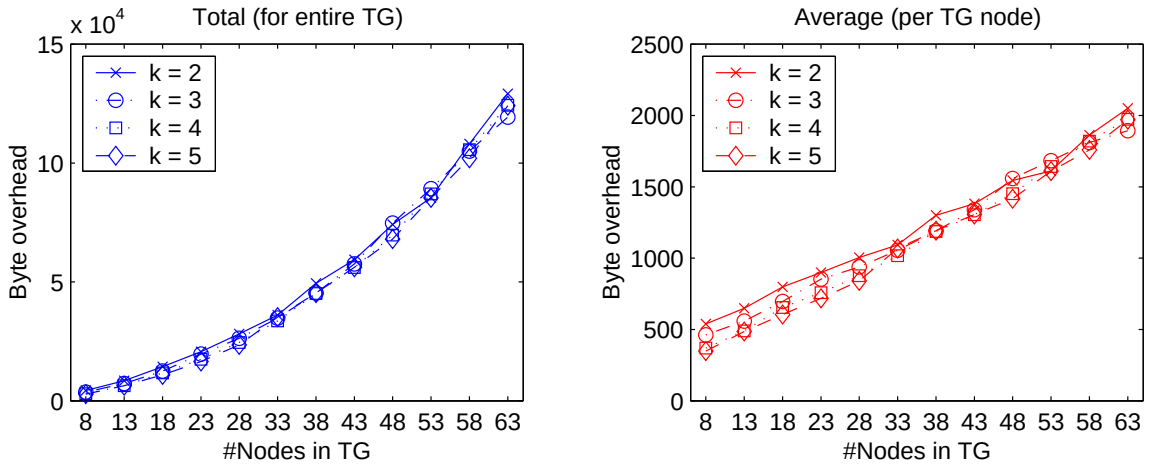


Figure 6.21: Byte Overhead for Instantiation of *k-ary* Tree TGs (static)

case no `REJECT` packets would have to be sent. The rest of the unicast control overhead is contributed by UDP `C_RESP` packets, the number of which can be estimated in the following manner: if there are N nodes uniformly distributed in area A, then the average number of neighbors (node degree) is given by $\rho = \frac{N}{A}\pi R^2 - 1$, where R is the transmission range of a device. Since all the devices in the network are assumed to be homogeneous in this analysis each broadcast query will have ρ responding candidates on average. Since a k-ary tree on n nodes results in $\lfloor \frac{n+k-2}{k} \rfloor$ originated broadcasts in the best case, the number of `C_RESP` packets can be estimated as:

$$n_{\texttt{C\_RESP}} \approx \rho \times n_{BCAST} \approx (\frac{N}{A}\pi R^2 - 1) \times \left\lfloor \frac{n+k-2}{k} \right\rfloor \times d_{avg} \tag{6.5}$$

Multihop broadcasts can result in more responses from candidate devices but at the same time the devices that have already been instantiated will not respond to future broadcasts. Because of these two opposing effects, the exact calculation of $n_{\texttt{C\_RESP}}$ in a succinct closed form is difficult. The important point here is that the slope of the curve is influenced by the spatial area density of devices in the network. We plot the unicast control packet overhead in Figure 6.20 and observe that it is almost linear in n. The overhead is slightly lower for higher values of k since the number of broadcasts (and hence the number

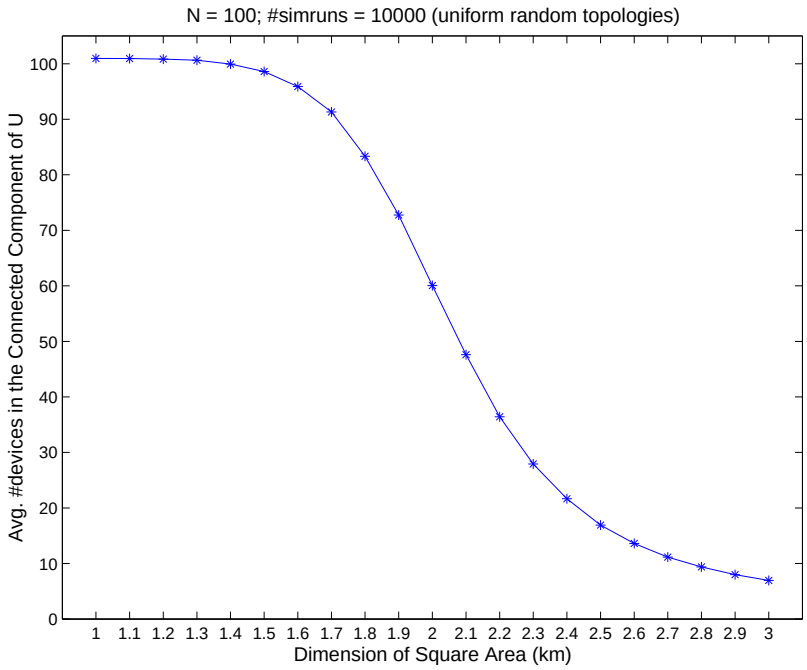


Figure 6.22: Effect of Spatial Node Density on the Size of the Connected Component of U

of responses) as well as number of `SUBTREE_CONF` packets is lower.

The byte overhead of the instantiation protocol is plotted in Figure 6.21. We can observe that the per device byte overhead increases linearly with n. This is because the size of the task graph increases linearly with n and since the TGs are encapsulated into `ACK` packets for instantiation, the per device byte overhead increases linearly with n as well. For this reason the total byte overhead increases quadratically. If a recursive definition of the tree task graph is possible, the per node byte overhead is likely to be constant.

6.2.2 Effect of Variation in Spatial Node Density

In this research we restrict the definition of instantiation in the sense that only one node in a given task graph can be mapped to a physical device in the network.[4] Under these assumptions, the size of the network restricts the size of TG that can be instantiated. We

[4]However, one device is allowed to participate in multiple tasks if possible.

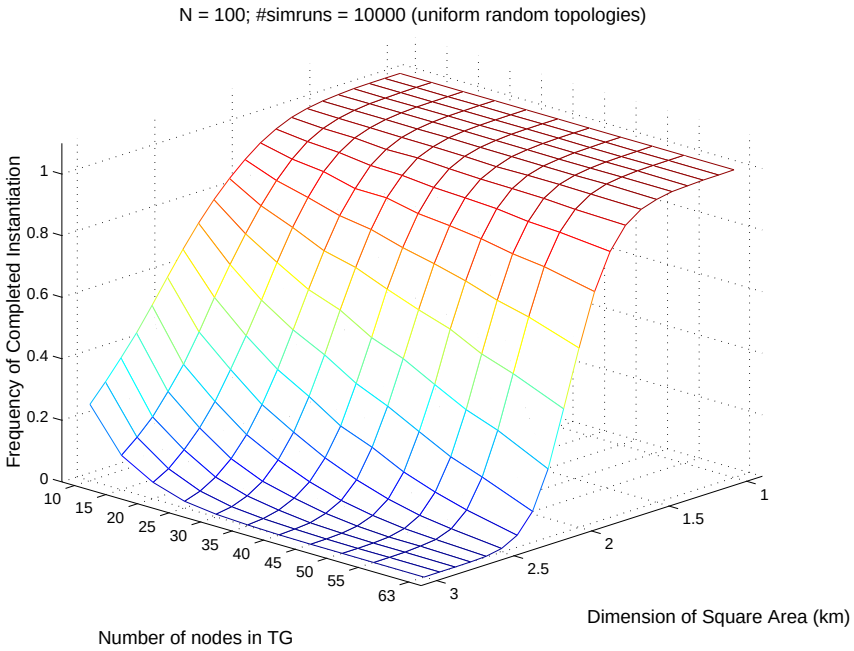


Figure 6.23: Effect of Spatial Node Density on the Probability of Instantiation

realized that connectivity of resources is the base minimum requirement for task instantiation. In theory a user device is capable of instantiating a TG if it can *reach* at least as many resources in the network as there are logical nodes in the TG.[5] In graph theoretic terms, if the connected component of devices in the network containing the user device has at least as many nodes as the size of TG, then instantiation is possible.

The sizes of connected components is primarily affected by physical parameters such as spatial density and mobility. In Figure 6.23 we show the effect of variation in spatial density of nodes on the probability of successful instantiation. Spatial density is varied by keeping the number of devices in the network constant ($N = 10$) and varying the dimensions of the square area (d) in which the devices are uniformly (randomly) distributed between 1 km and 3 km. The transmission range is fixed at 250 meters. We generated 10000 different topologies with (x,y) coordinates of devices distributed randomly and uniformly between 0

[5]In practice, reachability does not guarantee instantiation due to imperfection in network and transport protocols.

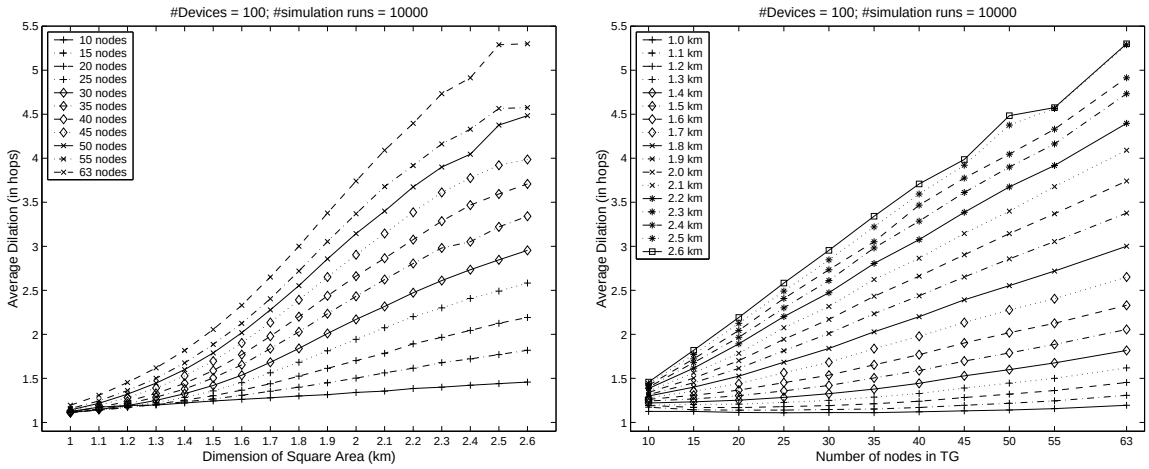


Figure 6.24: Dilation of a Binary Tree in Random Static Topologies

and d, and conducted one simulation run on each topology. We instantiated binary trees of various sizes on these randomly generated static topologies and recorded the number of times instantiation was possible.

We observe from Figure 6.22 that the size of the connected component of the user device U starts decreasing at a sharp rate when d is increased past 1.7 km and then starts to taper off when the dimension is increased past 2.4 km. Essentially, increasing the area gradually fragments the network into multiple components. For $1.7 \leqslant d \leqslant 2.4$, the size of the connected component containing U drops sharply than for other values of d. This causes a sharp drop in the probability of successful instantiation of large task graphs as shown in Figure 6.23. This analysis helps us to estimate the regimes where instantiation is possible. Note that the drop in probability of successful instantiation is more gradual for trees of smaller sizes (~ 10). This is because the probability that the size of the connected component of U is greater than a low value does not drop drastically.

We also calculate the average dilation of embeddings in this analysis; it is plotted in two different styles in Figure 6.24. From the plot on the left, we observe that for each size of the task graph, average dilation first increases with increase in dimension of square area (d) and then flattens. This phenomenon can be explained by the shape of the curve in Figure 6.22. As d increases, the average size of the connected component containing U reduces, and

Table 6.3: Simulation Parameters (Homogeneous Case with Mobility)

Simulation Parameter	**Value**
Number of Devices, N	100
Simulation Area, A	1000×1000, 1414×1414, 2000×2000
Transmission Radius, Tx	250m
Mobility Model	Random Waypoint
MANET Routing Protocol	Dynamic Source Routing (DSR)
#Classes of Devices, $\|C\|$	1 (homogeneous)
Size of TG: ($\|V_T\|$)	$\{3, 7, 15, 31, 63\}$ (complete binary trees)
Simulation Period	400s
MaxSpeed, v_{max}	1, 5, 10, 15, 20 m/s
MinSpeed, v_{min}	0 m/s
Pause Time, PT	0, 100s
Data Traffic Pattern	CBR: burst: 250 bytes; period: 1.0s

the richness of connectivity in the connected component reduces too. This results in TG edges being mapped to paths of greater lengths, and hence the average dilation increases. For larger task graphs, the curves taper off as d increases further because in most such cases, instantiation was unsuccessful and hence the average dilation calculation is performed over much smaller number of samples than in the other cases.

Another observation is that at any value of d the average dilation increases linearly as the size of the tree increases (see Figure 6.24(b)). This is a result of the greedy nature of the GreedyEmbed protocol. After nodes at higher levels in the tree TG are greedily embedded to devices near each other and the user device, the nodes at lower levels in the tree tend to be embedded at greater distances from their parents since the devices near the parent may all have been instantiated earlier. Hence, the average dilation increases with the size of TG at any given density. The linear slope is a result of the uniform distribution of devices.

We conclude from this analysis that low spatial density of nodes beyond a certain threshold can cause network fragmentation and reduce the probability of instantiation severely. This threshold can be visualized qualitatively for each size of TG as the lightly shaded area of the mesh plotted in Figure 6.23. Even if instantiation is possible, the resulting embedding is likely to have a high dilation which would drastically affect other critical metrics such as throughput and delay.

6.2.3 Performance Evaluation under Mobility

To conclude this chapter we describe the simulation results of our protocols under the effect of simultaneous variation in physical parameters such as mobility as well as spatial node density. The simulation parameters for this study are outlined in Table 6.3. We investigated three different spatial node densities in order to corroborate some of the predictions that we reported about the possibility of instantiation in Section 6.2.2. 100 devices are randomly distributed in square fields of areas $1, 2$ and $4km^2$. If the devices' locations obey uniform probability distribution, each device has approximately 18, 9 and 4 neighbors respectively. We refer to these three scenarios by *dense*, *moderately sparse* and *sparse*, respectively. We fix $v_{min} = 0$ which means $v_{avg} = \frac{1}{2}v_{max}$. Also, we kept the traffic injection rate low in these studies since we did not want the networks to be congested because of high aggregate traffic loads, as that situation can easily happen for tasks involving 63 nodes for greater application data-rates.

Instantiation Time We first study how long it takes to instantiate task graphs of different sizes on each of these MANETs. Figure 6.25 shows scatter-plots of instantiation times for each TG for both the constantly mobile case ($PT = 0$s) and the temporarily static case ($PT = 100$s). The dashed line joins the median values. The scales are same for each value of $|V_T|$ (for both values of PT). We can see some clear phenomena from the set of plots:

- The bimodal behavior observed in the static case (see Section 6.2.1) is observed again

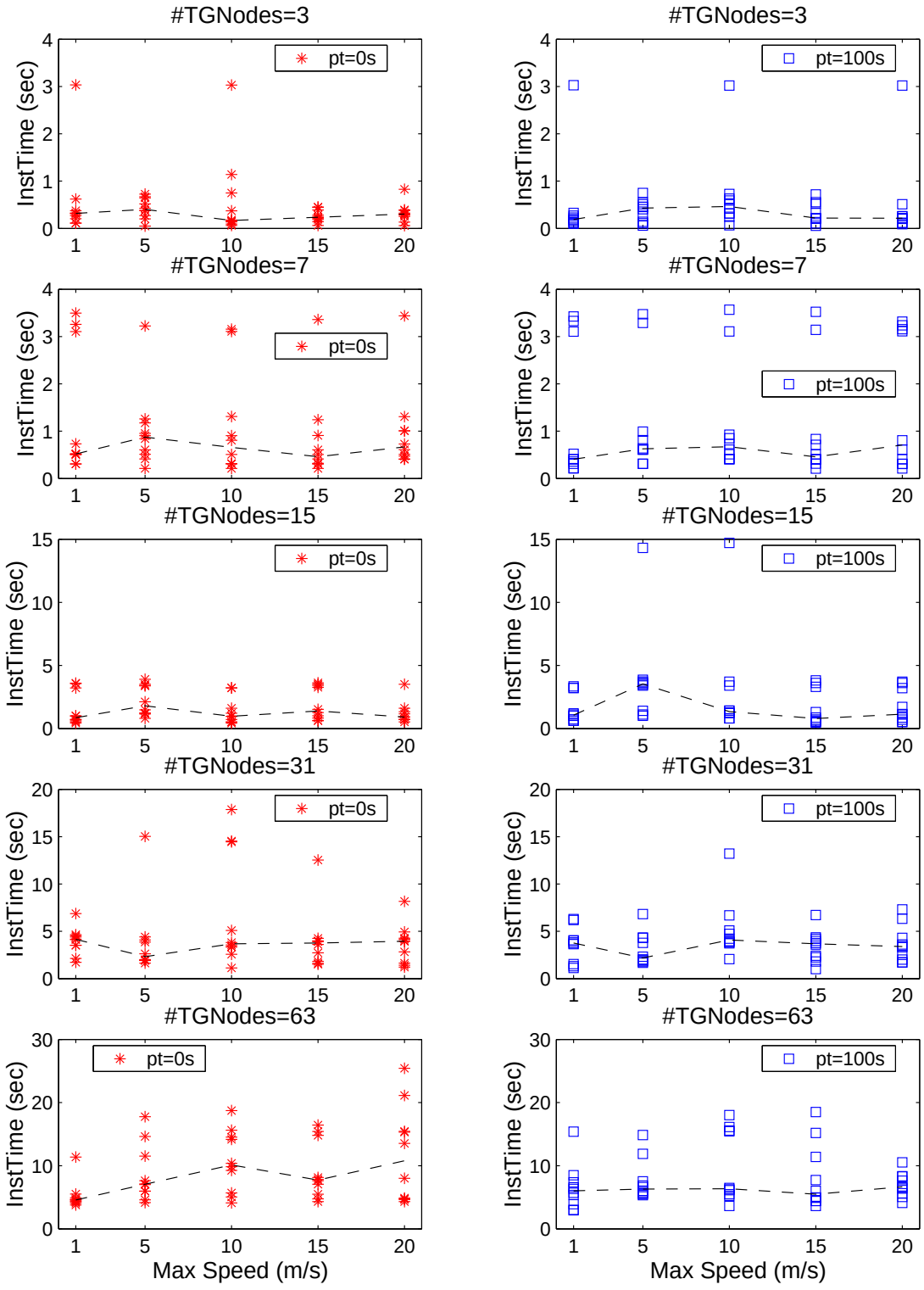


Figure 6.25: Instantiation Time (dense networks)

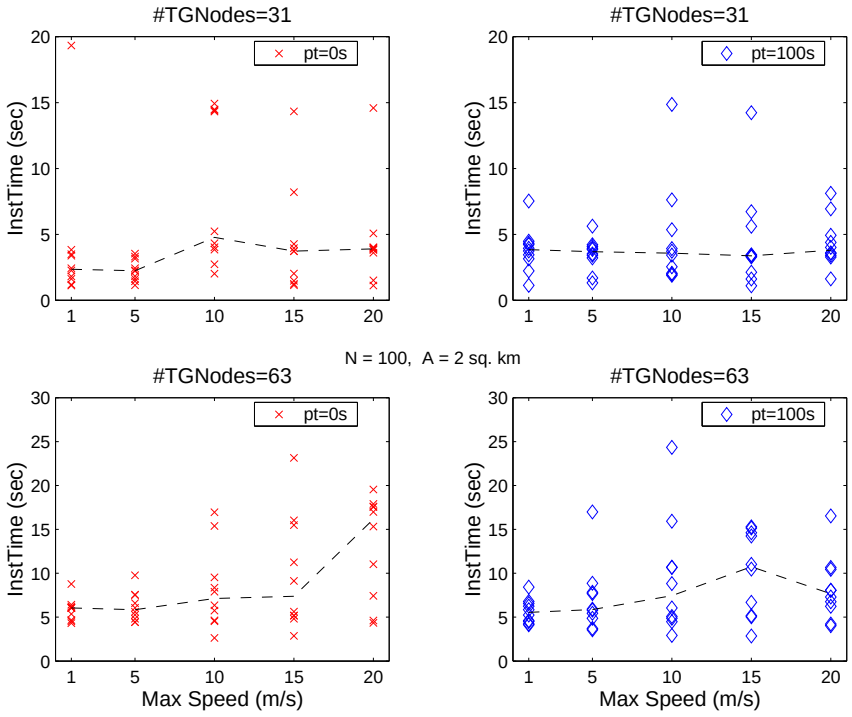


Figure 6.26: Instantiation Time (moderately sparse networks)

in this scenario. For $|V_T| \in \{3, 7, 15\}$, we observe that although most instantiation times are distributed in the sub-second mark, some values are distributed around the 3s mark. This was traced to TCP timeouts in all the cases. As the size of the TG increases, there are more TCP timeouts, and hence more values occur near the 3s mark.

- The median value is not heavily affected by device mobility with the exception of the case with $(|V_T| = 63, v_{max} = 20, PT = 0)$ which is the case of instantiating the largest TG in a scenario with the greatest mobility. In that scenario, the median instantiation time is above 10s.

- For $|V_T| \geqslant 15$, we observe a few values appearing around the 15s mark. These correspond to the cases where the instantiation protocol was disrupted by mobility and local patching helped the process complete successfully. Since the `HELLO_PERIOD` was chosen to be 14s for this set of experiments, several InstTime values are distributed

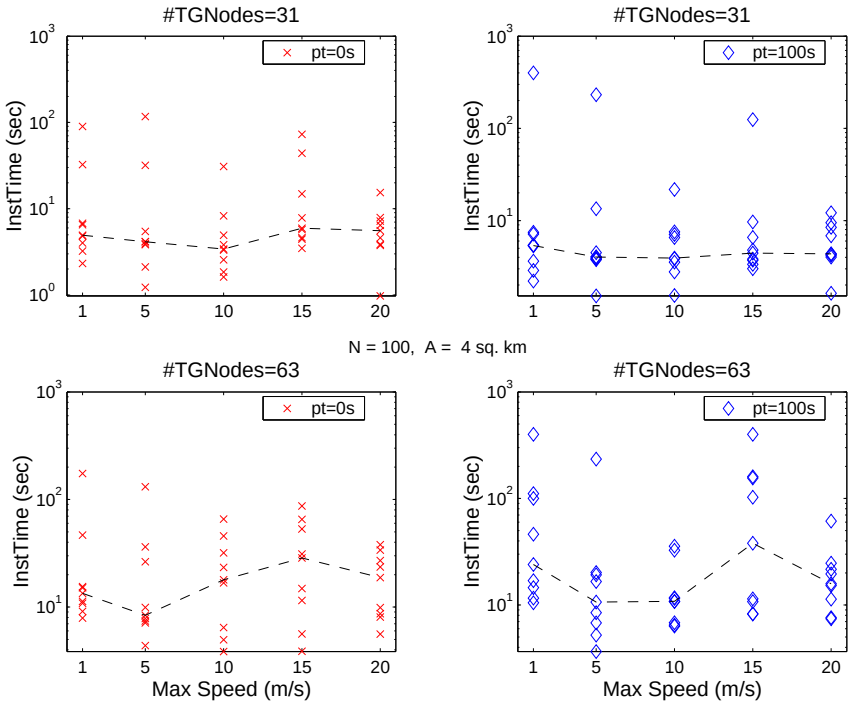


Figure 6.27: Instantiation Time (sparse networks)

around 15s.

We attempted to instantiate these task graphs using the centralized approach proposed by us elsewhere [50] but observed that the centralized algorithm was simply not able to instantiate task graphs with 63 nodes. This is because the root node which acts as a central controller in that algorithm is overwhelmed with traffic and suffers from heavy packet losses due to heavy congestion. It actually fails to instantiate disconnected nodes even after trying several times. This in fact illustrates the effectiveness of the distribution of the control of the instantiation process to children nodes as well as that of failure recovery techniques proposed in Section 5.3, namely local detection of disconnections and TG-patching.

Figure 6.26 shows the instantiation time values for moderately sparse networks (square with side 1.414 km). We only plot the cases of $|V_T| = 31$ and 63. We observe behavior very similar to that observed for dense networks. Like in the dense case, in the

most extreme scenario, i.e., ($|V_T| = 63, v_{max} = 20, PT = 0$) we observe that the instantiation times are higher. However, from Figure 6.27 it can be observed that instantiation times are much greater in sparse networks (square with side 2 km). We plot the Y-axis in logarithmic scale to illustrate the range of values and the variations. This analysis shows that the spatial density of nodes in the MANET is more important in governing the instantiation process than device mobility itself. From the analysis in Section 6.2.2, we observed that for a static uniform distribution of nodes, the number of nodes in the connected component of the user node starts dropping drastically for $d \geqslant 1.7$ km. For this reason, we see different results for $d = 2$ km than for $d = 1$ km or $d = 1.414$ km even in the mobile case. The network is partitioned on several occasions for every simulation run with $d = 2$ km, and sometimes these events result in failure of instantiation (values equal to 400 seconds in the figure).

To analyze this situation further we decided to measure the unreachability of devices in a mobile network via the size of the connected component of the user device as a function of time. Figures 6.28– 6.31 plot this quantity for the 10 simulation runs for a few representative values of v_{max} and PT. We plot these curves starting at 200 seconds since our instantiation process begins at that time. It is easy to find correspondence between these curves and the corresponding instantiation plots of Figure 6.27. For example, for ($|V_T| = 31; v_{max} = 5; PT = 0$), the higher instantiation times (see Figure 6.27) correspond to the third and the eighth scenarios in Figure 6.28. For ($|V_T| = 31; v_{max} = 5; PT = 100$), it is the eighth scenario in Figure 6.29. Although MaxSpeed and PauseTime do not seem to significantly affect instantiation time, they do have an impact on the throughput; this phenomenon is described later in the chapter.

Average Effective Throughput As mentioned in Section 6.1 this metric is affected most by mobility factors in the MANET. Figure 6.32 depicts the behavior of cumulative effective throughput with progression of time. Each curve corresponds to a random constant mobility scenario with specific parameters: $N = 100; A = 2km^2; v_{max} = 20m/s; PT = 0s$.

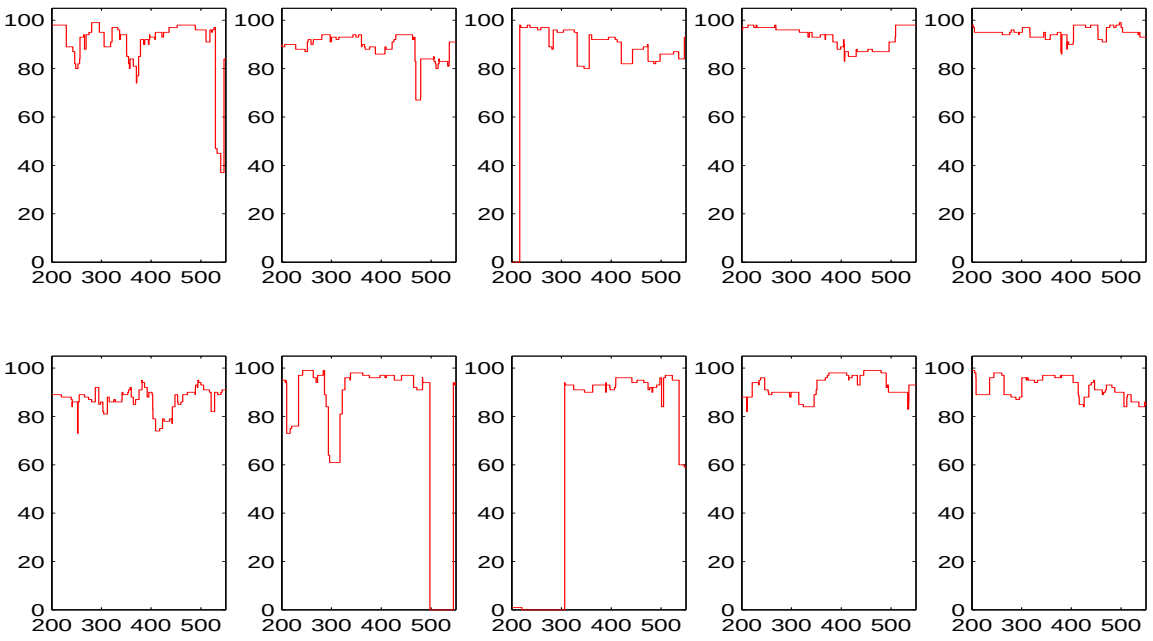


Figure 6.28: Size of the Connected Component Containing the User as a Function of Time ($N = 100, A = 4km^2, v_{max} = 5$ m/s; $PT = 0$ sec; 10 simulation runs)

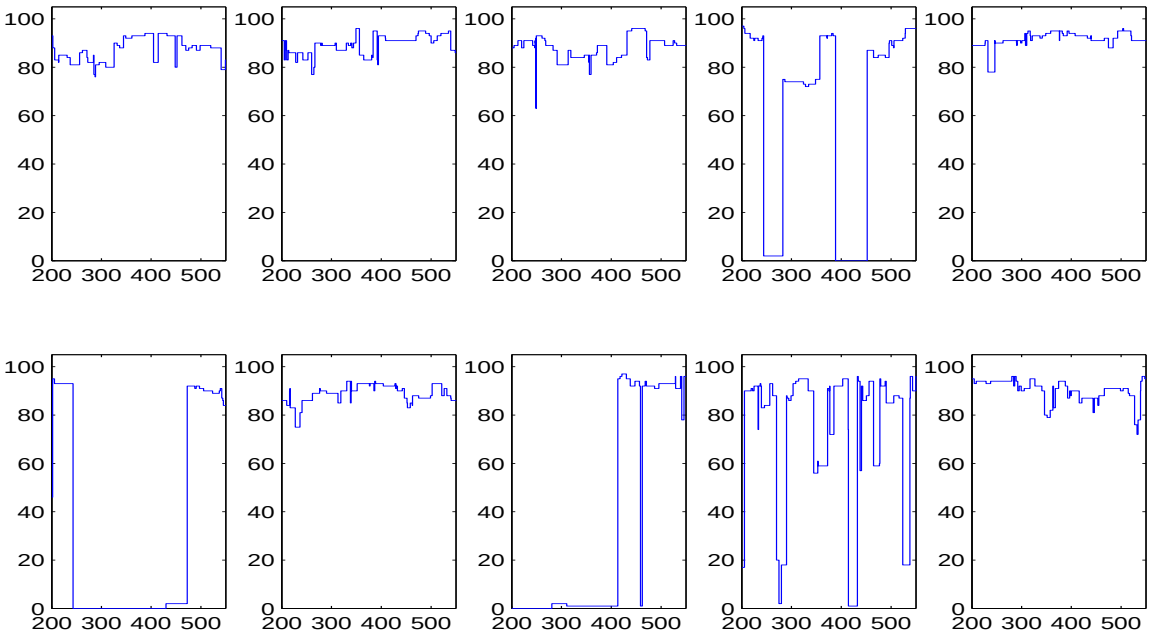


Figure 6.29: Size of the Connected Component Containing the User as a Function of Time ($N = 100, A = 4km^2, v_{max} = 5$ m/s; $PT = 100$ sec; 10 simulation runs)

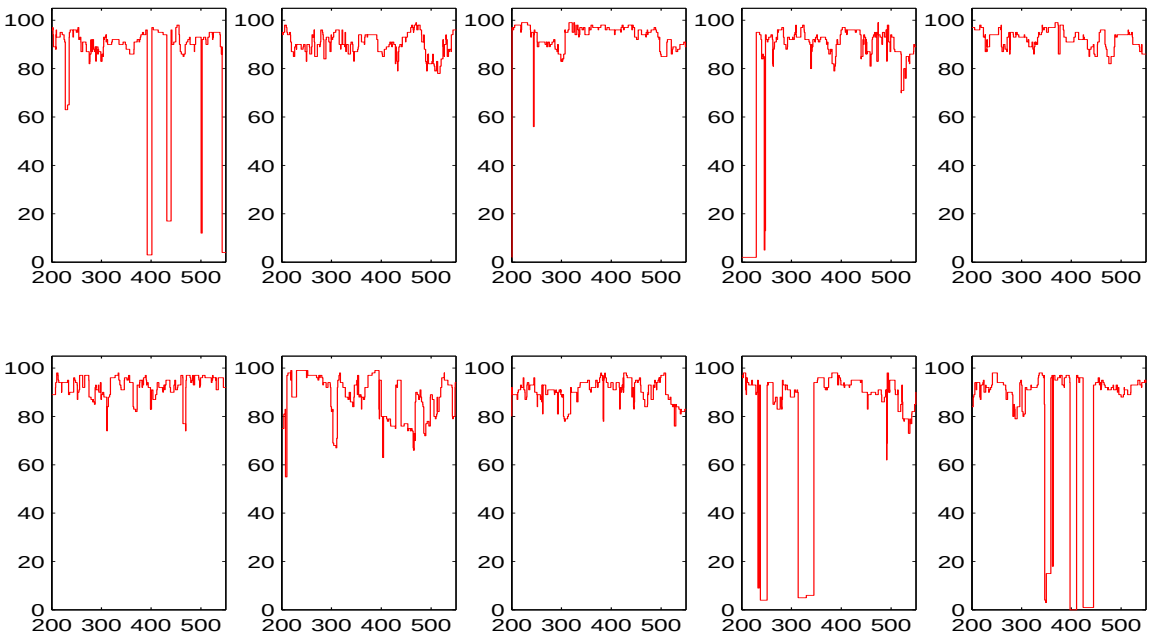


Figure 6.30: Size of the Connected Component Containing the User as a Function of Time ($N = 100, A = 4km^2, v_{max} = 20$ m/s; $PT = 0$ sec; 10 simulation runs)

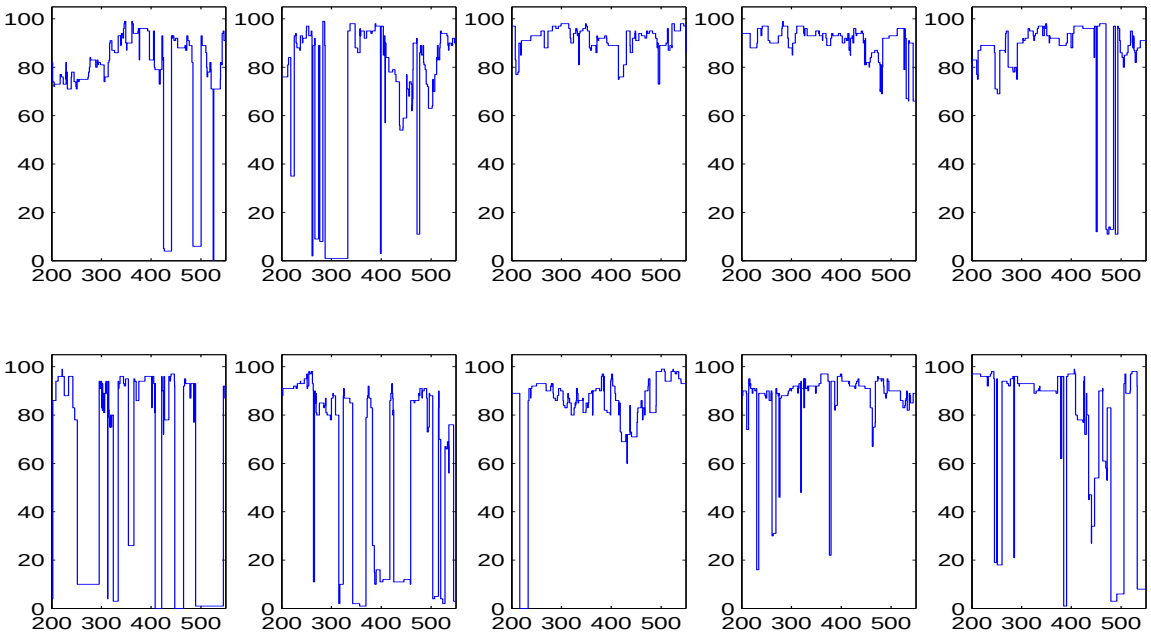


Figure 6.31: Size of the Connected Component Containing the User as a Function of Time ($N = 100, A = 4km^2, v_{max} = 20$ m/s; $PT = 100$ sec; 10 simulation runs)

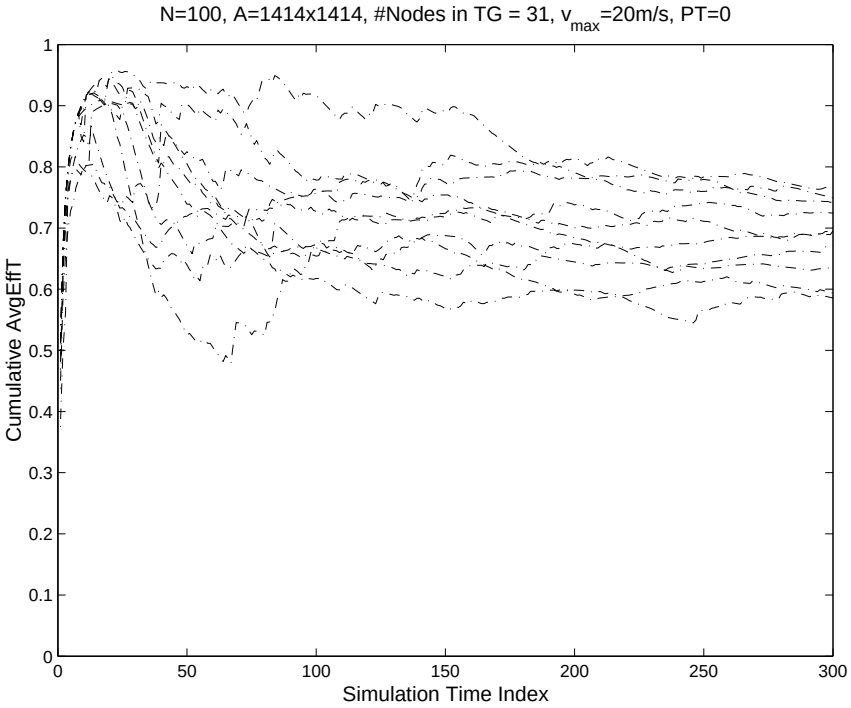


Figure 6.32: Cumulative Effective Throughput vs. Simulation Time (10 runs)

We observe that the throughput is quite high in the beginning (following instantiation); then due to constant device mobility, the data-flows get disrupted and the cumulative throughput drops in most mobility scenarios. However, cumulative throughput then increases and eventually stabilizes to a value in the 0.6–0.8 range. This is a result of the disconnection detection, re-instantiation and TG-patching protocols described in Section 5.3.

The 3-dimensional plot in Figure 6.33 illustrates how MaxSpeed and PauseTime can affect effective throughput even in dense networks for various sizes of TG. We can clearly observe a linear fall in average throughput with increase in MaxSpeed (v_{max}) for any given size of TG. This is observed for both constant mobility scenario ($PT = 0$) and the temporarily static scenario ($PT = 100$), the only difference being that the slopes of the curves in the former case are slightly steeper.

Although the network is almost always connected in the dense scenario, rapid, frequent motion of devices (at higher values of v_{max} and low values of PT) causes frequent

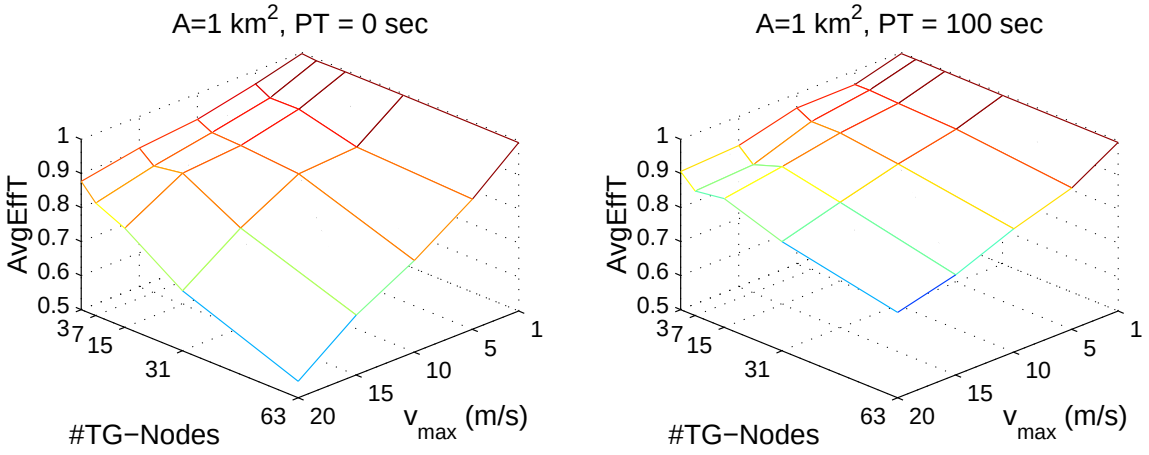


Figure 6.33: Effective Throughput vs. Size of TG (dense network)

disruption of routes and triggers route discovery. Sometimes the route changes are so frequent that TCP is unable to deliver the `HELLO` message to the appropriate logical neighbor in the `HELLO_PERIOD` time duration. With the route cache option *on*, DSR would try to use cached routes which have a high probability of being stale in dense networks especially for if they are long. This can naturally result in several TCP timeouts. Since TCP treats the event of failure in delivery as a sign of congestion, it backs its retransmission timer off exponentially and that reduces the probability of successful delivery of the `HELLO` message. In the worst case, this can result in a re-instantiation and hence loss of some throughput. The fall in average effective throughput is linear with increase in v_{max} because the average number of re-instantiations also increases linearly with increase in v_{max} (See Figure 6.34). Also, we observe from Figure 6.35 that it usually takes less than 1s time to re-instantiate a node in TG. The samples corresponding to the 3s mark are caused by TCP dynamics explained earlier in this section.

Similar symptoms are reported in Section 6.1. As mentioned in that section, the solution to this problem involves the design of a transport protocol which retains TCP's semantics but is friendlier in MANET environments. This is out of the scope of this dissertation.

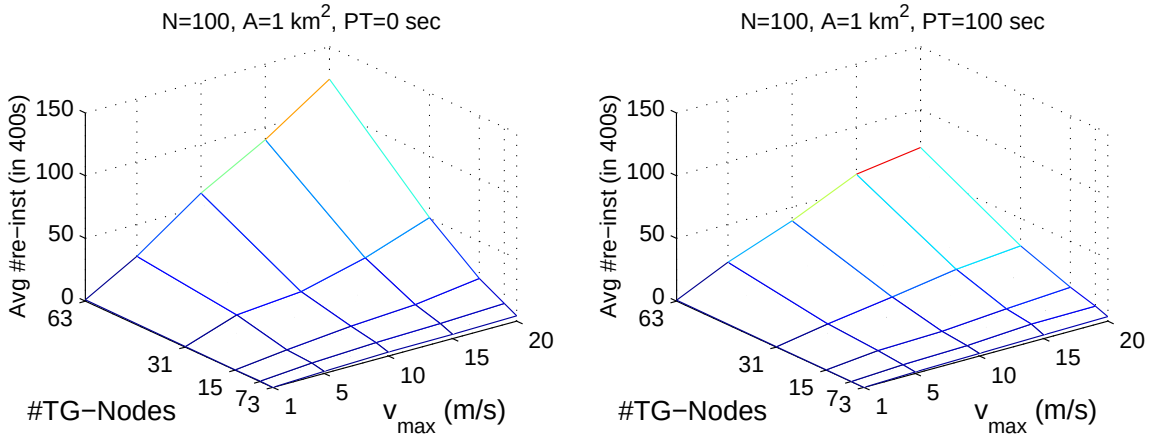


Figure 6.34: Average Number of Re-instantiations (dense network)

We also observe that effective throughput drops almost linearly[6] as the size of TG increases. The principal reason behind this is the stringent definition of throughput that we have imposed. In a tree TG, all the sinks are located at the leaf nodes and a contribution to throughput is counted only when an ADU originated at the user reaches a currently instantiated sink node. A 63 node binary tree TG has double the number of sink nodes than a 31 node binary tree possesses (32 against 16). Hence for a given mobility pattern (v_{max}, PT), the former is likely to have proportionally more disruptions than the latter and therefore the effective throughput drops accordingly.

Effective throughput curves for the moderately sparse network have been plotted in Figure 6.36. No significant variation can be observed from the results presented for the dense network case. However we do observe significant differences for the sparse network scenario. The scatter-plots in Figure 6.37 indicate that the variance in effective throughput is much greater for a sparse network scenario than its denser counterparts. The variance is particularly large for the higher PauseTime ($PT = 100$) for certain speeds such as $v_{max} = 5, 15,$ and 20 m/s. Figures 6.28–6.31 readily explain why this is so. From Figure 6.29 (corresponding to the $v_{max} = 5$ case), we can observe that for 4 scenarios (4th, 6th, 8th and 9th) either the network is fragmented for a long time or the user (root) device does not have

[6]Slope of the line is dependent on a function of various factors such as $N, |V_T|, v_{max}, PT$.

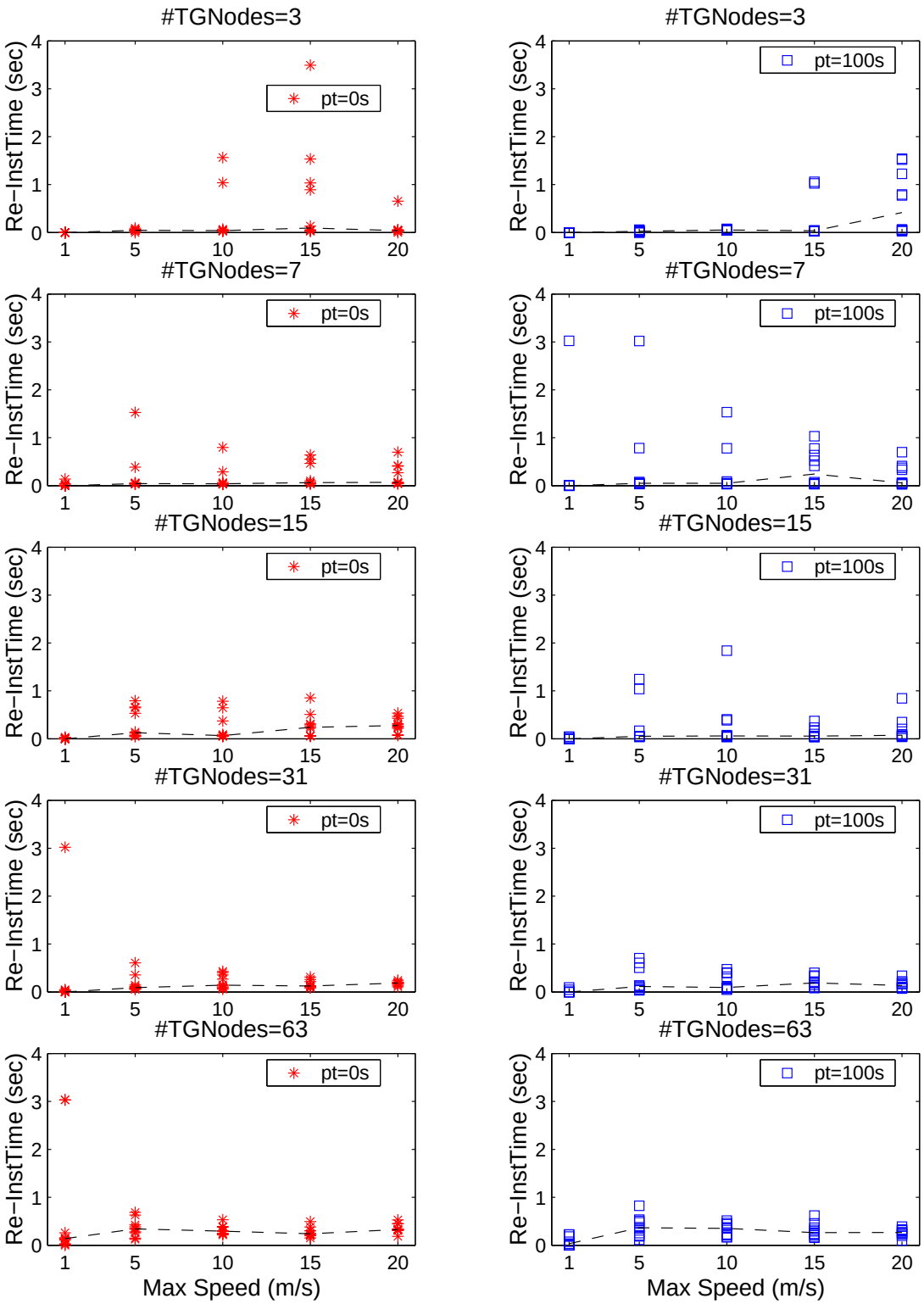


Figure 6.35: Time Taken to Re-Instantiate a TG Node (dense network)

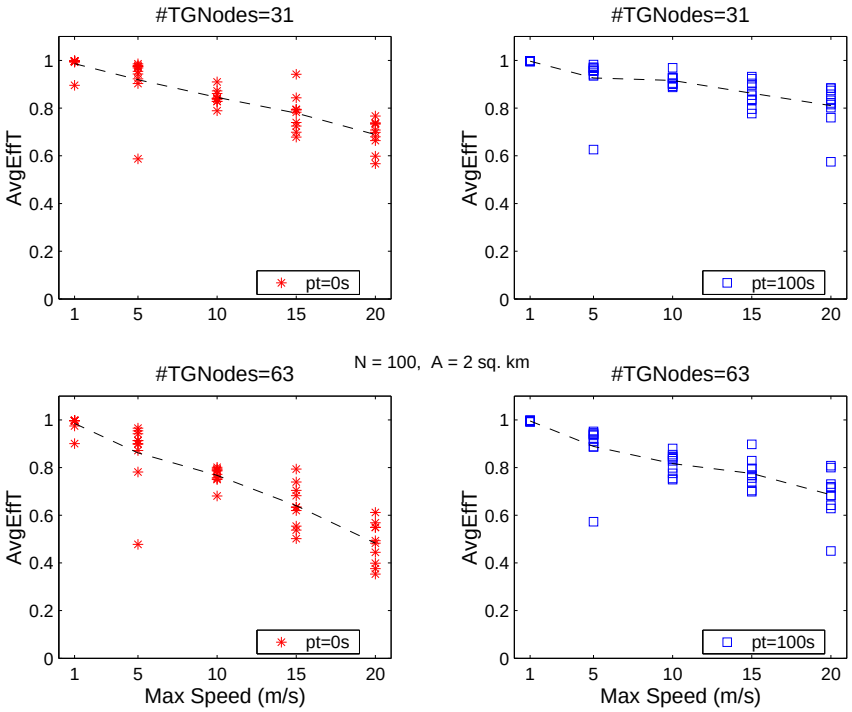


Figure 6.36: Effective Throughput in Moderately Sparse Networks

enough reachable devices in its connected component most of the time, thus hampering the task execution process. Since we compute effective throughput starting from the instant of completion of the initial instantiation, we only see 3 points in Figure 6.37 that correspond to low effective throughput.[7] Similarly from Figure 6.31 (corresponding to the $v_{max} = 20$ case), we observe several scenarios where there are severe disruptions in the network topology and hence more re-instantiations and low throughput.

We observe an additional phenomenon from Figure 6.37: contrary to the situation in denser topologies, constant mobility ($PT = 0$) in sparse topology yields better effective throughput than intermittent motion with temporary static phases ($PT = 100$ here). This is because in sparse topologies, islands are often formed and constant mobility often brings islands together forming connected networks with larger sizes. If motion is intermittent, although route changes occur less frequently, network partitions take more time to heal and

[7] Although instantiation time is high for the 8th scenario, its effective throughput is not low.

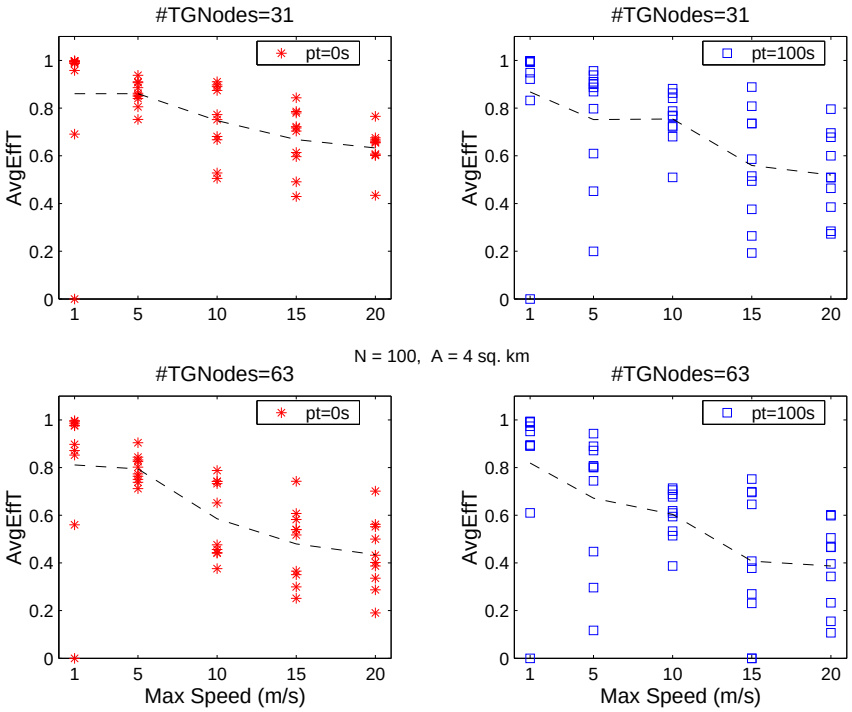


Figure 6.37: Effective Throughput in Sparse Networks

hence result in occasionally very low throughput.

Source-to-Sink Delay Another metric that captures the performance of the task execution protocol is the ADU delay (as defined in Section 3.4 and mentioned in Section 6.1). Along with the ADU delay we also plot the average number of hops traversed by an ADU in its flight from the user device (source) to a sink. Results for the dense network are plotted in Figure 6.38. From the "hops" curves, we can observe two things: (1) v_{max} and PT seem to have very little impact, and (2) number of hops traversed increases logarithmically with size of TG ($|V_T|$). The explanation for the second observation is that the number of hops traversed by an ADU is proportional to the depth of the TG (distance from source to sink) and also to the average dilation of the embedding. Actually on average an ADU needs to traverse $d_{avg} \times (\log_2 |V_T| - 1)$ hops from the source to a sink. For a static network, we showed by simulation in Section 6.2.1 that d_{avg} was slightly above 1.0 for binary trees.

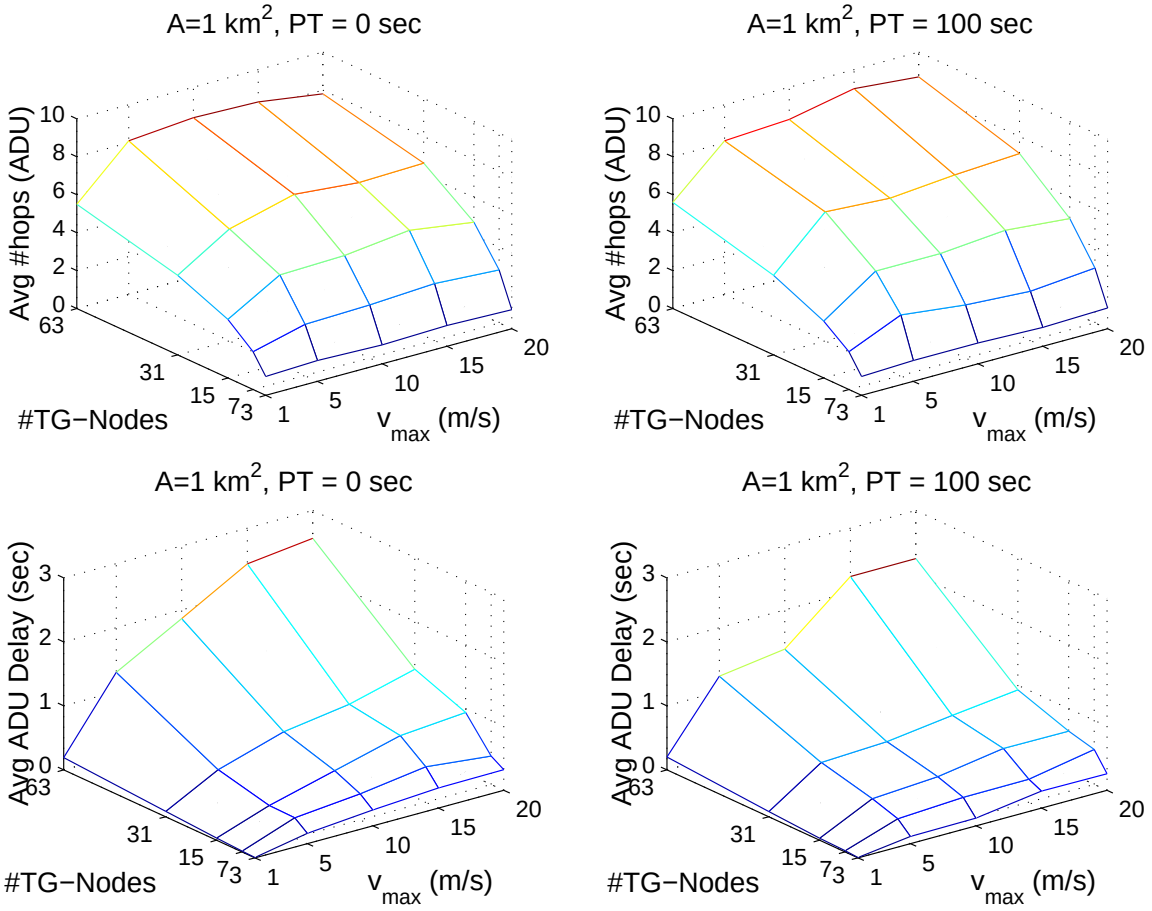


Figure 6.38: Source→Sink ADU Statistics: Hops Traversed and Delay (dense network)

This closely agrees with the number of hops traversed in the $v_{max} = 1$ case (least degree of mobility). However for higher degrees of mobility, the value of d_{avg} increases because the paths to which TG edges have been mapped at the time of instantiation are stretched due to motion. Therefore, the number of hops traversed slightly increases while preserving the logarithmic nature of the curves.

In an ideal situation, ADU delay over a MANET hop should be constant. Therefore the "delay" curves should also appear similar in shape as the "hops" curves (with a constant multiplicative factor). Unfortunately, that is not the case in a real MANET in which large fraction of devices ($\geqslant 63\%$ for $|V_T| = 63$) are participating in instantiation and subsequent

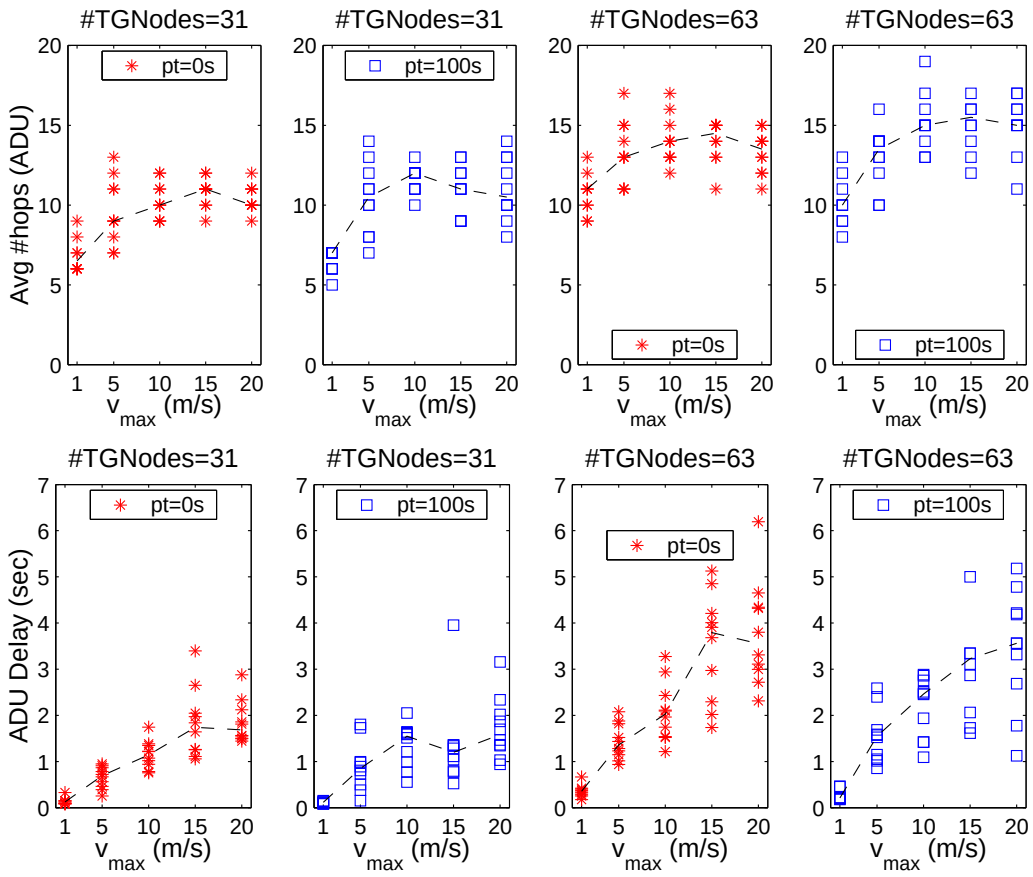


Figure 6.39: Source→Sink ADU Statistics: Hops Traversed and Delay (sparse network)

task execution. In fact, we observe from Figure 6.38 that the average ADU delay increases *linearly*, not logarithmically, as $|V_T|$ is increased from 31 to 63. The principal reason behind this is that ADUs are sent as TCP packets and TCP throughput drops drastically (hence delays increase accordingly) as length of the source–destination route increases in MANETs [42]. Another important reason is that for large task graphs, a large fraction of devices in the network participate in the task and that results in significant contention or even congestion in the network which in turn increase TCP delays.

Increase in v_{max} also results in more frequent route errors which in turn cause more TCP delays. This culminates in increased ADU delays. Therefore, delay sensitive tasks that are large are not ideally suited for highly mobile ad hoc networks, although they can

be perfectly executed on MANETs with low mobility (see delay values at low speeds in Figure 6.38).

From Figure 6.39 we observe that for the sparse network, "hops" values are greater than in the dense network case. This is because the network is sparser and the average dilation of the embedding is high. Consequently, the average ADU delays are larger than the corresponding values in the dense situation.

Protocol Overhead Finally, we study how protocol overhead (in terms of TG-layer control packets and bytes) is affected by mobility and spatial density. We measure TG-layer overhead similar to the way it is done in Section 6.2.1. For the dense network, the overhead curves are presented in Figure 6.40 in four categories. We observe that the number of originated broadcasts increases linearly up to $|V_T| \leqslant 31$ and increases at a faster rate beyond that. This is because the average dilation of the embedding found by the protocol tends to increase as the size of TG increases, thus resulting in more "expanding ring search" type broadcasts. Each broadcast that has a TTL value larger than 1 implies additional forwarded broadcasts, a fact that can be readily observed in Figure 6.40. Also the number of originated broadcasts increases with increase in MaxSpeed due to the occurrence of more re-instantiations.

We observe from Figure 6.40 that the unicast control traffic tends to dominate the protocol overhead and it increases linearly with $|V_T|$. The reasons for this are the same as those given in Section 6.2.1. The values are larger in these plots than in the ones in Figure 6.19 because we had only simulated the instantiation phase in the plots presented earlier. We observe a linear increase in unicast control overhead as well as byte overhead, as MaxSpeed is increased for the case of $|V_T| = 63$. This is simply a result of a linear increase in the number of re-instantiations (see Figure 6.34). The overhead plots for the sparse network are similar for the constant mobility scenarios (see Figure 6.41) but the overhead is much greater and non-uniform for the $PT = 100$ case. This can be attributed to the frequent

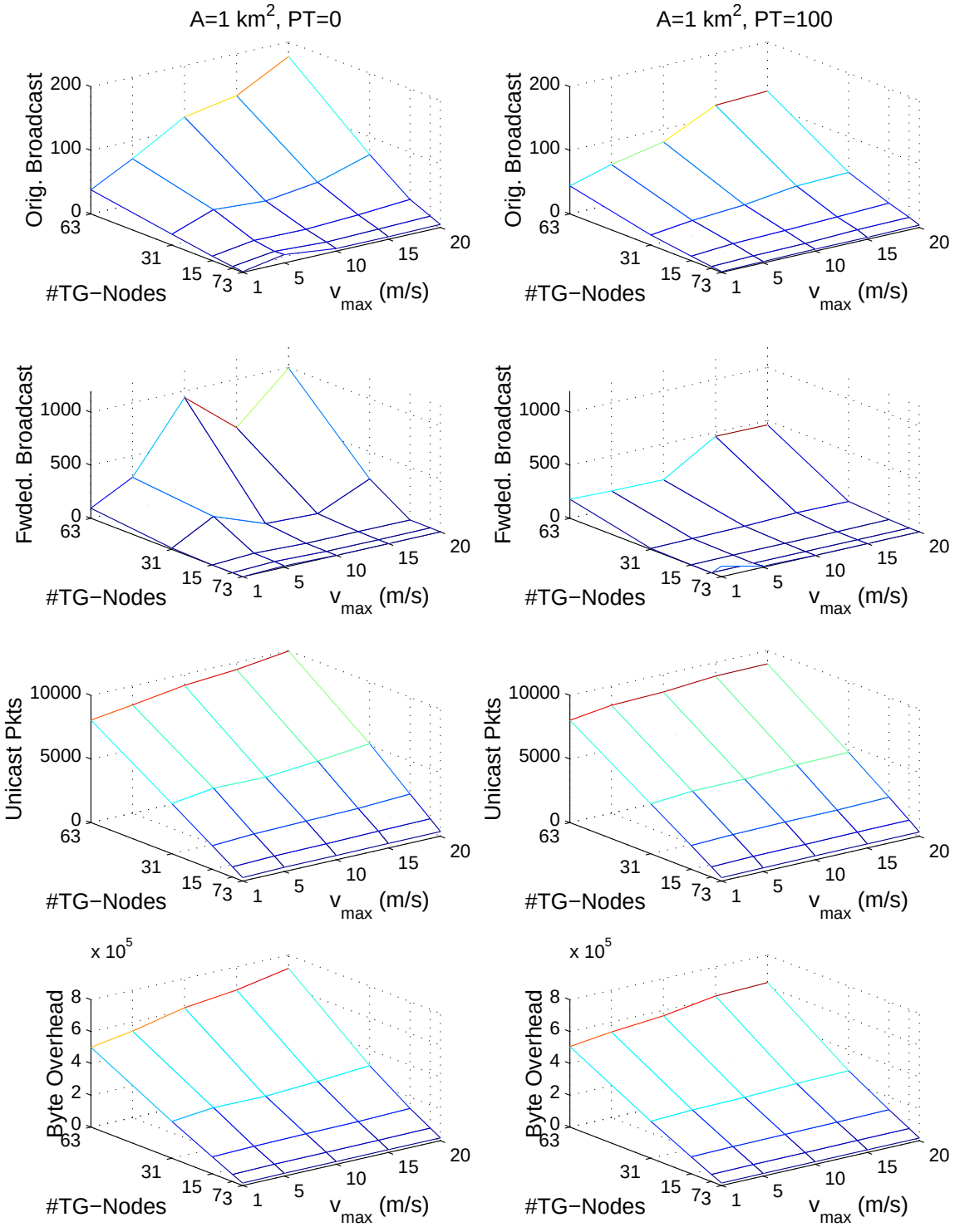


Figure 6.40: Average Protocol Overhead in Packets and Bytes (dense network)

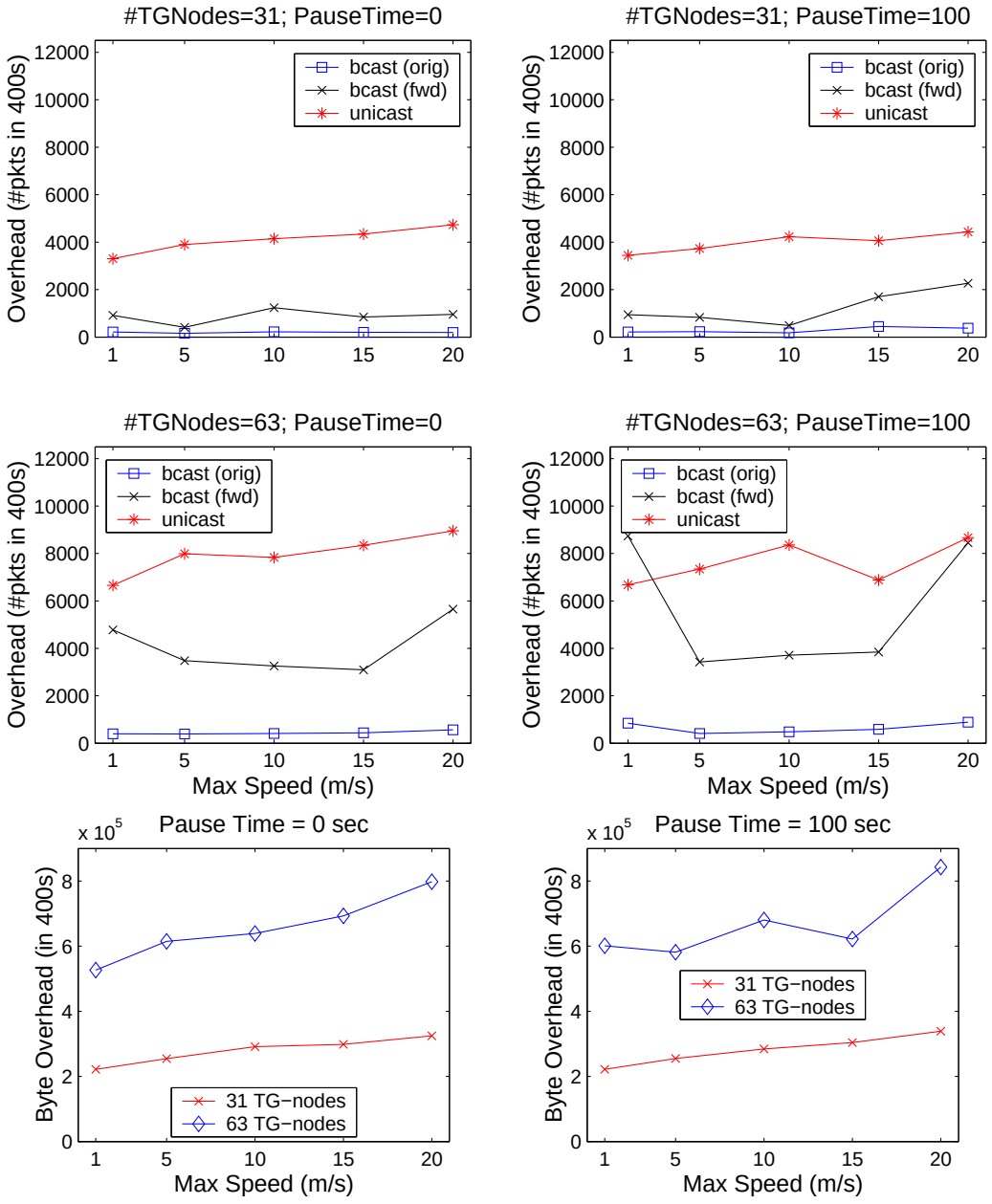


Figure 6.41: Average Protocol Overhead in Packets and Bytes (sparse network)

network partitions that occurred in that case (see Figure 6.31 for the ($PT = 100, v_{max} = 20$) case). Explanations for this phenomenon have been given before.

6.3 Discussion

In this chapter we reported detailed simulation results and analysis of our instantiation protocols for several different types of task graphs and networks while varying the spatial density and mobility of devices in the network. The primary motivation behind this analysis was to identify the factors that have the most impact on our instantiation algorithms. Thus, the analysis can prove useful for a system designer wishing to deploy certain mobile networking applications introduced in this dissertation.

We simulated heterogeneous as well as homogeneous task graphs. Heterogeneous task graphs were chosen to be smaller in size than their homogeneous counterparts. This was because distributed tasks that require cooperation of computing elements embodying heterogeneous functionality are likely to be less complex in size than tasks which have homogeneous components.

The analysis presented in this chapter is by no means exhaustive as one can envision many additional styles of task graphs in future applications. The networks supporting those applications may have very different characteristics from the ones studied here. For example, specific application scenarios may result in more predictable mobility models than pure random waypoint. We claim that the principal contribution of this chapter is the style of analysis with respect to the performance metrics proposed in this dissertation. This is independent of the exact network parameters, and hence can be applied to future systems with a possibly different set of parameters. In other words, as long as we have a MANET of devices running a reactive routing protocol, we can analyze our instantiation protocols with respect to the proposed metrics in much the same way as was done in this chapter.

From the simulations of the heterogeneous TGs we observed that our protocol is able

to instantiate and re-instantiate TG nodes quickly with low average dilation, and yield high effective throughput at low to medium mobility, and not much lower than 70% effective throughput for high mobility scenarios. In this work, we did not simulate heavy loads since standard TCP acts as a major bottleneck over multiple hops under heavy loads [42]. Our focus instead was to demonstrate the viability of the proposed dynamic task-based anycasting approach in MANETs using existing routing and transport protocols. We believe that much superior task execution throughput can be achieved at more demanding loads if standard TCP is replaced by feedback based schemes such as the one proposed in [22]. Future efforts should be directed towards such endeavors.

We also note that in some high mobility scenarios, the performance of our protocols suffered due to the non-aggressive retransmission timeout mechanism of TCP and aggressive route caching done by the underlying routing protocol, DSR. Introducing cross traffic load actually helps improve the effective throughput in such situations. Even in mobile scenarios experiencing no network partitions, disruptions in task execution are observed when periodic hello messages failed to be exchanged within a reasonably large time window ($T = 7$s or $T = 14$s). This typically occurs when the routes between instantiated neighboring devices grow very long and a route change occurs. The combined dynamics of MANET routing and TCP then result in slow recovery from route errors and hence uninstantiation. Although this process automatically maintains a low average dilation (since the re-instantiation protocol is likely to discover a device nearby), one has to pay a cost for this: the application suffers a disruption during the re-instantiation process.

We observe that for large task graphs of 63 nodes (by MANET standards), instantiation times are dominated by TCP timeouts even in both static and mobile scenarios. Most of these delays were caused not due to congestion but due to drops from interface queues due to high channel contention. This usually happens when a large fraction of the devices in the network participates in a task. One possible solution to this is modifications to TCP to make it resilient to contention based packet drops as suggested in reference [33].

We observe little change in performance of our protocols when the spatial density was reduced from 18 neighbors per device (100 devices in 1 km^2 area) to 9 neighbors per device (100 devices in $2km^2$ area) but there is drastic change in throughput when the spatial density was further reduced to 4 neighbors per device (100 devices in 4 km^2 area). This is because in the last case the network occasionally became so sparse that the connected component of the user did not have enough devices to continue executing the task. This however does not mean that no task can be performed in sparse networks successfully. If the tasks do not have long lifetimes, even a sparse network can support them with a reasonably high probability which can be estimated for given parameters such as MaxSpeed and PauseTime (say, from Figure 6.28). Also, in sparse scenarios, device mobility helps in healing network partitions and hence is beneficial to task instantiation. In contrast, in dense scenarios, it has a detrimental effect on effective throughput as it results in frequent route changes and re-instantiations.

Chapter 7

Design and Implementation of a Proof of Concept Prototype

The simulations presented in Chapter 6 demonstrate that the distributed task instantiation protocols proposed in this dissertation operate satisfactorily and yield desirable performance in a variety of mobile scenarios even when reasonably large task graphs are instantiated on large mobile ad hoc networks. To further validate the proposed concepts, we also developed a proof of concept prototype to demonstrate the viability of the solution. The salient features of the prototype and testbed are described here.

7.1 Overview of the Experimental Testbed

Hardware Our experimental testbed is comprised of the following hardware components:

- 3 desktop computers (HP Vectra and Gateway)
  - CPU: x86 architecture with clock speed: 200 MHz – 866 MHz
  - RAM: 128 MBytes – 512 MBytes

- 7 portable notebook computers (Dell Inspiron 2500 and 4000 Series)
  - CPU: x86 architecture with clock speed: 900 MHz – 1 GHz
  - RAM: 128 MBytes – 256 MBytes
- 5 handheld computers (Compaq iPAQs)
  - CPU: StrongARM architecture with clock speed: 206 MHz
  - ROM: 32 MBytes
  - RAM: 64 MBytes

Operating Systems All hosts in the experimental testbed execute different versions of the same Linux operating system. More specifically, the notebooks and the desktop machines run different versions of the Linux 2.4 kernel, and the handheld computers run a version of the Linux 2.4 kernel ported for the StrongARM architecture by Compaq Research Laboratories [31].

Networking Hardware and Software Each of the above 15 units is equipped with Cisco Systems 350 Series Aironet wireless PCMCIA cards which are connected to the handheld machines through expansion sleeves. The radios in the wireless cards operate in the unlicensed ISM 2.4GHz frequency band using Direct Sequence Spread Spectrum (DSSS) at a data rate of 11Mbps (with fall-back rates of 5.5, 2, and 1Mbps). The radios are tuned to the same frequency and their transmit power is programmable (varies from 1 mW to 100 mW). All wireless cards support the IEEE 802.11b standard; they are operated as an IBSS (Independent Basic Service Set) using the DCF (Distributed Coordination Function) mode thus forming an ad hoc network. All versions of the Linux kernel used on testbed hosts have full wireless networking support for the Cisco 350 Series PC cards.

Routing Protocol Support The ad hoc mode in IEEE 802.11b standard only specifies how hosts should communicate at the medium access (MAC) layer. A MANET routing

protocol is necessary to form a multihop wireless network. Hence we utilized public domain implementations of two different MANET routing protocols for our needs, namely AODV (from Uppsala University, Sweden) [54], and OLSR (from INRIA, France) [58]. Both the above implementations reside in the user space of the Linux OS. We were able to port these to the StrongARM platform (iPAQ) without incident.

Issues with Forming Multihop Networks in a Small Area One issue that we face during the deployment of our testbed is that of achieving multihop connectivity. Since all units are hosted inside our laboratory, even with all the cards operating at their lowest configurable level of transmission power (1 mW), we cannot attain multihop connectivity naturally without signal attenuation. Since we are primarily interested in demonstrating a proof-of-concept prototype and in testing the correctness of the protocol, we use the MobiEmu toolkit from Hughes Research Laboratory [71] to solve the problem.

The MobiEmu Toolkit The MobiEmu toolkit allows one to emulate a multihop MANET in the absence of multihop paths and real mobility. Each host that wants to participate in the MANET runs a MobiEmu slave. An additional host runs a MobiEmu master which is aware of the entire time-varying mobility pattern that we want to emulate using the slave hosts. The master and all slave hosts communicate with each other via IP multicast over either the same wireless channel or preferably over a wired network.[1]

The MobiEmu master **S** imposes multihop connectivity in an otherwise fully connected network by using certain features of IP packet filtering software (*iptables*) which comes bundled with all recent Linux distributions. This process can be illustrated with a simple example: suppose that at time instant t_i, host X is expected to disconnect from hosts Y and Z, but at $t_j > t_i$ it is expected to reconnect with host Y. To emulate this sequence of event, at time t_i, **S** multicasts a message that X and Y should disconnect and that X and Z should do so too. When X receives the message, it adds the MAC addresses of Y and Z

[1]This is to mitigate the effect of loss of control traffic information in a shared wireless network.

to its *block list* of addresses. Y and Z receive the same message and both put X in their block lists. Hence, at time $t_k : t_i < t_k < t_j$, if X sends a packet destined for Y or Z, then will be blocked by `IPtables` at the MAC layer and will not be passed up to the IP layer. Consequently X must look for a multihop route to reach Y or Z. Similarly, at time t_j, **S** multicasts a message to the entire group that X and Y should drop each other from their block lists. Shortly afterward, any intelligent MANET routing protocol would be able to figure out that a cheaper one-hop route is available between X and Y, and both X and Y should converge to that route.

In this manner, complicated mobility patterns can be emulated while using real hardware and real routing protocols without physically moving the hosts. However, we note that the above approach is useful only for establishing the correct operation of any distributed protocol; it is inappropriate for the measurement of protocol performance in terms standard metrics such as throughput, delay etc. This is because: (1) MobiEmu does not accurately capture the effects of physical phenomena that affect wireless communication such as cumulative effects of channel fading over multiple hops since all hosts are located in radio range of each other, (2) reliability of communication drops drastically when two hosts are near the boundaries of each other's transmission ranges, and (3) excessive packet collisions can result under situations of heavy load since all hosts form a complete graph with respect to physical layer connectivity. Nonetheless, MobiEmu serves as an effective tool for testing the feasibility of our protocols on real hardware under random mobility patterns.

7.2 Design and Implementation of a Task Based Application Framework

After the deployment of a testbed of modest size, we designed and implemented an application framework for task based resource discovery. The principal objective in this endeavor is

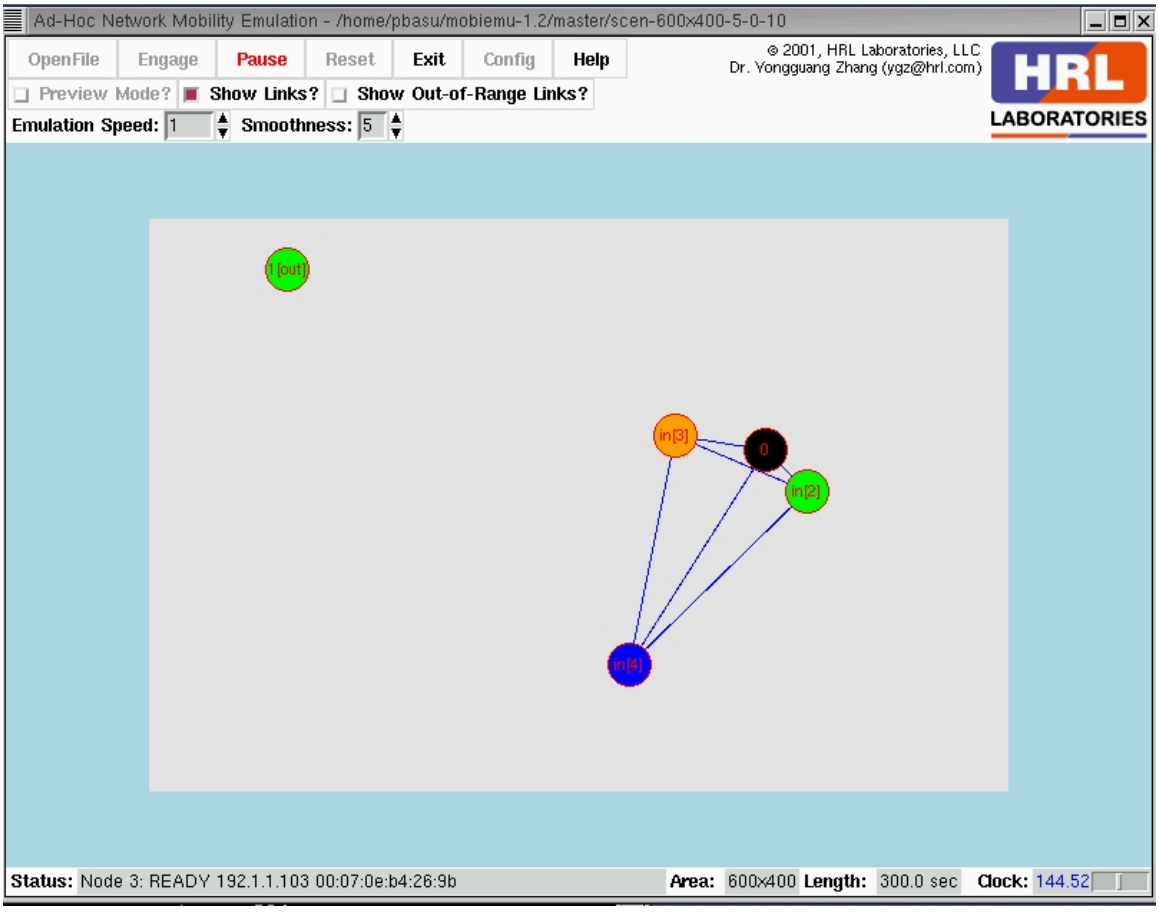


Figure 7.1: Screen-shot of MobiEmu Master Screen

to demonstrate that the algorithms and protocols proposed in Section 5.1 can indeed be implemented on a MANET comprised of devices with varying degrees of computing resources. Each host in the testbed emulates a resource that can be useful in a certain task that the user wants to accomplish. One host in the MANET emulates a user node that hosts a tuple representation of the task as outlined in Section 3.2. The tuple representations were chosen such that the sizes of the corresponding task graphs were smaller than the number of physical resources available in the network. In fact, multiple hosts are often made to emulate instances of the same resource. The hosts were deployed in both static and mobile configurations (real as well as emulated) in which the effectiveness and robustness of the discovery protocols were tested.

Figure 7.1 shows an image of the MobiEmu visualization screen at a particular time instant before which a four node task graph (same as the one used in Figure 5.4) has been instantiated and then re-instantiated. Of the five nodes in the MANET, device-0 belongs to the user, device-1 and device-2 possess identical attributes, and device-3 and device-4 possess distinct attributes. Originally devices 1, 3, and 4 were instantiated (depicted by `[in]`); at the time of screen capture, device-1 has moved out of range and hence has been uninstantiated (depicted by `[out]`). In its place, the corresponding TG node has been instantiated with device-2.

We demonstrated the following processes during this effort:

1. Quick instantiation of a task graph in both static and mobile scenarios with occurrence of changes in underlying MANET topology during the process of instantiation.
2. Timely delivery of task specific data to the currently chosen recipients.
3. Timely detection of disconnections between two participating hosts as a result of failure or mobility.
4. Rapid re-instantiation of the task graph with mostly local coordination, and resumption of task data delivery.

7.2.1 Design Decisions

We implemented the protocol finite state machine (FSM) outlined in 5.1 in the application layer of the networking stack of each host in the testbed, thus requiring no change to the existing Linux kernel. Linux socket APIs were used to facilitate communication between nodes in the testbed for exchange of both protocol and data messages. We decided to implement the task graph instantiation protocol module as a daemon process *taskd* which runs on each node in the testbed. *taskd* listens on a well known port `20345` for incoming protocol messages of type `S_QRY`. *taskd* is a single threaded daemon and uses `select()`

system call for I/O multiplexing. The advantages of `select()` based concurrent server design include (1) simplified design due to one logical control flow, and (2) the absence of thread or process control overhead [66]. *taskd* blocks on a `select()` while listening on a set of active socket descriptors. Whenever data arrives on any of the active socket descriptors, `select()` returns and allows the process to determine the identity of the socket which received data. Upon determining the above, appropriate action is taken on the incoming packet as prescribed by the protocol FSM.

The module corresponding to the distributed task initiated by the user which requires services and resources from other nodes in the MANET is referred to as a *taskapp*. A *taskapp* is aware of the corresponding task graph and it instructs its local *taskd* to instantiate it. The aforementioned inter-process communication (IPC) is achieved by means of a local UNIX domain socket.

After receiving the task graph from *taskapp*, *taskd* initiates the instantiation process by creating a datagram socket capable of performing IP neighbor broadcast or *neighborcast*. The broadcast capability is facilitated by the `setsockopt()` system call on the socket descriptor with the `SO_BROADCAST` option enabled at the `SOL_SOCKET` level. Any packet with the destination address set to `255.255.255.255` is intended to be delivered to all the reachable neighbors which receive it barring channel errors, collisions in the wireless medium, or buffer overflow in the interface queue. The broadcast packet is not re-forwarded at the IP layer by the neighboring nodes because each node in a MANET behaves like a single node LAN segment and the broadcast is limited to within the LAN only. However, while performing expanding ring search, a broadcast packet may be required to reach nodes that are farther than the one hop radius of the originator. This is achieved at the TG-layer by including a TaskTTL field in the packet which is decremented before neighborcasting the packet. The structures of both broadcast and unicast packets are illustrated in Figure 7.2.

The *taskd* daemon of a candidate device responds to the `S_QRY` packet by a UDP

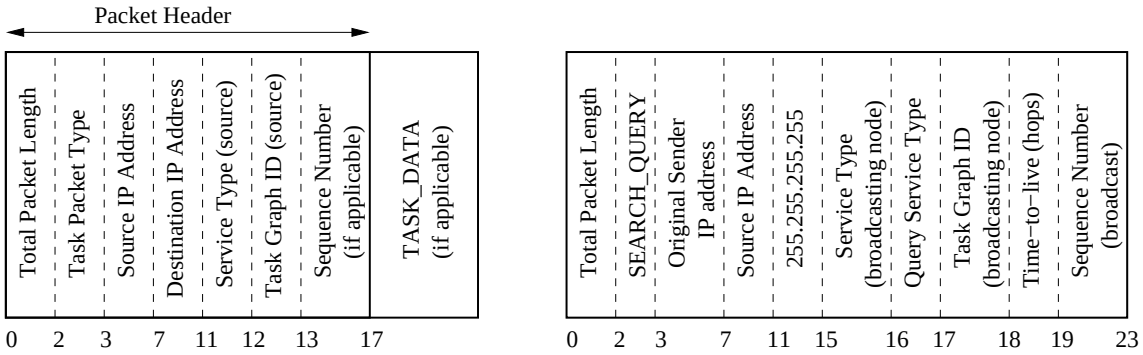


Figure 7.2: Protocol Packet Formats

`C_RESP` packet which can traverse multiple MANET hops using the underlying routing protocol. Since UDP is an unreliable protocol, a `C_RESP` packet can be lost in transit because of route changes due to mobility, or congestion due to buffer overflows in the interface queue or channel contention. We choose to use UDP for this purpose because our assumption is that multiple instances of a resource are normally available in the network, thus resulting in multiple responses to a query. In such a scenario, managing TCP connections to all candidate nodes is cumbersome and not warranted since the loss of a few UDP `C_RESP` packets does not significantly hurt protocol operation. However, if instances of a desired resource are fewer in number, a reliable transport protocol must be used.

After the selection (instantiation) of a TG node according to a certain instantiation criteria,[2] the local *taskd* daemon sends an `ACK` packet to the candidate over a reliable transport protocol. Reliable transport is needed in all subsequent steps of the protocol since packet loss impacts performance in those stages much more adversely than in the earlier stages. We experimented with two reliable transport protocols during the development of the proof-of-concept prototype: (1) standard TCP on Linux, and (2) UDP with mechanisms of timeout and retransmissions built into the TG-layer itself. We found that both approaches function well on our modest sized testbed where there was no network conges-

[2]In our experiments, this was minimum tardiness in response.

tion. However, the latter approach can have unforeseen impacts in large MANETs where network congestion is likely to be a norm than an exception [25].

An ideal reliable transport protocol for large MANETs preserves the congestion control semantics of TCP but at the same time is intelligent enough to track the cause of packet loss to one of the following: (1) route errors due to failures or mobility of devices, (2) bit errors in the channel, (3) drops from interface queue due to channel contention, or (4) buffer overflow signifying network congestion. As mentioned in earlier chapters, several researchers have begun investigating these problems for MANETs [42, 22, 33] but research on this topic has not yet yielded conclusive results.

An important issue while using TCP is that the `connect()` system call issued during the connection establishment phase can "block" due to lack of active routes to the destination. This can result in a complete halt of other protocol processing in *taskd* since the latter is single threaded. A remedy to this problem is to make the connection oriented socket non-blocking. In that case, however, `connect()` may return without the establishment of an actual connection, and the corresponding socket descriptor has to be monitored for *writability* by using `select()`. Data can be sent on the socket as soon as the connection is established and the socket becomes writable.

On the receiving end, *taskd* blocks on `select()` instead of `accept()`. In fact, when `select()` returns and it is determined that the socket descriptor corresponds to a new incoming TCP connection, `accept()` is called and it immediately returns with a new socket descriptor `sd`. *taskd* then adds `sd` to the "to-be-monitored" list of `select()`. Data is then read from `sd` which is then used for all subsequent communication between these two nodes.

A major point of difference between ns-2 simulations and real prototype development is that the protocol processing times and overheads are completely ignored in ns-2. In reality, these are indeed different on desktop PCs, notebook PCs and handheld computers. A main difference in the methods of using TCP in the ns-2 simulator and the prototype

implementation is in the mechanism of opening and closing connections. In the ns-2 simulator (version 2.1b7a), TCP connections between any two given endpoints are not closed after they were opened unless there is a timeout due to inactivity. This is because the absence of a "port mapper" in ns-2 made the repeated opening and closing process very cumbersome to handle. However on a real system running Linux, TCP connections must be dynamically opened and closed whenever necessary. In our implementation, a device closes its TCP connections in the following circumstances: (1) after it uninstantiates itself and transits to `UNINSTANTIATED` state, (2) after it detects disconnections from its instantiated children nodes, and (3) after the other end has issued a `close()` and a signal arrives at this end. After a TCP connection is closed, the corresponding socket descriptor is freed up and is released from the "to-be-monitored" list of `select()`.

7.2.2 A Skeleton Application *taskapp*

Although *taskd* is an application layer daemon, it must be logically separated from a *taskapp*. Each *taskapp* needs a *taskd* associated with it. While the latter handles instantiation, failure and disconnection detection, and re-instantiations, the former handles application level details such as processing the data sent by a remote service, reaction to failures and delays at the *taskd* layer, and buffering of Application Data Units (ADUs) for possible retransmission at a later stage. A *taskapp* process at the user node executes Algorithm 7.1 whereas that at a participating node executes Algorithm 7.2.

7.2.3 Currently Unsupported Features

Here, we describe two key features that can be important for a successful deployment of the protocol but are currently unsupported on our testbed.

Code Mobility In Algorithms 7.1 and 7.2, we assume that an application process (*taskapp*) exists on the service providing devices even before instantiation is completed by the corre-

Algorithm 7.1 UserTaskApp($FILE$)

1: `/* This taskapp executes on a user device which requests a given distributed application to be executed on the network. */`
2: $tup \leftarrow$ ReadTuples($FILE$); `/* FILE contains data-flow tuples */`
3: $TG \leftarrow$ CreateTaskGraph(tup); `/* create TG from tuples */`
4: Open a UNIX domain socket S for IPC with *taskd*;
5: Encapsulate TG and send to *taskd* over S;
6: Block on S for receiving a confirmation that instantiation of TG has been completed by *taskd*;
7: Send `TASK_DATA` to *taskd* on S; `/* Note that the physical addresses of instantiated TG nodes are hidden from the taskapp. */`

Algorithm 7.2 OtherTaskApp

1: `/* This taskapp executes on a device which provides the desired resource or service to an interested device. */`
2: Open a UNIX domain socket S for IPC with *taskd*;
3: Block on S for receiving an ADU from the local *taskd*;
4: Receive ADU (`TASK_DATA`) from *taskd* that was sent by the instantiated parent `taskapp`;
5: Process ADU and send results to *taskd*; `/* Note that the physical addresses of instantiated TG nodes are hidden from the taskapp */`
6: Perform ADU buffering and reliability steps described in Section 7.2.3, if necessary.

sponding *taskd* process. However, this is not a realistic assumption in many real systems. If a service providing host H is unaware of the exact structure of the distributed application, it is the responsibility of another participating host to inform H about the application logic that the latter is expected to execute (after being instantiated) towards performing the task. Although we did not address this issue at the time of writing of this dissertation, the desired feature can be achieved by the use of mobile code. Executable code encapsulating the *taskapp* can be sent to the relevant hosts that have been instantiated to execute the application. Upon receiving the encapsulated *taskapp* code, the local *taskd* can extract the code from the received buffer to a file which can then be executed as a separate process. This process then acts as the *taskapp*.

The scheme described above suffers from the same problems that any system using mobile code suffers from; the principal one being security. A *sandbox* environment running a virtual machine is a usual solution mechanism for mitigating security threats since that imposes necessary restrictions on the access of local resources by the mobile code. Solution approaches to this problem exist in the literature [37, 67].

Reliability of Task Execution Another issue that we have not addressed is that of reliable execution of a given distributed task after discovery of the desired resources. To understand the problem better, consider the following example of a *line* task graph $U - X - Y - Z$. X needs a service of type Y which in turn needs a service of type Z to complete a higher level task for the user. The instantiation protocols proposed in Chapter 3 first instantiate at user node u a device x_i that can provide service X, then x_i instantiates a device y_j that can provide service Y, and finally y_j instantiates a device z_k that provides service Z. After the instantiation is complete, u sends a task data unit (ADU:A_s) to x_i which then transforms it according to its service specification before sending it to y_j and so on. The *taskapp* at u may expect a response from that at x_i soon to proceed further. Under these circumstances, suppose that y_j gets disconnected from z_k before the latter is able to respond to the former. If the nature of the data flow is such that y_j's response to x_i is contingent upon z_k's response

to y_j, then the latter will not be able to respond to x_i soon. Instead, after the `HELLO` timer runs off, y_j will detect a loss of child and will attempt rediscovery of a service of type Z.

The $AvgEffT$ metric proposed in Chapter 3 captures the average number of ADUs that travel from a source of a data-flow to its intended sink (where the data-flow ends). However, in a situation described above, although the effective throughput is perfect for the one-way data flow (an ADU from u reaches a sink z_k), its semantics are not adequate from a practical standpoint. From this application's perspective, the data flow will be useful only when the user receives the output from the chain of services in the task graph. To solve this problem, end-to-end reliability semantics must be built into the protocol. To this end, schemes for buffering and retransmission of ADUs are necessary in the *taskapp* in addition to the rediscovery and re-instantiation that happen inside the *taskd*. Although we have not implemented the semantics of end-to-end *taskapp* reliability in our prototype, we believe that Algorithm 7.3, if executed at all instantiated nodes, is a useful first step.

Algorithm 7.3 ADUTRANSMIT(A_s)

1: **G**iven: ADU A with sequence number s;
2: Transmit A_s to relevant instantiated node,
3: $Q_{outstanding} \leftarrow Q_{outstanding} \cup A_s$; `/* buffer into window of unacked ADUs */`
4: $T.sched()$; `/* schedule application Timer */`
5: **if** ($Response_s$ is received before T expires) **then**
6: $Q_{outstanding} \leftarrow Q_{outstanding} \setminus A_s$; `/* remove acked ADU from window */`
7: $T.cancel()$; `/* cancel application Timer */`
8: **else**
9: Retransmit A_s;
10: $T.resched()$; `/* reschedule application Timer */`
11: **end if**

One delicate issue in Algorithm ADUTRANSMIT is the period of the timer settings. Ideally, an ADU timer should be adaptive to the round trip time estimates similar to the TCP timers. In the least, it should be proportional to the depth in the Task Graph with

respect to the corresponding data flow. For example, if y_j sets its timer for a period P, then x_i should set it to close to $2P$, and u to around $3P$.

Algorithm ADUTransmit only guarantees delivery of an ADU from the source of a data-flow to its sink. It does not guarantee ordered delivery of ADUs. Since ordered delivery of ADUs is a highly application specific issue, we do not address it in this work. However, the use of TCP guarantees ordered delivery of ADUs from a particular source to a particular sink in the absence of re-instantiations.

Chapter 8

Extensions to the Task Graph Based Modeling Approach

In the earlier chapters of this dissertation, we have described the basic building blocks for task based resource discovery in mobile failure-prone environments such as MANETs. In this chapter, we describe the service composition problem in this context and discuss how it can be modeled satisfactorily with hierarchical task graphs. This material is excerpted from [9].

Consider a ubiquitous computing application of image capture, storage and printing. A user wants to locate a digital camera nearby, get her image captured, preview it, and then store it (perhaps temporarily) on available storage devices in her surroundings, since her PDA may not possess sufficient storage capacity to store the image. The user may also want to print the image on a nearby printer after capture. Figure 8.1 depicts two task graphs for performing the same task at different levels of abstraction.

Figure 8.1(a) depicts a scenario in which a user requests and discovers a `CAMERA` service near her current location, gets her picture taken, previews the picture on her PDA screen, and then instructs the camera to store it in a distributed peer-to-peer (P2P) storage

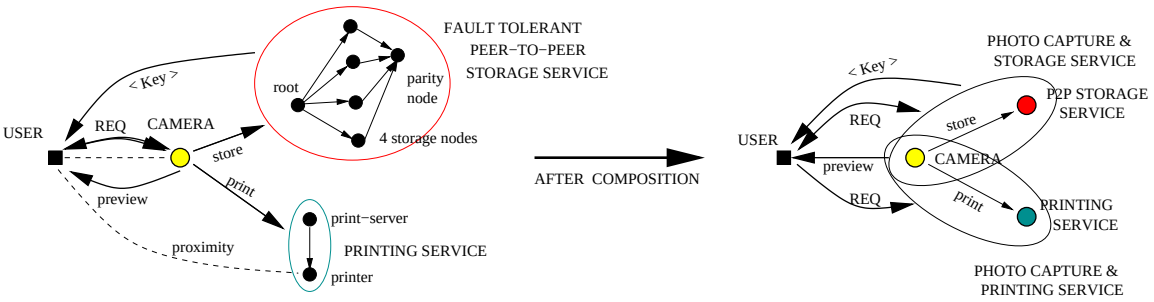


Figure 8.1: An Example of Service Composition

system in the surroundings. A fault-tolerant distributed P2P storage service is collaboratively provided by a number of devices with available storage space. Fault-tolerance is achieved by replication or coding as indicated in the figure. After the camera discovers such a storage service, it stores the image, and the latter returns a `Key` to the user for accessing the particular image in future. The printing also proceeds in a similar fashion. However, in Figure 8.1(b), the user just requests an instance of a `Photo Capture & Storage` service which is a service *composed* of two simpler services, namely, `CAMERA` and `P2P Storage`. In this case, a camera would already have chosen a nearby instance of the `P2P Storage` service in the network. Hence, when the user sends a "store" instruction, the image is stored on the already selected storage nodes with no discovery latency.

Note that the fault tolerant `P2P Storage` service is a complex service itself as it is composed of simpler storage services, hence the composition hierarchy is two levels deep. Each service is represented logically by a "node" in the task graph. Also, a task graph corresponding to a complex service such as `P2P Storage` is represented by a node in the hierarchical task graph corresponding to the higher level service. We develop the necessary notation in the next section that is used throughout this chapter.

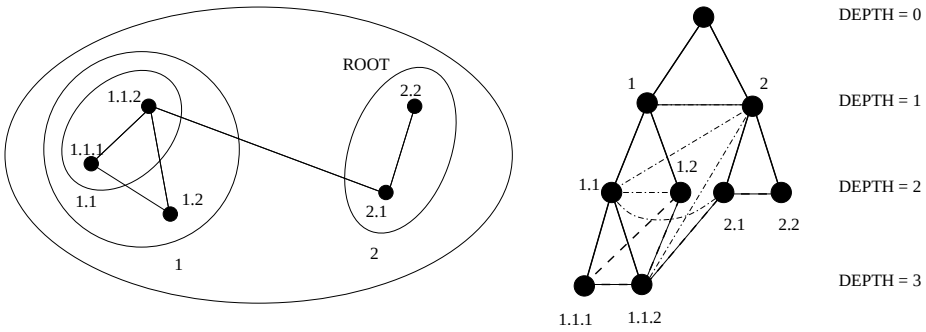


Figure 8.2: A Hierarchical Task Graph and Layered Graphs

8.1 Hierarchical Task Graphs

A hierarchical task graph contains nodes that are simple or complex. Simple nodes are the leaves of the containment hierarchy (this is intuitively clear but is defined formally in the next paragraph) and are characterized by their *type* and *attributes*. A complex node is at an intermediate level of the containment hierarchy and contains another hierarchical graph of smaller size.

Definition 8.1 The **containment hierarchy**, $\mathcal{CH}$, of a hierarchical task graph is a *tree* structure as depicted in Figure 8.2. The containment hierarchy possesses a tree structure because there are no overlaps between complex nodes at any levels in the hierarchy. In Figure 8.2, the nodes have been uniquely labeled according to their *depths* in $\mathcal{CH}$.[1] For example, nodes S_1, S_2, and $S_{1.1}$ are complex nodes, whereas $S_{1.1.1}$, $S_{1.1.2}$, $S_{1.2}$, $S_{2.1}$, and $S_{2.2}$ are simple nodes (node S_X represents service X in this chapter).

Definition 8.2 A **hierarchical task graph** G_H can be defined by a tuple $(V, E, \mathcal{CH})$, where $V = V_s \cup V_c$ is the set of nodes, and $E = E_s \cup E_c$ is the set of edges between certain pairs of nodes. V_s is the set of simple nodes which are *leaf* nodes of $\mathcal{CH}$, and V_c is the set of complex nodes which are the intermediate nodes. E_s is the set of *simple* edges between simple nodes, and E_c is the set of *complex* edges between complex nodes as well as between simple and complex nodes.

[1]The "depth" of the top node in $\mathcal{CH}$ is assumed to be 0 and it increases as the tree is traversed downward.

The definition of a hierarchical task graph that we have given here is general thus far. In Section 8.2.2, we establish rules in our instantiation framework that govern the derivation of complex edges from simple edges.

8.2 Service Composition using Hierarchical Graphs

In this section, we describe a framework for service composition (defined in Section 3.1) using hierarchical task graphs. We present more terminology necessary for a complete description of our proposed instantiation algorithms.

8.2.1 Structure of a Complex Node

A complex node in a hierarchical task graph is an abstract representation of a smaller task graph corresponding to a complex service which can be requested by another device. Consider a complex node $C_i \in V_c$ at depth $= i$ in $\mathcal{CH}$. Let $V(C_i) = \{C_{i+1}^{(1)}, C_{i+1}^{(2)}, \ldots, C_{i+1}^{|V(C_i)|}\}$ be the set of nodes at depth $= i+1$ which are C_i's children in $\mathcal{CH}$. Then $V(C_i)$ is the vertex set for the task graph corresponding to the complex node C_i.

Definition 8.3 A **service instance** $\tilde{S}$ of a service S is comprised of a set of devices in the network which can perform S. $\tilde{S}$ is formed after a successful execution of the instantiation algorithm on the hierarchical TG corresponding to S. If S is simple, then $\tilde{S}$ contains one device; otherwise, $\tilde{S}$ contains multiple devices that collaboratively perform S. A complex node is said to be instantiated if all its children nodes in $\mathcal{CH}$ have been instantiated.

Definition 8.4 A **controller** of a service instance $\tilde{S}$ is a device $\overline{S} \in \tilde{S}$ that is responsible for advertising service S and also for responding to queries for S by other devices. In the case of a simple service, $\overline{S} = \tilde{S}$. However, in the case of a complex service, the controller must be selected or *elected* from among all the participating devices.

The controller device also has a responsibility of carrying out discovery, instantiation, and re-instantiation of other services that S needs. The specific functions of the controller will become clearer in Section 8.2.2 where we describe the instantiation algorithm in detail.

Definition 8.5 Interface nodes are nodes in $V(C_i)$ that interact with services outside C_i at that particular depth in CH. In Figure 8.2, nodes $S_{2.1}$ and $S_{1.1.2}$ act as the interface nodes for services depicted by nodes S_2 and S_1. If $IF(C_i)$ denotes the set of interface nodes in C_i, then $1 \leqslant |IF(C_i)| \leqslant |V(C_i)|$.

Interface nodes are necessary during instantiation for continuing the discovery of services which are located downstream from the current node in the task graph at a particular depth in the containment hierarchy. They also serve as points of data exchange between services.

The *Type* of a Complex Node The "type" of a complex node reflects the set of attributes that all its constituent nodes possess. In our proposed framework, we favor strong typing of services and hence complex nodes. In other words, in our framework, every service belongs to a particular type. When a certain device requests a service, it refers to the latter by its unique type. There are some "atomic types" in any network of devices and some "composed types" which are constructed from the atomic types. A close analogy is that of the basic data types in a higher level programming language and the advanced data types that can be constructed from the former.

8.2.2 Instantiation of Hierarchical Task Graphs

In this subsection, we describe an algorithm proposed for instantiation of hierarchical task graphs in order to achieve service composition. As in Chapter 5, we assume the presence of a reactive MANET routing protocol such as AODV [61], DSR [46], OLSR [44] etc. and

a reliable transport protocol such as TCP.

When the system is in its initial stages, devices only respond to queries about simple services, i.e., their core capabilities. However, task graphs can be stored on these devices either by a user requesting their services or during their initialization. For example, a device hosting a print server ($\overline{PS}$) may possess a task graph during initialization that will instruct it to discover a printer device ($\overline{PRT}$). In case the task graph is not available during system initialization, it may be supplied by the user making the request for a printing service ($P = \textsc{Composition}(PS, PRT)$). There is an issue to be noted here: if $\overline{PS}$ possesses a task graph corresponding to P in the beginning, then it will discover a $\overline{PRT}$ and can answer user queries about P, since $\overline{P} = \overline{PS}$. On the contrary, if $\overline{PS}$ does not possess a task graph corresponding to P, it cannot respond to user queries about P because it is not aware of that type of service. Hence the user (or the administrator) is expected to supply the relevant task graph to $\overline{PS}$ so that it can discover an instance of PRT to be able to offer a complete instance of the printing service P.

The information known to the user must be up to a certain depth d in the containment hierarchy of G_H such that all services needed at or below d must either already have available instances in the network, or their instances must be formed in the network on-demand. These associations can be cached so that in future time instants, users can query for P and be served with low delay.

We believe that a community of users can continually construct and augment a set of services from their simpler components and the logical task graphs which represent these complex services can be shared among the user population. This will result in more users using services that have been composed earlier. At the same time the protocol framework allows more innovative users to create complex services and then share them with other users. The instantiation of a complex node in most cases will however be done at runtime; we assume that the user device possesses the task graph G_H along with the containment hierarchy $\mathcal{CH}$ so that it does not have to depend upon the existence of simpler services in

the network. However, if any instance of a required simpler service exists in the network, our protocol attempts to utilize it.

Derivation of Complex Edges from Simple Edges Let $TG_{flat} = (V_s, E_s)$ be the *flat* task graph that can be constructed from simple nodes and edges alone. In TG_{flat}, the user node is the root node and all other nodes are located "downstream" from it. As described in Chapter 5, the instantiation of TG_{flat} proceeds along the branches of a breadth first search spanning tree[2] (BFST) rooted at the user node. We use the terminology of level $L[.]$ introduced in Algorithm 4.1. If X and Y are nodes in BFST, Y is said to be downstream of X if $L[X] > L[Y]$. Algorithm 8.1 establishes the rules for creation of complex edges in G_H from simple edges given the containment hierarchy of nodes. Application of the concept of a rooted BFS tree at every "depth" in G_H helps the instantiation process. This will be more apparent in Section 8.2.3.

Algorithm 8.1 DeriveComplexEdgesFromSimpleEdges

1: **Given**: $TG_{flat} = (V_s, E_s)$; $\mathcal{CH}$; $V = V_s \cup V_c$; $e = (n_i, n_j) \in E_s$;
2: depth$(n_i) \leqslant$ depth(n_j) in $\mathcal{CH}$;
3: $BFST \leftarrow$BFS(TG_{flat}); `/* generate BFS tree with levels` $L[.]$ `*/`
4: $q \leftarrow$ Common Ancestor of n_i, n_j in $\mathcal{CH}$ with lowest value of depth;
5: **if** $(L[n_i] > L[n_j])$ **then**
6: Find $p \in V$ such that p is a child of q and an ancestor of n_i in $\mathcal{CH}$.
7: $\forall n_c : n_c = n_j$ or is an ancestor of n_j and depth$(n_c) \geqslant$ depth(p), $E_c \leftarrow E_c \cup (p, n_c)$.
8: **else**
9: Find $p \in V$ such that p is a child of q and an ancestor of n_j in $\mathcal{CH}$.
10: $\forall n_c : n_c = n_i$ or is an ancestor of n_i and depth$(n_c) \geqslant$ depth(p), $E_c \leftarrow E_c \cup (p, n_c)$.
11: **end if**

The rationale behind Algorithm 8.1 is that an instantiated upstream node can in-

[2]In a BFS spanning tree, vertices which are equidistant from the root are at the same level. It is unrelated to G_H's containment hierarchy $\mathcal{CH}$ which is also a tree.

stantiate a downstream node on-demand if an instance of the latter does not exist in the network. Hence, for every simple edge between two nodes, complex edges exist between the corresponding complex upstream node and simple as well as complex downstream nodes, assuming all these nodes are in one common subtree in $\mathcal{CH}$. These edges are necessitated by some aspects of the process of discovery or instantiation of downstream nodes by upstream nodes as illustrated by an example later in this section. In Figure 8.2, S_2 is a complex root node in the BFST, a simple edge between simple nodes $S_{1.1.2}$ and $S_{1.2}$ generates a complex edge between complex nodes $S_{1.1}$ and $S_{1.2}$. Similarly, the simple edge $(S_{2.1}, S_{1.1.2})$ generates the following complex edges: $(S_{2.1}, S_{1.1})$, $(S_2, S_{1.1})$, $(S_2, S_{1.1.2})$, and (S_2, S_1).

We now illustrate the principal idea behind our instantiation algorithm by means of a simple example. Detailed steps are explained later in Section 8.2.3. Consider the hierarchical task graph shown in Figure 8.2. Suppose node S_2 has been already instantiated completely, i.e., $\overline{S_{2.1}}$ and $\overline{S_{2.2}}$ are both known. Suppose that $\overline{S_2} = \overline{S_{2.2}}$; $\overline{S_2}$ wants to discover an instance of node S_1 and broadcasts a query packet with the relevant information. There are two possibilities: (1) an instance of node S_1 exists, and (2) there is no reachable instance of service S_1. In case (1), the controller device of S_1 (i.e. $\overline{S_1}$) receives the `S_QRY` and responds to $\overline{S_2}$ which is the same as $\overline{S_{2.2}}$ in this example. In the `C_RESP` packet, $\overline{S_1}$ sends information about interface node instances ($\overline{S_{1.1.2}}$ in this case) between nodes S_1 and S_2, so that data flow can occur between services 1 and 2 later. When $\overline{S_{2.2}}$ receives the response with interface node information, it informs $\overline{S_{2.1}}$ about $\overline{S_{1.1.2}}$. Device $\overline{S_{2.1}}$ also updates its task graph with the new instances of devices participating in service 1.

In case (2), no device in the network answers the query for node S_1. It can mean that no instance of service 1 has been composed yet, or that those instances are unreachable from the instances of node S_2. In such a situation, node S_1 needs to be instantiated on-demand. $\overline{S_{2.2}}$ then queries for the complex node $S_{1.1}$. It is indeed possible that $\widetilde{S_{1.1}}$ exists without $\widetilde{S_1}$ existing. In that case, $\overline{S_{1.1}}$ would want to discover $\overline{S_{1.2}}$ in order to have the instance $\widetilde{S_1}$. But, if $\overline{S_{1.1}}$ does not exist, $\overline{S_2}$ will query for simple node $S_{1.1.2}$. Note that this

is possible because of the rules in Algorithm 8.1. An instance $\widetilde{S_{1.1.2}} = \overline{S_{1.1.2}}$ that replies to the query will be instantiated, and then it will continue the search process further. After all requested services have been instantiated, the application can start data transmission. After data transmission is over, the associations between participating nodes can be cached for future use so that a request for a complex service can be fulfilled immediately, thus achieving low service delays.

8.2.3 Distributed Algorithms for Hierarchical Task Graph Instantiation

Here we describe the steps of the distributed algorithm for achieving service instantiation and composition. All devices execute a copy of the distributed algorithm except the core user device which acts as a coordinator for state synchronization. The user wants to execute an application and is assumed to possess a relevant task graph for it. Earlier in this section, we presented examples of situations in which high-level knowledge of complex nodes and the interactions between them is not enough. Hence, in order to facilitate instantiation under these circumstances, a user's PDA must contain information about G_H at all depths in $\mathcal{CH}$. After some instances of a complex service have been created in the network, subsequent users of the system only require high-level (or low depth) information about G_H to discover these services.

Algorithm 8.2 depicts the execution of the instantiation algorithm at a coordinator device, whereas Algorithm 8.3 is executed on all other devices in a MANET. As in Chapter 5, these algorithms too have been presented in pseudo-code form with descriptive names for variables and useful routines. All routines have not been explained in complete detail here but the general idea is easy to grasp. Both algorithms are distributed and are driven by events such as arrival of packets of certain types or by expiry of certain timers. The steps within each single execution of the `for()` loops are assumed to be atomic; in other words, if a packet arrives during the execution of an "if–then–else" statement, it will be queued and processed only after the control returns to the WaitForPacket() call.

Querying for Services The instantiation process begins at the user device (coordinator, U) which contains a complete description of the hierarchical task graph, G_H corresponding to the application that the user desires to execute. In the beginning, a task graph TG is extracted from G_H at depth $d = 1$, i.e., only the top level node descriptions are known to the algorithm at this stage. Then the nodes in TG which are neighbors of U are queried (Alg.-8.2::9–13). The `S_QRY` packet contains a query string ($Q_{str} = S_{k_1} :: S_{k_2} :: \ldots :: S_{k_n}$) which constitutes of the aggregate type of the requested service.[3] In Q_{str}, S_{k_1} is the simplest service and S_{k_n} is the most complex service.

The coordinator device starts a timer after broadcasting a query. If the timer expires before it gets any response, it queries for a less complex node in G_H, an instance of which may be available in the network (HierTGHandleBroadcastTimeouts() at Alg.-8.2::23); detailed steps can be found in function 8.12. The complex edges that were included in G_H by Algorithm 8.1 are useful at this stage since the coordinator knows exactly what node to query for if it does not hear a response from an instance of a complex node.

As in the case of instantiation of flat TGs, we assume that all steps to control broadcast storms are taken; some of these salient steps include the use of expanding ring search and the maintenance of broadcast sequence numbers to avoid the rebroadcast of an old `S_QRY` packet.

State Management in Devices and Response to Broadcast Queries We depict a generic state in the non-coordinator algorithm by variable $S[m]$ (Alg.-8.3::1) where m is the current level of complexity of the service. At the time of system initialization, no instances of complex nodes exist in the network; hence, each device is a member of an atomic service instance. At that time instance, $m = 0$ and $S[m] = S_{atom}$. With progression of time, devices start taking part in multiple services at various levels of complexity. This is not to be confused with the value of "depth" of the corresponding node in $\mathcal{CH}$. A "higher-level

[3] In general, the entire service hierarchy is included in the query string starting from the depth of the querying device.

service" always refers to one at a lower depth in $\mathcal{CH}$.

At any level of complexity $i \in [0, m]$, a device's state is given by $state[i]$ which has two disjoint components: whether it has been instantiated at that level (denoted by one of the 3 states, `UNINST`, `WAIT_FOR_ACK`, or `INST`) or if it is a controller of that service or just a plain member (denoted by `SRV_CONTROLLER` and `SRV_MEMBER`, respectively).

When a device receives a service query packet (Alg.-8.3::4) with a certain query string, it tries to find a match with the most complex service that it can offer at that instant of time.[4] If the device is a controller of the service at that level and it is uninstantiated, then it accepts the request and sends a candidate response to the sender. It also changes its state from `UNINST` to `WAIT_FOR_ACK` and starts a timer (not shown). If the timer expires before receiving an `ACK` from the sender, the state is changed back to `UNINST`.

Instantiation of Services and Confirmation A coordinator can receive responses from several candidate service instances out of which only one is sent an `ACK` (function 8.4). However, the instantiation of that service is completed only when the controller of that service replies to the `ACK` packet with a `CONFIRM` packet (function 8.5). We explain these above steps below in greater detail. The coordinator sends the hierarchical task graph G_H encapsulated in the `ACK` packet so that the chosen candidate can continue the process of discovery of children nodes and their instantiation. Hence, at any stage, the responsibility of instantiation of hitherto uninstantiated services is distributed among the controllers of the nodes in G_H that have already been instantiated.

After a candidate controller device receives an `ACK`, it changes its state to `INST`. Imagine a situation when a coordinator queries for a complex service CS, then times out because there are no instances of CS, and then queries for a simpler constituent of CS, namely SS. Suppose, a candidate device D responds to the SS query but by the time D

[4]Function MaxMatch whose detailed listing has been omitted tries to find the most complex entry in the query string that matches $S[m]$. It returns -1 if no match is found.

Algorithm 8.2 HierTGCoordinatorDeviceInstantiation

1: Given: Hierarchical Task Graph, $G_H = (V, E, \mathcal{CH})$.
2: $d \leftarrow 1$; /* depth in $\mathcal{CH}$ */
3: $TG(V_T, E_T) \leftarrow$ ExtractTG(G_H, d); /* extract task graph at depth=1 */
4: $V_{ngb} \leftarrow \{v \in V_T \mid v$ is user node U's neighbor in $TG\}$;
5: $UninstSrv \leftarrow V_T$;
6: $UninstSrv_{ngb} \leftarrow V_{ngb}$; /* uninstantiated services:(all+neighboring) */
7: $InstSrv \leftarrow \phi$;
8: $InstSrv_{ngb} \leftarrow \phi$; /* instantiated services:(all+neighboring) */
9: **for** $(\forall v \in UninstSrv_{ngb})$ **do**
10: $v.acked \leftarrow$ FALSE;
11: $v.Q_{str} \leftarrow v.service\_type$; /* initialize *acked* variable + Query string */
12: BroadcastQuery$(v.Q_{str})$; /* query for required services */
13: **end for**
14: **for** (; ;) **do**
15: WaitForPacket(pkt); /* wait for packets to arrive */
16: **if** $(pkt.type ==$ C_RESP$)$ **then**
17: HierTGHandleCResp(pkt); /* handle response from a candidate */
18: **else if** $(pkt.type ==$ CONFIRM$)$ **then**
19: HierTGHandleConfirm(pkt); /* handle confirm from a candidate */
20: **else if** $(pkt.type ==$ SUBTREE_CONF $\wedge\ (\exists s \in InstSrv_{ngb} : \overline{s} == pkt.srcaddr))$ **then**
21: HierTGHandleSubtreeConfirm(pkt, s); /* handle subtree confirm */
22: **end if**
23: HierTGHandleBroadcastTimeouts(); /* function 8.12 */
24: **end for**

Algorithm 8.3 HierTGNonCoordinatorDeviceInstantiation

1: **Given:** Own service types and instantiation states at different depths in $\mathcal{CH}$:

$S[m] = \{i_0,\ i_0.i_1,\ i_0.i_1.i_2,\ \cdots,\ i_0.i_1.\ldots.i_m\}$ /* $m = 0 \Rightarrow S = \{S_{atom}\}$ */

$A =$ {UNINST, WAIT_FOR_ACK, INST} ;

$B =$ {SRV_MEMBER, SRV_CONTROLLER} ;

$\forall i \in [0, m] : state[i] \in A \times B$

2: **for** (; ;) **do**

3: WaitForPacket(pkt); /* can forward broadcast pkts here */

4: **if** (($pkt.type ==$ S_QRY) $\wedge$ (MaxMatch($pkt.Q_{str}, S[m]$) $\neq -1$)) **then**

5: HierTGHandleSearchQuery(pkt);

6: **else if** (($pkt.type ==$ ACK) $\wedge$

($\exists i : S[i] == pkt.service\_type$) $\wedge$ ($state[i]$ & WAIT_FOR_ACK)) **then**

7: HierTGHandleAck(pkt);

8: **else if** ($pkt.type ==$ C_RESP) **then**

9: HierTGHandleCResp(pkt);

10: **else if** ($pkt.type ==$ CONFIRM $\wedge$ ($s \leftarrow pkt.service\_type$)) **then**

11: HierTGHandleConfirm2(pkt);

12: **else if** ($pkt.type ==$ SUBTREE_CONF $\wedge$

($\exists s \in InstSrv_{ngb} : \overline{s} == pkt.srcaddr$)) **then**

13: HierTGHandleSubtreeConfirm2(pkt);

14: **else if** ($pkt.type ==$ CONTINUE_SEARCH) **then**

15: ContinueDownstreamSearch(pkt);

16: **end if**

17: HierTGHandleBroadcastTimeouts(); /* function 8.12 */

18: **end for**

function 8.4 HIERTGHANDLECRESP(PACKET *pkt*)

1: **if** $((\exists w \in UninstSrv_{ngb} \,|\, w.service\_type == pkt.service\_type;) \wedge (w.acked == \texttt{FALSE}))$ **then**

2: SENDACK($pkt.srcaddr, G_H$); `/* send ACK along with the task graph */`

3: $w.acked \leftarrow \texttt{TRUE}$;

4: **end if**

function 8.5 HIERTGHANDLECONFIRM(PACKET *pkt*)

1: `/* the controller of a service is confirming an ACK */`

2: $g \leftarrow$ EXTRACT_SUBGRAPH(pkt);

3: $w \leftarrow pkt.service\_type$; $\overline{w} \leftarrow pkt.srcaddr$;

4: INSTANTIATE(G_H, g); `/* instantiate nodes in` G_H `with addresses in` g `*/`

5: $UninstSrv_{ngb} \leftarrow UninstSrv_{ngb} \setminus \{w\}$;

6: $InstSrv_{ngb} \leftarrow InstSrv_{ngb} \cup \{w\}$; `/* update service vars (neighbor) */`

7: $UninstSrv \leftarrow UninstSrv \setminus \{w\}$;

8: $InstSrv \leftarrow InstSrv \cup \{w\}$; `/* update service vars (all) */`

function 8.6 HIERTGHANDLESUBTREECONFIRM(PACKET *pkt*; HTG_NODE s)

1: $g \leftarrow$ EXTRACT_SUBGRAPH(pkt);

2: INSTANTIATE(G_H, g); `/* instantiate nodes in` G_H `with addresses in` g `*/`

3: $SubtreeSrv \leftarrow \{s' |\, s'$ is in the subtree at depth $d = 1$ rooted at $s\}$;

4: $UninstSrv \leftarrow UninstSrv \setminus SubtreeSrv$;

5: $InstSrv \leftarrow InstSrv \cup SubtreeSrv$; `/* update service vars (all) */`

6: **if** $(UninstSrv == \phi)$ **then**

7: Send Interface Node information regarding *non-tree edges* to service controllers.

8: Instantiation is complete.

9: Start application and begin sending `TASK_DATA`.

10: Start exchanging periodic `HELLO` messages with controllers of top level services.

11: **end if**

function 8.7 HierTGHandleSearchQuery(Packet pkt)

1: **if** ($state[m]$ & (`SRV_CONTROLLER` | `UNINST`)) **then**
2: SendCResp($pkt.srcaddr, S[m]$); `/* send response to upstream controller */`
3: $state[m] \leftarrow state[m]$ & ~`UNINST` | `WAIT_FOR_ACK`; `/* change state */`
4: **end if**

function 8.8 HierTGHandleAck(Packet pkt)

1: $G_H \leftarrow$ Extract_hierTG(pkt); `/* extract rest of TG */`
2: $state[i \ldots m] \leftarrow state[i \ldots m]$ & ~`WAIT_FOR_ACK` | `INST`; `/* update to INST */`
3: $C_{out} \leftarrow$ ExtractInterfaceNodeInfo($G_H, S[m]$); `/* interface nodes */`
4: SendConfirm($pkt.srcaddr, S[m]$); `/* confirm service instantiation */`
5: **if** (($c \leftarrow$ FindChildren($G_H, S[m]$)) $\neq \phi$) **then**
6: `/* send CONTINUE_SEARCH pkt to interface nodes */`
7: $\forall x \in C_{out}$: InstructNodeToContinueSearch($x, c(x), G_H$);
8: **else**
9: SendSubtreeConfirm($pkt.srcaddr, S[m]$); `/* send to parent controller */`
10: **end if**

function 8.9 HierTGHandleConfirm2(Packet pkt)

1: **if** ($\overline{S[m]} \neq MyAddr$) **then**
2: InformController(pkt); **continue**;
3: **end if**
4: $g \leftarrow$ Extract_Subgraph(pkt);
5: $w \leftarrow pkt.service\_type$; $\overline{w} \leftarrow pkt.srcaddr$;
6: Instantiate(G_H, g); `/* instantiate nodes in` G_H `with addresses in` g `*/`
7: $UninstSrv_{ngb} \leftarrow UninstSrv_{ngb} \setminus \{w\}$;
8: $InstSrv_{ngb} \leftarrow InstSrv_{ngb} \cup \{w\}$; `/* update service vars (neighbor) */`
9: $UninstSrv \leftarrow UninstSrv \setminus \{w\}$;
10: $InstSrv \leftarrow InstSrv \cup \{w\}$; `/* update service vars (all) */`

function 8.10 HIERTGHANDLESUBTREECONFIRM2(PACKET pkt)

1: **if** ($\overline{S[m]} \neq MyAddr$) **then**
2: INFORMCONTROLLER(pkt); **return**;
3: **end if**
4: $g \leftarrow$ EXTRACT_SUBGRAPH(pkt);
5: INSTANTIATE(G_H, g); `/* instantiate nodes in` G_H `with addresses in` g `*/`
6: $SubtreeSrv \leftarrow \{s' |\ s'$ is in the subtree rooted at s in $G_H\}$;
7: $InstSrv \leftarrow InstSrv \cup SubtreeSrv$;
8: $UninstSrv \leftarrow UninstSrv \setminus SubtreeSrv$; `/* update service vars (all) */`
9: **if** ($UninstSrv \neq \phi$) **then**
10: **return**; `/* all children have not confirmed yet */`
11: **end if**
12: **if** ($S[m] ==$ FINDROOTNODE(G_H, m)) **then**
13: Send Interface Node information regarding *non-tree edges* to controllers.
14: $S[m+1] \leftarrow$ CONSTRUCTSRVTYPE($InstSrv$); `/* construct a complex node */`
15: $state[m+1] \leftarrow$ `SRV_CONTROLLER | INST`;
16: **for all** ($s \in InstSrv; C_i \leftarrow \overline{s}$) **do**
17: INFORMNODE(C_i, `SRV_MEMBER`); `/* inform controllers to change state */`
18: **end for**
19: **if** (I had received a query for $S[m+1]$ in the past) **then**
20: $upstream \leftarrow$ Controller of the instance of parent node;
21: SENDCRESP($upstream, S[m]$); `/* send response to upstream controller */`
22: $state[m] \leftarrow$ `SRV_CONTROLLER | WAIT_FOR_ACK`;
23: **end if**
24: $m \leftarrow m+1$;
25: **else**
26: SENDSUBTREECONFIRM($\overline{S[m].parent}, S[m]$); `/* send to parent controller */`
27: **end if**

function 8.11 ContinueDownstreamSearch(Packet pkt)

1: $G_H \leftarrow$ Extract_hierTG(pkt);
2: $TG(V_T, E_T) \leftarrow$ ExtractTG($G_H, S[m]$); `/* extract TG with me at root*/`
3: $V_{ngb} \leftarrow \{v \in V_T \mid v$ is my neighbor in $TG\}$;
4: $UninstSrv \leftarrow V_T$;
5: $UninstSrv_{ngb} \leftarrow V_{ngb}$;
6: **for all** $v \in UninstSrv_{ngb}$ **do**
7: $v.acked \leftarrow$ `FALSE`;
8: $v.Q_{str} \leftarrow v.service\_type$;
9: BroadcastQuery($v.Q_{str}$); `/* continue search */`
10: **end for**

function 8.12 HierTGHandleBroadcastTimeouts()

1: $S[m] = \{i_0,\ i_0.i_1,\ i_0.i_1.i_2,\ \cdots,\ i_0.i_1.\ldots.i_m\}$ `/*` $m = 0 \Rightarrow S = \{S_{atom}\}$ `*/`
2: `/* check for all active, uninstantiated services */`
3: **for all** ($u \in UninstSrv_{ngb} : u.active ==$ `TRUE`) **do**
4: **if** (TimeOut($u.service\_type$) == `TRUE`) **then**
5: $V' \leftarrow \{v' \in V \mid v'$ is u's child in $\mathcal{CH}$ and $(S[m], v') \in E\}$; `/* if no instance found for` u `in a specified timeout period, find instances of simpler nodes in` G_H `*/`
6: $UninstSrv_{ngb} \leftarrow UninstSrv_{ngb} \cup V'$; `/* update service vars (neighbor) */`
7: $u.active \leftarrow$ `FALSE`;
8: **for all** ($x \in V'$) **do**
9: $x.Q_{str} \leftarrow u.Q_{str} :: x.service\_type$;
10: BroadcastQuery($x.Q_{str}$); `/* broadcast queries for simpler nodes */`
11: **end for**
12: **end if**
13: **end for**

gets an ACK from the coordinator, it has independently formed an instance of CS which is available for use. In such a scenario, when the ACK for SS ($S[i]$ in line Alg.-8.3::6) arrives at D, it replies with a confirmation message for CS ($S[m], m \geqslant i$ in function 8.8::4) since the requester had originally requested an instance of CS instead. Also, D extracts the interface node information from G_H as well as information about its children nodes in the embedded BFS tree of G_H rooted at node $S[m]$. This information is important since interface nodes interact with the controllers of downstream nodes in order to continue the discovery (function 8.8::5–10).

When a CONFIRM packet arrives at the coordinator, the latter concludes that the sender of the packet, S is acting as a controller on behalf of the devices that are together offering a desired service. The arrival of this packet also means that S assumes the responsibility of discovering downstream services. The coordinator meanwhile instantiates S in it own copy of the task graph.

Subtree Confirmations Data transmission can begin only when all required services have been instantiated. Since the instantiation process is distributed and local, the coordinator needs to be informed when instantiation is complete. We achieve this by means of SUBTREE_CONF packets as in the flat TG case (see Chapter 5. When all downstream nodes rooted at a particular node, R in the task graph have been instantiated, R propagates a SUBTREE_CONF packet upstream all the way up to the coordinator. The coordinator extracts the task graph from the packet and instantiates the downstream nodes indicated in the task graph in its own copy. If all services in its list have been instantiated then the flow of application data can begin (function 8.6::8–9). Also, periodic exchange of HELLO messages begins between the coordinator and the controllers of services in G_H at the top level. This is for monitoring failures and disconnections between service instances. We explain this in further detail in Section 8.3.

The process that we described above occurs at all levels of the containment hier-

archy and inside every complex node for the completion of instantiation inside them. The controller of the root node of TG at a certain depth in $\mathcal{CH}$ also acts as the controller of the corresponding complex service, e.g., the user node acts as the controller for the entire application.

Composition and Instantiation of Complex Downstream Nodes As we have mentioned earlier in this section, the coordinator device is deemed responsible for instantiating the devices that are its neighbors at the top level of G_H. Instantiation of downstream services needs to be performed by instances of nodes interfacing with the downstream services. Addresses of such devices within an instantiated service $S[m]$ can be extracted by the service controller from the task graph (function 8.8::3), and those devices can be instructed by the controller to continue the search downstream. When an instance of an interface node receives a `CONTINUE_SEARCH` packet, it executes steps shown in function 8.11.

There are some subtle differences between the steps for instantiating downstream nodes at coordinators and at other nodes. We illustrate these differences with an example. Suppose that a service S with a corresponding task graph TG_S (this can itself be hierarchical) has been instantiated; in other words, $\tilde{S}$ and $\overline{S}$ exist. Suppose S can form a more complex service if it uses another service Z which is complex as well. If Z was instantiated earlier independently, $\tilde{Z}$ and $\overline{Z}$ exist too. Now, from a higher level task graph description G_H (involving smaller nodes TG_S and TG_Z), $\overline{S}$ knows its outgoing interface between $\tilde{S}$ and $\tilde{Z}$.[5] Let us denote this interface device by S_{if}. $\overline{S}$ instructs S_{if} to search for Z. When S_{if} broadcasts a query, $\overline{Z}$ responds, and then upon getting an `ACK`, sends a `CONFIRM` to S_{if}. The latter then relays the `CONFIRM` to $\overline{S}$ (function 8.10::1-3). The broadcast timeout rules apply to S_{if} as they apply to the coordinator device.

If a controller C of a service at level m finds that it is at the BFS-root of a complex node in G_H (at depth d in $\mathcal{CH}$) and that it has received `SUBTREE_CONF` messages from all

[5]This information is extracted from G_H by ExtractInterfaceNodeInfo() shown in function 8.8.

controllers C_i at level m, it concludes that a new service $S[m+1]$ can be *composed* and *instantiated* at depth $d-1$ from all these existing service instances.[6] This is illustrated in function 8.10::12–27. Function ConstructSrvType() constructs a name for the complex service using a standardized convention known to all devices in the network. After this new service instance is created, C assumes the role of the controller of this service. In other words, $\overline{S[m+1]} = C$. C also informs all controllers C_i (available from the variable $InstSrv$) that they should change their state to `SRV_MEMBER` with respect to $S[m+1]$. Also, since the instantiation occurred only along the edges of the BFS tree of the appropriate subgraph of G_H, if the latter has non-tree edges, the affected interface node instances are informed about each other's addresses so that they can communicate. Now, if C had been queried and all the above steps happened on-demand, then after $S[m+1]$ is formed, C responds to the upstream interface node which had initiated broadcast for $S[m+1]$ (function 8.10::19–23). Otherwise, C does nothing and waits to be queried and utilized by other users in the network.

We note that although the descriptions of the algorithms proposed in this section are slightly complicated due to the specific details in the representation of task graphs and services, the basic idea behind hierarchical service composition and instantiation is simple and very useful.

8.3 Recovering from Disconnections caused by Mobility

In this section, we present in detail how the algorithms presented in Section 8.2 react to the mobility of devices after the required services have been instantiated. In a MANET, if relative mobility of devices is very low and if the existing network topology does not change due to mobility, the application may not be disrupted at all. However, on most occasions,

[6]Note that in this context *depth* is an absolute value in $\mathcal{CH}$ which increases with decrease in complexity of services; on the contrary, *level* is a relative value which increases with increase in the complexity of a composed service.

existing routes can fail due to device mobility that results in a change in network topology, and that can cause a temporary disruption in the application. The underlying MANET routing protocol then attempts to rediscover alternate routes to the same destination. If the rediscovery happens with low delay, the application does not perceive the disruption and continues to progress smoothly. However, if the route discovery latency is large or if a network partition occurs because of device movement or failure, the application is disrupted for a longer time period.

Because of these reasons, in MANETs, it is not sufficient to discover specific instances of services and appoint them permanently to execute the application; accessibility of those devices needs to be continually monitored in order to detect disconnections, and replacement services must be discovered, if possible, for resuming the application. Since this can be a regularly occurring phenomenon in real MANETs, it is extremely important to augment the algorithms proposed in Section 8.2 such that the applications can recover from such disruptions as rapidly as possible. In this section, we present steps for performing recovery from such situations.

8.3.1 Detection of Disruptions in Service

Mobility of devices may cause network partitions or disconnections, and instantiated devices executing a service may no longer be able to communicate if all paths between them are broken. The first essential step for recovery from a disruption of service is its detection. We extend the `HELLO` messaging protocol proposed in Section 5.3 for detecting disconnections in a flat instantiated task graph to handle the hierarchical case. The protocol requires the controller of a service to send periodic `HELLO` messages to all other controllers that are cooperatively offering the service at the same depth in the containment hierarchy. These controllers reply with a `HELLO_ACK` message within a pre-determined period of time T. This is illustrated in Figure 8.3.

A "hierarchically clustered" disconnection detection scheme is favorable in this con-

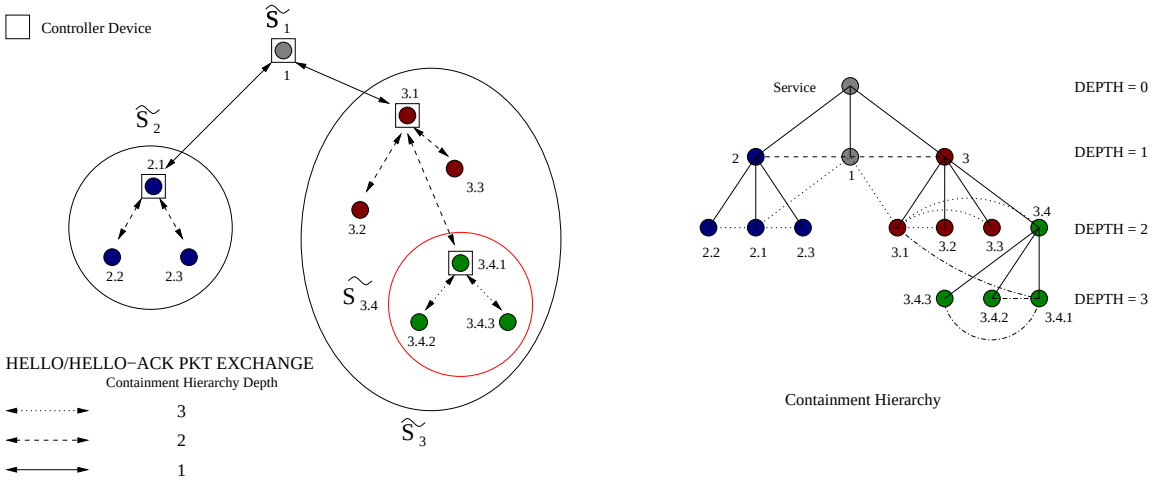


Figure 8.3: Disruption Detection in a Hierarchical Service Instance

text instead of a purely distributed one since most of the state of a particular service at a certain depth in $\mathcal{CH}$ lies with the controller of that service. The `HELLO` packet exchange is limited to between the controllers of nodes that belong to a larger complex node. In Figure 8.3, device $\overline{S_1}$ exchanges `HELLO` and `HELLO_ACK` packets only with devices $\overline{S_{2.1}}$ and $\overline{S_{3.1}}$, since they are the controllers of service instances $\widetilde{S_1}$, $\widetilde{S_2}$ and $\widetilde{S_3}$, respectively, and they are represented by nodes that belong to the same depth in the containment hierarchy of the task graph. Note that device $\overline{S_{2.1}}$, as the controller device of $\widetilde{S_2}$, also needs to exchange `HELLO` and `HELLO_ACK` packets with the other controllers, namely, $\overline{S_{2.2}}$ and $\overline{S_{2.3}}$ that form $\widetilde{S_2}$ together.

The `HELLO_ACK` packet is sent by the controller of a complex service only when all the constituents of the service have replied with their `HELLO_ACK` packets within a proper time-out period. Thus, in Figure 8.3, $\overline{S_{2.1}}$, as the controller of $\widetilde{S_2}$, would only send a `HELLO_ACK` back to $\overline{S_1}$ when it has received `HELLO_ACK` from $\overline{S_{2.2}}$ and $\overline{S_{2.3}}$ within proper time-out periods.

The above `HELLO` messaging approach is similar to the one proposed by us elsewhere [50]. This reduces the number of devices that must be tracked simultaneously by any single device but increases the time required to obtain all the `HELLO_ACK` packets from

devices that are further down in the containment hierarchy. To deal with this problem, we propose a time-out value that is proportional to the number of levels left to reach the highest depth in the $\mathcal{CH}$ of an instantiated task graph. Therefore, if d is the current depth and N is the highest value of depth in $\mathcal{CH}$ and T is the time-out value at the highest depth, then the time-out at depth d is given by: $T(d) = (N - d)T$. This approach results in a more frequent packet exchange rate between devices offering simple services than the packet exchange rate between controllers of complex services.

If a device requesting a particular service knows the maximum depth in $\mathcal{CH}$ (e.g., it is aware of the hierarchical TG representation of the service composed by simple nodes), a suitable initial timeout value can be determined during instantiation. However, if the maximum containment hierarchy level is unknown, design criteria or user input should determine how long to wait before concluding that there are no service instances available in the network.

8.3.2 The Recovery Process

The recovery process is always initiated by a controller after it detects a disconnection. If the controller of a complex node X does not receive a `HELLO_ACK` from the controller of one of its constituent nodes Z, the former deems the latter to be unreachable. In such a situation, $\overline{X}$ also attempts to re-instantiate Z.

In Figure 8.3, if device $\overline{S_{3.4.2}}$ does not send a `HELLO_ACK` to $\overline{S_{3.4.1}}$ within a proper timeout period, the latter will not send a `HELLO_ACK` back to $\overline{S_{3.1}}$, which in its turn will not send back a `HELLO_ACK` to $\overline{S_1}$. Because the timeout period involving $\overline{S_{3.4.1}}$ and $\overline{S_{3.4.2}}$ is smaller, the former is likely to re-instantiate $S_{3.4.2}$ before $\overline{S_{3.1}}$ times out for $\overline{S_{3.4.1}}$. If a replacement instance of $S_{3.4.2}$ is successfully found before a timeout, $\overline{S_{3.1}}$ will receive a `HELLO_ACK` within its timeout period, and will not be even aware that a recovery occurred at a downstream controller. An exception to this is when a lost service acts as an interface node in the task graph; in that case, the address of the new instance must be notified to an

upstream controller.

The common fate-sharing characteristic of the components of a complex service in the detection and recovery processes may arguably not yield the optimal performance in terms of latency. However, allowing a partially instantiated complex service to respond as if it were fully instantiated means that application data can be delivered to that instance. This introduces problems of where to buffer the data intended for the missing component. Since devices in a MANET are prone to experience unreachability, the best device to buffer application data is their originator. This, in conjunction with the fact that the internal details of a complex service should be hidden from the device requesting that service prompt us to treat an instance of a complex service as a whole.

We emphasized data delivery in our design. Of course, if greater importance is attached to the timely arrival of any available data, a desirable outcome can be achieved by changing the `HELLO_ACK` response policy; for example, a device could respond with a `HELLO_ACK` as long as there are at least k instantiated components in the complex service available.

8.4 Discussion

In this chapter, we investigated a related dimension of the service discovery problem, namely, service composition. In general terms, service composition refers to the process of combination of multiple simple services in order to form a larger, more complex distributed service. This offers users a great degree of transparency in discovery and selection of required services, instead of having to be cognizant of all the details about the simpler services that constitute the complex ones. Also, this can reduce discovery latency at the time of executing distributed applications. Owing to the hierarchical nature of the service composition framework, we represented a distributed application and its components using logical hierarchical task graphs. At every level of hierarchy in the graph, nodes representing logical services and

edges between nodes representing required data-flows between the corresponding services form a task graph with a specific degree of detail.

The distributed framework proposed in this chapter attempts to construct complex service instances from those of simpler services on-demand from a logical hierarchical description of the application. After a service is composed on-demand and used, its components retain their associations for a certain interval of time. If another user requests the service after the first one has finished using it but before it is scheduled to disintegrate, she does not have to compose it again on-demand. This approach can lead to better resource utilization, lower resource and service discovery latency, and can offer users a powerful abstraction of being able to query and use complex services even if they are not aware of their individual components.

The topic of scalable reuse of composed instances of distributed services in not addressed in this dissertation. In the current architecture, a user queries the network with a hierarchical task graph and the controller of a complex instantiated service responds if it matches the requirements of the query. One can envision mechanisms of propagating information about such controllers (or service access points) of instantiated services throughout the network for achieving greater scalability. Proposals such as INS [1] can be leveraged in such endeavors.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

The recent proliferation of computing devices (both tethered and untethered) that are capable of wireless communication enables a new class of distributed mobile applications. To fully exploit the diversity of services and the ubiquity of devices, approaches are required that map applications to mobile nodes. A first step toward the execution of such a collaborative application is the discovery of resources that are needed by it at runtime. In this dissertation, we propose a framework that specifies the needs of a distributed task/application in terms of a *resource dependency graph* or *task graph*. Nodes in this graph logically represent services or resources that the application requires and the edges represent the data-flows that must exist between the resource nodes to facilitate appropriate task execution. We propose algorithms and protocols for the discovery of suitable devices in the network for performing the task cooperatively while obeying the constraints imposed by the logical task graph structure. Our task execution framework includes mechanisms that adapt to mobility of devices in the MANET and the link dropouts that this creates. These mechanisms endeavor to detect partitions critical to the ongoing task and then attempt recovery.

A task graph is a key vehicle for the abstract representation of the needs of a distributed application. Specifically, it captures the various logical data-flow relationships between different components of the application that can be executed on a suitable set of resources in the network which in turn match the criteria specified in the task graph specification. In essence this allows a decoupling of the binding between the types of resources required by the application and the physical addresses of the devices offering the resources. This proves to be especially useful in mobile environments like MANETs. When a participating device fails or moves out of range during the execution of a task, another device that is aware of the corresponding logical task graph structure is able to rediscover a replacement device that offers the same type of service as the failed device. Hence, only the node to physical device binding changes in the process and the application can continue execution after the temporary glitch.

We formulate the problem of embedding a task graph onto any network (static snapshot of a MANET) as a combinatorial optimization problem. We propose several metrics that can be optimized in this framework, the most important one being average dilation: the average stretch that a task graph edge suffers when embedded on the network. Average dilation directly affects other metrics such as average throughput and ADU delay. We show that the optimization problem with respect to the average dilation metric is NP-complete even when the nodes in the task graph possess distinct attributes. However, we also prove that for tree TGs with nodes of distinct types, exact optimal polynomial time algorithms indeed exist - we proposed one such dynamic programming based algorithm, namely, TreeEmbed. TreeEmbed can be used for non-tree TGs as part of a heuristic solution.

Because TreeEmbed searches the entire network for optimal devices, we propose a greedy algorithm GreedyEmbed which is faster (as it searches only in the local neighborhood), albeit yielding suboptimal results. One major advantage of GreedyEmbed is that it is easily amenable to distributed implementations. Hence the distributed version

of GreedyEmbed became the algorithm of our choice. We propose a distributed protocol that implements the above on a MANET. In addition, we propose distributed bookkeeping algorithms including TG-patching for recovery from disconnections. The proposed algorithms can recover reasonably quickly from single persistent disconnects as well as certain classes of burst disconnects. In the worst case of affecting burst disconnects, the recovery process is much slower.

Via large scale simulations we demonstrate the feasibility of task embedding on reasonably large MANETs involving up to 100 devices that are moving in rectangular grids of several sizes according to the random waypoint mobility model. We found that our algorithms were able to yield embeddings with efficient average dilation most of the time and were able to yield high effective throughput at low mobilities. Effective throughput drops gracefully as maximum speed is increased. This is mainly because of the success of our TG-patching techniques. We also found that spatial node density played an important role in the execution of a task since it is the most significant factor affecting the connectivity of the network. Constant mobility in sparse networks yielded better performance than intermittent mobility as it often healed network partitions that were created due to low spatial node density.

Finally, we demonstrate the viability of our approach by the implementation of a real proof-of-concept prototype using off-the-shelf hardware and public domain software. Large scale mobility was emulated using packet filtering at the kernel level. The distributed protocol code can execute on machines with computing power ranging from powerful workstations to handheld devices.

9.2 Future Work

Since this was the first initiative towards modeling distributed applications on MANETs using logical task graphs, much work remains to be done. In this section we enumerate a

few directions for future research in this area.

9.2.1 Effect of Proactive Routing Protocols

Proactive MANET routing protocols usually outperform reactive protocols in handling mobility under heavy traffic loads [45]. We conducted simulations for 63-node task graphs by replacing DSR with OLSR as the underlying routing protocol and performance degradation was observed in general. This is due to congestion caused by frequent broadcasts performed by OLSR as well as our instantiation protocol. One idea that can be explored further is the coupling of resource discovery with the proactive routing protocol. Essentially the proposal is to make the routing layer task-aware. When a device floods the link state (LS) updates to the network, it can tag the values of its service attributes along with the LS packets. Then other devices upon receiving LS updates will become aware of the service attributes of remote devices in the MANET. This can save the `S_QRY` step in the current protocol and a device wishing to instantiate its children nodes in TG can directly send `ACK` packets to certain devices according to desired optimization criteria.

The trade-off is between the bandwidth savings due to `S_QRY` broadcasts and the increased flooding traffic introduced by proactive routing. In the presence of other cross traffic, this scheme may yield better delay and effective throughput. However it may consume more energy because of the periodic LS broadcasts. Applications in battlefields can benefit from the this approach whereas other less demanding applications are likely to benefit from the current reactive approach.

9.2.2 Scalability Issues

In Chapter 8 we discussed how service composition can improve the scalability of the system by allowing the reuse of already instantiated distributed services. The current proposal advocates the use of broadcast queries for discovering such service instances. For the system

to scale more effectively, concepts akin to name resolution in INS [1] can be applied. The access node of the distributed service needs to advertise the composite service to certain name resolvers present in the network. This information will propagate throughout the network via the overlay that the name resolvers form among themselves (much like mirrored servers). Then a user's query about a composite service can be answered promptly by one or more name resolvers that reside near the user. The trade-offs between the updates of composite service information and broadcasts by users (current model) have to be studied more carefully.

9.2.3 Optimizations in the Embedding Process

Our instantiation algorithms map each node in a TG to a distinct device in the network. Although this may be realistic for a class of devices such as display screens (peripherals in general) sometimes multiple nodes in a TG can be mapped to a single device, (e.g., a general computing device). Our algorithms need to be slightly adapted to suit these requirements. The concept of maximum number of TG nodes mapped onto the same physical device can be utilized for load sharing and fairness. This can be useful in sparse networks where network partitions are very common. Even if the connected component of the user device contains a smaller number of devices than $|V_T|$, multiple nodes can be mapped onto single devices so that the entire TG is instantiated successfully.

We have introduced metrics in Appendix B that can be useful for optimization purposes in certain application scenarios. Examples of such metrics include load and node/edge congestion of the mapping. These metrics will have a direct impact on the energy usage of the MANET as a whole. Hence, instantiation protocols that attempt to optimize these metrics must be developed in future.

9.2.4 Better Handling of Burst Disconnects

Although the concepts of distributed TG-patching are simple and powerful in case of single and certain types of burst disconnects, they are inadequate for handling "affecting" burst disconnects. A centralized protocol (as proposed in [50]) is better suited for detecting such disconnects as a central controller monitors all the instances in the network periodically. Hence the main challenge here is to combine the merits of the centralized and distributed protocols in order to design better failure recovery schemes.

9.2.5 Implementation Challenges

In the proof-of-concept prototype described in Chapter 7, we barely touched upon the concepts of reliable execution of tasks. ADU buffering and end-to-end ACKs for ADUs must be implemented to offer reliability at the TaskApp layer. Recovery from the failure of certain distributed components is application-dependent and needs further investigation.

We used a mobility emulator to test large scale device mobility. This does not allow accurate performance analysis of metrics such as delay, and effective throughput – only real device mobility does. Systematic studies involving real mobility of devices need to be conducted and real tasks need to be evaluated on such networks.

Finally, a component currently missing from the architecture is security. Because we allow mobile TaskApp code to be injected into sandboxed environments, it exposes many security related issues that have been plaguing mobile code/agent based systems in the past few years. While this is easier to handle in controlled environments such as battlefield applications (all devices issued by a common agency), it is a more difficult problem in heterogeneous systems (such as a network of PDAs used for polling).

Appendix A

Approximation Factors for the TreeEmbed Algorithm

Since exact optimal polynomial time solutions are not known for NP-complete optimization problems, it is worthwhile to look for approximation algorithms which can yield suboptimal solutions in polynomial time, but with bounded suboptimality. An approximation factor characterizes the upper bound on the degree of suboptimality that a certain approximation algorithm can yield on a general instance of the input. We derive the approximation factors that the optimal tree-based graph embedding algorithm TreeEmbed (described in Chapter 4, Algorithm 4.1) can achieve when used for embedding non-tree task graphs.

Let $TG = (V, E)$ be a task graph that needs to be embedded onto a network G. Denote the tree induced by breadth first search on TG by TG_{BFS}. If TG is a tree, then as shown in Section 4.2.3, TreeEmbed yields the optimal *average dilation*. Let d^* be the optimal average dilation when $TG = TG_{BFS}$ is embedded onto G.

When $TG \neq TG_{BFS}$, the number of non-tree edges possessed by TG is $e - n + 1$, since there are $n - 1$ tree edges for $n = |V|$ and $e = |E|$. The $n - 1$ BFS edges in TG contribute $(n-1)d^*$ hops to the total length of mapped paths. Consider the non-BFS edges

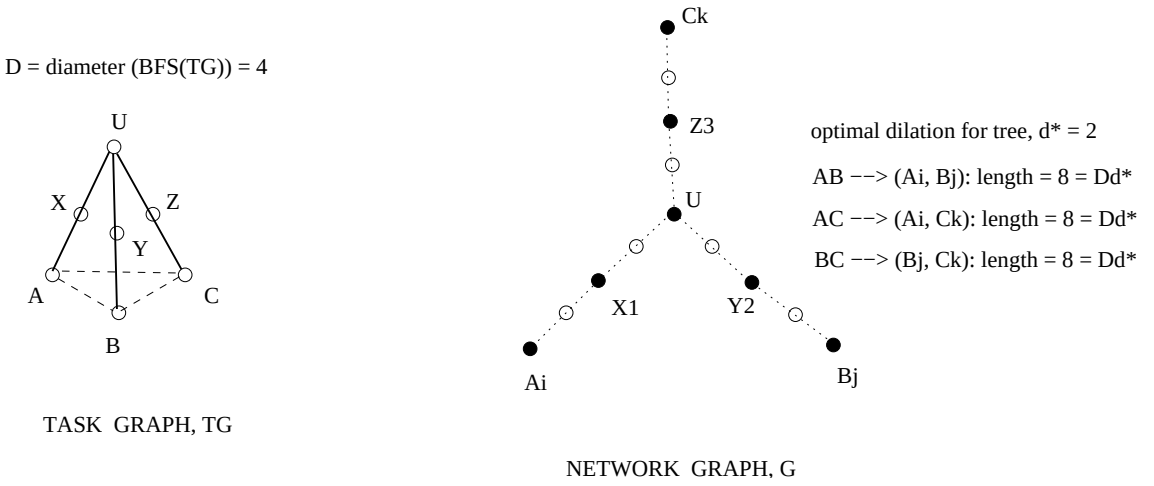


Figure A.1: The Worst Case Scenario for a Non-Tree Task Graph

in TG: in the worst case, each such edge can be mapped through a path in TG that is a subset of the edges in TG_{BFS}, and upper bounded by $D = diameter(TG_{BFS})$ in length. Hence, the contribution due to these edges to the total length of mapped paths is $\leqslant Dd^*$.

In the worst case, the upper bound of the dilation yielded by TreeEmbed algorithm is represented by $d^{TEMB}(G)$ which can be computed as follows:

$$\begin{aligned} d^{TEMB}(G) &\leqslant \frac{1}{e}[(n-1)d^* + (e-n+1)Dd^*] \\ &\leqslant Dd^* - \frac{1}{e}[(D-1)(n-1)d^*] \\ &= [D - \frac{(D-1)(n-1)}{e}]d^* \end{aligned} \tag{A.1}$$

Note that when TG is a tree, $e = n-1$, and hence Equation A.1 reduces to $d_{avg} = d^*$. The upper bound given by Equation A.1 is achieved only in some cases as shown in Figure A.1. In this example, the average dilation due to our heuristic algorithm is $d_{avg} = (4 - \frac{(4-1)\times(7-1)}{9})d^* = 2d^*$.

Let $\varphi^{OPT} : V_{TG} \rightarrow V_G$ be the optimal embedding that yields optimal average dilation $d^{OPT}(G)$. Denote $d_{BFST}(G)$ as the dilation obtained by embedding the BFS spanning tree of TG (the same as the one considered in Equation A.1) onto G using φ^{OPT}. By definition,

$d_{BFST}(G) \geqslant d^*$. The lower bound of the optimal dilation in terms of the dilation of this corresponding BFS subtree is achieved when all $e-n+1$ non-tree edges in TG are mapped to single hop paths in G. This is indicated by:

$$\begin{aligned} d^{OPT}(G) &\geqslant \frac{1}{e}[(n-1)d_{BFST}(G) + (e-n+1)] \\ &\geqslant \frac{1}{e}[(n-1)d^* + (e-n+1)] \end{aligned} \tag{A.2}$$

A.1 Calculation of the Approximation Factor

Definition A.1 An α-approximation algorithm for an optimization problem π is a polynomial time algorithm that outputs a solution whose cost is within the multiplicative factor $\alpha \geqslant 1$ of the optimal cost for any instance of the problem. In other words, if $C^{OPT}(x)$ is the cost yielded by an optimal algorithm for an input x, the cost yielded by an α-approximation algorithm on that same input is $C^{APPX}(x) \leqslant \alpha\, C^{OPT}(x)$.

From Equations A.1 and A.2, the approximation factor α of the TreeEmbed algorithm is calculated as follows:

$$\begin{aligned} \alpha &= \max\left\{\frac{d^{TEMB}(G)}{d^{OPT}(G)}\right\} \leqslant \frac{\max\ d^{TEMB}(G)}{\min\ d^{OPT}(G)} \\ &= \frac{(n-1)d^* + Dd^*(e-n+1)}{(n-1)d^* + (e-n+1)} \\ &= \frac{1 + D(\frac{e}{n-1} - 1)}{1 + \frac{1}{d^*}(\frac{e}{n-1} - 1)} \end{aligned} \tag{A.3}$$

The above upper bound on α is heavily dependent upon the structure of the TG. More specifically, the number of edges e and diameter D of TG_{BFS} (both are correlated) play a major role, as does the value of dilation (d^*) of embedding TG_{BFS}. We observe from Equation A.3 the trivial upper bound of $\alpha \leqslant Dd^*$ which is not tight.

In the remaining portion of this appendix, we derive tighter approximation factors of the TreeEmbed algorithm for various special cases of task graph embeddings onto

a general network, namely, the cycle graph C_n, the square mesh graph $M_{m\times m}$, and the complete graph K_n.

Theorem A.1 (Cycle graph) TreeEmbed *yields* $\alpha = 2$ *for a cycle graph* C_n.

Proof. Let $C_n = U, v_1, v_2, \ldots, v_{n-1}, U$. The BFS spanning tree of C_n is the line graph $L_n = U, v_1, v_2, \ldots, v_{n-1}$. Suppose d^* is the optimal average dilation obtained by embedding L_n onto a graph G using TreeEmbed. $d_T(G)$ denotes the dilation obtained by embedding C_n onto G by using TreeEmbed algorithm. Its lower bound is given by:

$$d_T(G) \geqslant \frac{d^*(n-1)+1}{n} = d^*(1-\frac{1}{n})+\frac{1}{n} \tag{A.4}$$

The above lower bound is achieved when the only non-tree edge of C_n, i.e., (v_{n-1}, U) is mapped onto a single hop path in G. The upper bound is achieved when the edge (v_{n-1}, U) is mapped onto a longer path constructed by concatenation of paths in G which are embeddings of the edges in L_n. It is given by:

$$d_T(G) \leqslant \frac{d^*(n-1)+d^*(n-1)}{n} = 2d^*(1-\frac{1}{n}) \tag{A.5}$$

Let $\varphi^{OPT} : C_n \rightarrow V_G$ be the optimal embedding which yields optimal average dilation $d^{OPT}(G)$. Denote by $d_{L_n}(G)$ the dilation obtained by embedding L_n onto G using φ^{OPT}. By definition, $d_{L_n}(G) \geqslant d^*$. The lower bound of the optimal dilation in terms of the dilation of this corresponding BFS subtree L_n is given by:

$$d^{OPT}(G) \geqslant d_{L_n}(G)(1-\frac{1}{n})+\frac{1}{n} \geqslant d^*(1-\frac{1}{n})+\frac{1}{n} \tag{A.6}$$

From Equations A.5 and A.6, the approximation factor α is calculated as follows:

$$\begin{aligned} \alpha &= \max\left\{\frac{d_T(G)}{d^{OPT}(G)}\right\} \leqslant \frac{\max\ d_T(G)}{\min\ d^{OPT}(G)} \\ &= \frac{2d^*(1-\frac{1}{n})}{d^*(1-\frac{1}{n})+\frac{1}{n}} = 2-\frac{2}{d^*(n-1)+1} \leqslant 2 \end{aligned} \tag{A.7}$$

□

Theorem A.2 (Square mesh) TreeEmbed *yields* $\alpha = \sqrt{n}$ *for a* $M_{m\times m}$ *mesh where* $n = m^2$.

Proof. Suppose the user node is at a corner of $M_{m\times m}$. A BFS tree T_M of $M_{m\times m}$ consists of a set of lines $L_{11,1m}, L_{21,2m}, \ldots, L_{m1,mm}$ and one additional line $L_{11,m1}$ that connects all these lines together to yield a tree structure. Let d^* be the optimal dilation obtained by TreeEmbed algorithm. Since there are $n-1$ edges in T_M, the contribution due to T_M to the total length of mapped paths is $d^*(n-1)$. There are $2m(m-1)$ edges in $M_{m\times m}$ out of which $2m(m-1)-(n-1) = (m-1)^2$ are non-tree edges that have not been explicitly mapped by TreeEmbed. $d_T(G)$ denotes the dilation obtained by embedding M onto G by using TreeEmbed algorithm. The upper bound of $d_T(G)$ is achieved when all the non-tree edges of M are mapped to long paths in G obtained by concatenating shorter paths which are embeddings of T_M. Therefore, $d_T(G)$ is given by:

$$d_T(G) \leqslant \frac{d^*(n-1) + (m-1)(3+5+\ldots+2m-1)d^*}{2m(m-1)} = \frac{m+1}{2}d^* \tag{A.8}$$

Let $\varphi^{OPT} : M \rightarrow V_G$ be the optimal embedding which yields optimal average dilation $d^{OPT}(G)$. Let us denote by $d_{T_M}(G)$ the dilation obtained by embedding T_M onto G using φ^{OPT}. By definition, $d_{T_M}(G) \geqslant d^*$. In the best case, all the non-tree edges in M will be mapped to single hop paths in G. Thus the lower bound of the optimal dilation in terms of the dilation of this corresponding BFS subtree T_M is given by:

$$\begin{aligned} d^{OPT}(G) &\geqslant \frac{d_{T_M}(G)(m^2-1) + (m-1)^2}{2m(m-1)} \\ &= \frac{1}{2}\Big\{d_{T_M}(G)(1+\frac{1}{m}) + (1-\frac{1}{m})\Big\} \\ &\geqslant \frac{1}{2}\Big\{d^*(1+\frac{1}{m}) + (1-\frac{1}{m})\Big\} \end{aligned} \tag{A.9}$$

From Equations A.8 and A.9, the approximation factor α can be calculated as

follows:

$$\begin{aligned}\alpha &= \max\Big\{\frac{d_T(G)}{d^{OPT}(G)}\Big\} \leqslant \frac{\max\ d_T(G)}{\min\ d^{OPT}(G)} \\ &= \frac{m+1}{2}d^* \Big/ \frac{1}{2}\Big\{d^*(1+\frac{1}{m}) + (1-\frac{1}{m})\Big\} \\ &= m\Big\{1 - \frac{m-1}{(m+1)d^* + (m-1)}\Big\} \leqslant m = \sqrt{n} \end{aligned} \tag{A.10}$$

□

Theorem A.3 (Complete Graph) TreeEmbed *yields* $\alpha = n - 1$ *for a complete graph* K_n.

Proof. The BFS tree of K_n is a star graph S_n on n vertices with $n-1$ leaf nodes. The number of edges $e = \binom{n}{2} = \frac{1}{2}n(n-1)$. Hence the number of non-tree edges is $e - n + 1 = \binom{n-1}{2} = \frac{1}{2}(n-1)(n-2)$. Using notation described earlier in this section, we calculate the upper bound for $d_T(G)$ and the lower bound for $d^{OPT}(G)$ as follows:

$$d_T(G) \leqslant \frac{d^*(n-1) + \binom{n-1}{2}2d^*}{\binom{n}{2}} = 2d^*(1 - \frac{1}{n}) \tag{A.11}$$

$$d^{OPT}(G) \geqslant \frac{d_{S_n}(G)(n-1) + \binom{n-1}{2}}{\binom{n}{2}} \geqslant \frac{2d^* + n - 2}{n} \tag{A.12}$$

From Equations A.11 and A.12, the approximation factor α can be calculated as follows:

$$\alpha = \max\Big\{\frac{d_T(G)}{d^{OPT}(G)}\Big\} \leqslant \frac{\max\ d_T(G)}{\min\ d^{OPT}(G)} = \frac{2d^*(n-1)}{2d^* + n - 2} = \frac{n-1}{1 + \frac{n-2}{2d^*}} \tag{A.13}$$

Unlike in the previous special cases, the value of d^* impacts α significantly. If G is large and d^* is high, then α tends to increase and is upper bounded by $n-1$. In other words, $\alpha \leqslant n - 1$. □

Appendix B

Metrics for Performance Evaluation of Distributed Anycastable Applications

As proposed throughout this dissertation, several devices, specialized or multipurpose, can participate in the execution of a given distributed task. Also, there are likely to be multiple occurrences of devices that offer similar services, hence a user need not be bothered about the choice of particular devices that participate in the task, as long as it is completed successfully. Therefore, the quality of the overall choice of devices that participate in a task must be quantified in such a framework.

This material is excerpted from [6]. As mentioned in Chapter 3, a user specifies the structure of an application by means of a task graph or resource dependency graph consisting of nodes representing "logical" computing resources and edges representing data flow dependencies or other requirements such as physical proximity between nodes, to facilitate the application. At runtime, particular instances of devices/resources need to be selected from among multiple available instances for efficient execution of the given application. We

refer to this process as Application Mapping or Instantiation. Note that the issues here are different from those in simple service discovery since we are concerned with the discovery of all resources needed for the execution of the entire application while obeying the constraints specified in the task graph. We also need to measure the performance of the actual execution process as a function of several system parameters.

In earlier chapters of this dissertation, we proposed and utilized system level metrics to quantify success or failure of performing a particular distributed "anycastable" task. In this appendix, we propose a taxonomy for various types of metrics which can measure the performance of applications during their entire lifetimes. For completeness we enumerate additional metrics which we believe help in the process of performance evaluation of such applications:

1. **Application mapping time**: Time taken to map or instantiate an application characterized by a resource graph onto a set of suitable devices which will actually execute the application.

2. **Application mapping efficiency**: The following metrics can evaluate different aspects of the mapping efficiency.

   (a) **Average/Maximum Dilation**: Each edge of the task graph can get mapped onto a multi-hop shortest path in the network. This metric measures the average/maximum length of a shortest path between two instantiated devices in the network after the mapping is complete. Its value is always $\geqslant 1$. A low dilation mapping is good because if most required devices are located nearby, the probability of achieving high overall throughput are greater.

   (b) **Node Congestion**: A mapping with low dilation can suffer from bottleneck paths passing through a single device or a few devices. Minimization of this metric encourages mappings with lesser number of paths passing through bottleneck devices. Hence, this automatically helps in load balancing.

(c) **Messaging Overhead**: This metric is characterized by the number of control messages exchanged between devices during the establishment of a mapping. Broadcast based solutions tend to have a high message overhead although they are usually better suited for highly mobile networks.

3. **Resilience of Application Mapping to Device Mobility**: Mobility of devices can cause network partitions and therefore, application outages. The following set of metrics measure the impact of device mobility on an application mapping.

   (a) **Frequency of Application Disruption**: The rate at which an application is disrupted due to network partitions and device failures.

   (b) **Application Recovery Time**: Time taken to discover replacement devices for re-mapping the affected parts of the resource graph plus the time to perform state migration to these new devices.

   (c) **Average Connected time**: Time for which an application runs without any disruptions.

4. **Application Performance after Mapping**: This set of metrics attempts to measure how well an application executes after all the required resources have been discovered and instantiated.

   (a) **Average Effective Throughput**: The cumulative ratio of the number of ADUs received by their intended recipients to those were intended to be received under perfect network conditions. This metric is normalized with respect to the number of data sinks in a task graph and attains values between 0 and 1. Mobility of devices can cause network partitions which can result in task disruptions, hence lowering this. This metric is usually an indicator of the loss of application data.

   (b) **Average number of Re-transmissions**: Since application data can be lost because of disruptions, buffering and re-transmissions may be required at data sources. This metric measures the number of times a data unit is needed to be re-transmitted before successful reception by an intended instantiated recipient.

(c) **Average ADU Delay**: The amount of time that elapsed between the transmission of an application data unit and its successful reception at an intended recipient. This may include the recovery time following a disruption.

We note that different distributed applications on a MANET can have different levels of tolerance with respect to the metrics proposed above (e.g., some applications can tolerate large delays due to re-mappings but not data loss.) Hence a designer of a particular application must consider the relative importance of the metrics for the application. We have not attempted to optimize our protocols with respect to all of these metrics in this dissertation. This remains a topic of future research.

Bibliography

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. "The Design and Implementation of an Intentional Naming System". In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, SC, December 1999.

[2] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. "Resilient Overlay Networks". In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, October 2001.

[3] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu, and D. Zappala. "Improving Simulation for Network Research". Technical Report 99-702, University of Southern California, Los Angeles, CA, March 1999. URL: `http://www.isi.edu/nsnam/ns`.

[4] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman, and D. Zukowski. "Challenges: An Application Model for Pervasive Computing". In *Proceedings of the 6th ACM MobiCom Conference*, Boston, MA, August 2000.

[5] S. Basagni. "Distributed Clustering for ad hoc Networks". In A. Y. Zomaya, D. F. Hsu, O. Ibarra, S. Origuchi, D. Nassimi, and M. Palis, editors, *Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms, and Networks*

(I-SPAN'99), pages 310–315, Perth/Fremantle, Australia, June 1999. IEEE Computer Society.

[6] P. Basu, W. Ke, and T. D. C. Little. "Metrics for Performance Evaluation of Distributed Application Execution in Ubiquitous Computing Environments". In *Position paper at the ACM UbiComp 2001 Workshop on Evaluation Methodologies for Ubiquitous Computing*, Atlanta, GA, September 2001.

[7] P. Basu, W. Ke, and T. D. C. Little. "A New Task-Based Approach for Supporting Distributed Applications on Mobile Ad Hoc Networks". *Computer Communication Review*, 32(1):63–63, January 2002. Published as a summary of a poster that appeared in the ACM SIGCOMM Conference, San Diego, CA, 2001.

[8] P. Basu, W. Ke, and T. D. C. Little. "A Novel Approach for Execution of Distributed Tasks on Mobile Ad Hoc Networks". In *Proceedings of the IEEE Wireless Computing and Networking Conference (WCNC)*, Orlando, FL, March 2002.

[9] P. Basu, W. Ke, and T. D. C. Little. "Scalable Service Composition in Mobile Ad hoc Networks using Hierarchical Task Graphs". In *Proceedings of the 1st Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net)*, Sardegna, Italy, September 2002. IFIP.

[10] P. Basu, W. Ke, and T. D. C. Little. "Dynamic Task Based Anycasting in Mobile Ad Hoc Networks". *ACM/Kluwer Journal for Mobile Networks and Applications (MONET)*, 2003. In Press.

[11] P. Basu and T. D. C. Little. "Networked Parking Spaces: Architecture and Applications". In *Proceedings of the 56th IEEE Vehicular Technology Conference - Fall*, Vancouver, Canada, September 2002.

[12] P. Basu, N.Khan, and T. D. C. Little. "A Mobility Based Metric for Clustering in Mobile Ad Hoc Networks". In *Proceedings of the IEEE ICDCS Workshop on Wireless Networks and Mobile Computing*, Phoenix/Mesa, AZ, April 2001.

[13] J. Beck, A. Gefflaut, and N. Islam. "MOCA: A Service Framework for Mobile Computing Devices". In *Proceedings of the International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE)*, Seattle, WA, August 1999.

[14] D. P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 1. Athena Scientific, Belmont, MA, 1995.

[15] V. Bharghavan, A. Demers, S. Shenker, and L. Zhang. "MACAW: A Media Access Protocol for Wireless LANs". In *Proceedings of the ACM SIGCOMM Conference*, pages 212–225, London, UK, September 1994.

[16] S. Bhattacharjee, M. Ammar, E. Zegura, V. Shah, and Z. Fei. "Application Layer Anycasting". In *Proceedings of the IEEE INFOCOM Conference*, Kobe, Japan, April 1997.

[17] Bluetooth Consortium. URL. `http://www.bluetooth.com`.

[18] D. Boggs. *Internet Broadcasting*. PhD thesis, Xerox PARC, October 1983. Available as Technical Report CSL-83-3.

[19] S. Bokhari. "On the Mapping Problem". *IEEE Transactions on Computers*, 30(3), 1981.

[20] J. Broch, D. A. Maltz, D. B. Johnson, Y. C. Hu, and J. Jetcheva. "A Performance Comparison of Multi-Hop Ad Hoc Network Routing Protocols". In *Proceedings of the 4th ACM MobiCom Conference*, Dallas, TX, 1998.

[21] D. Chakraborty, F. Perich, A. Joshi, T. Finin, and Y. Yesha. "A Reactive Service Composition Architecture for Pervasive Computing Environments". Technical Report TR-CS-02-02, University of Maryland at Baltimore County, Department of Computer Science, March 2002.

[22] K. Chandran, S. Raghunathan, S. Venkatesan, and R. Prakash. "A Feedback-Based Scheme for Improving TCP Performance in Ad Hoc Wireless Networks". *IEEE Personal Communications Magazine*, February 2001.

[23] Y. Chu, S. G. Rao, and H. Zhang. "A Case For End System Multicast". In *Proceedings of the ACM SIGMETRICS Conference*, pages 1–12, Santa Clara, CA, June 2000.

[24] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1990.

[25] S. Corson and J. Macker. Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations. RFC 2501 (Informational), January 1999. URL: `http://www.ietf.org/rfc/rfc2501.txt`.

[26] B. P. Crow, I. Widjaja, J. G. Kim, and P. T. Sakai. "IEEE 802.11 Wireless Local Area Networks". *IEEE Communications Magazine*, 35(9):116–126, September 1997.

[27] S. Deering. *"Scalable Multicast Routing Protocol"*. PhD thesis, Stanford University, 1989.

[28] A. Ephremides, J. E. Wieselthier, and D. J. Baker. "A Design Concept for Reliable Mobile Radio Networks with Frequency Hopping Signaling". *Proceedings of the IEEE*, 75(1):56–73, January 1987.

[29] H. Eriksson. "Mbone: The Multicast Backbone". *Communications of the ACM*, 37(8):54–60, 1994.

[30] M. Esler, J. Hightower, T. Anderson, and G. Borriello. "Next Century Challenges: Data-Centric Networking for Invisible Computing The Portolano Project at the University of Washington". In *Proceedings of the 5th ACM MobiCom Conference*, Seattle, WA, August 1999.

[31] The "Familiar" Project: Linux Distribution for Compaq iPAQ. URL. `http://familiar.handhelds.org/`.

[32] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison Wesley, Reading, MA, November 1999.

[33] Z. Fu, P. Zerfos, H. Luo, S. Lu, L. Zhang, and M. Gerla. "The Impact of Multihop Wireless Channel on TCP Throughput and Loss". In *Proceedings of the IEEE INFOCOM Conference*, San Francisco, CA, March-April 2003.

[34] M.R. Garey and D.S. Johnson. *Computers and Intractability*. W.H. Freeman and Company, San Francisco, CA, 1979.

[35] M. Gerla and J. T.-C. Tsai. "Multicluster, Mobile Multimedia Radio Network". *Wireless Networks 1*, pages 255–265, 1995.

[36] J. D. Gibson, editor. *The Mobile Communications Handbook*. CRC Press Inc., 1996.

[37] R. S. Gray. "Agent Tcl: A Flexible and Secure Mobile-agent System". In M. Diekhans and M. Roseman, editors, *Proceeding of Fourth Annual USENIX Tcl/Tk Workshop*, pages 9–23, Monterey, CA, 1996.

[38] E. Guttman. "Service Location Protocol: Automatic Discovery of IP Network Services". *IEEE Internet Computing*, July 1999.

[39] J. Guyton and M. Schwartz. "Locating Nearby Copies of Replicated Internet Servers". In *Proceedings of the ACM SIGCOMM Conference*, Cambridge, MA, August 1995.

[40] Z. J. Haas. "A New Routing Protocol for the Reconfigurable Wireless Networks". In *Proceedings of the IEEE International Conference on Universal Personal Communication (ICUPC)*, San Diego, CA, October 1997.

[41] T. Hodes, R. Katz, E. Servan-Screiber, and L. Rowe. "Composable Ad-Hoc Mobile Services for Universal Interaction". In *Proceedings of the 3rd ACM MobiCom Conference*, Budapest, Hungary, September 1997.

[42] G. Holland and N. Vaidya. "Analysis of TCP Performance over Mobile Ad Hoc Networks". In *Proceedings of the 5th ACM MobiCom Conference*, pages 219–230, Seattle, WA, August 1999.

[43] C. C. Hui and S. T. Chanson. "Allocating Task Interaction Graphs to Processors in Heterogeneous Networks". *IEEE Transactions On Parallel And Distributed Systems*, 8(9):908–925, September 1997.

[44] P. Jacquet, P. Muhlethaler, T. Clausen, A. Laouiti, A. Qayyum, and L. Viennot. "Optimized Link State Routing Protocol for Ad Hoc Networks". In *Proceedings of IEEE INMIC*, Lahore, Pakistan, 2001.

[45] P. Jacquet, L. Viennot, and T. H. Clausen. "Analyzing control Traffic Overhead in Mobile Ad-hoc Network Protocols versus Mobility and Data Traffic Activity". In *Proceedings of the 1st Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net)*, Sardegna, Italy, September 2002. IFIP.

[46] D. B. Johnson and D. A. Maltz. "Dynamic Source Routing in Ad Hoc Wireless Networks". In T. Imielinski and H. Korth, editors, *Mobile Computing*, chapter 5, pages 153–181. Kluwer Academic Publishers, 1996.

[47] J. Jubin and J. D. Tornow. "DARPA Packet Radio Network Protocols". *Proceedings of the IEEE*, 75(1):21–32, January 1987.

[48] P. Karn. "MACA – A New Channel Access Method for Packet Radio". In *Proceedings of the 9th ARRL/CRRL Amateur Radio Computer Networking Conference*, pages 134–140, London, Ontario, Canada, September 1990.

[49] B. Karp and H. T. Kung. "Greedy Perimeter Stateless Routing for Wireless Networks". In *Proceedings of the 6th ACM MobiCom Conference*, pages 243–254, Boston, MA, August 2000.

[50] W. Ke, P. Basu, and T. D. C. Little. "A Task Graph Based Application Framework for Mobile Ad Hoc Networks". In *Proceedings of the IEEE International Conference on Communications (ICC)*, New York, NY, April-May 2002.

[51] Y. Ko and N. H. Vaidya. "Location Aided Routing in Ad Hoc Networks". In *Proceedings of the 4th ACM MobiCom Conference*, Dallas, TX, October 1998.

[52] P. Krishna, N. H. Vaidya, M. Chatterjee, and D. K. Pradhan. "A Cluster Based Approach for Routing in Ad Hoc Networks". *ACM SIGCOMM Computer Communication Review*, 1997.

[53] S. Lee, W. Su, J. Hsu, M. Gerla, and R. Bagrodia. "A Performance Comparison Study of ad hoc Wireless Multicast Protocols". In *Proceedings of the IEEE INFOCOM Conference*, pages 565–574, Tel-Aviv, Israel, March 2000.

[54] H. Lundgren, E. Nordström, and C. Tschudin. "Coping with Communication Gray Zones in IEEE 802.11b based Ad hoc Networks". In *Proceedings of the WoWMoM Workshop*.

[55] K. Mills. "AirJava: Networking for Smart Spaces". In *Proceedings of the USENIX Embedded Systems Workshop*, Cambridge, MA, March 1999.

[56] R. Monien and H. Sudborough. "Embedding one Interconnection Network in Another". *Computing Suppl.*, 7:257–282, 1990.

[57] S. Oaks and H. Wong. *JINI in a Nutshell*. O'Reilly, first edition, March 2000.

[58] OLSR Implementation for Linux: INRIA, France. URL. `http://hipercom.inria.fr/olsr/`.

[59] V. D. Park and J. Macker. "Anycast Routing for Mobile Services". In *Proceedings of the 33rd Annual Conference on Information Sciences and Systems (CISS)*, Baltimore, MD, March 1999.

[60] C. Partridge, T. Mendez, and W. Milliken. Host Anycasting Service. RFC 1546 (Informational), November 1993. URL: `http://www.ietf.org/rfc/rfc1546.txt`.

[61] C. E. Perkins and E. M. Royer. "Ad hoc On-Demand Distance Vector Routing". In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, pages 90–100, New Orleans, LA, February 1999.

[62] R. Ramanathan and M. Steenstrup. "Hierarchically-Organized Multihop Mobile Networks for Quality-of-service Support". *ACM/Baltzer Journal on Mobile Networks and Applications*, 3(2), August 1998.

[63] Project: Rapid Task-based Self-Organization in Distributed Ad-hoc Spaces, Multimedia Communication Laboratory, Boston University. URL, 2000–2003. `http://hulk.bu.edu`.

[64] P. Sass and J. Freebersyser. "FCS Communications Technology for the Objective Force". Technical report, MITRE Corporation, 2002.

[65] M. Steenstrup. "Cluster-based Networks". In C. Perkins, editor, *Ad hoc Networking*, pages 75–138. Addison-Wesley Longman Publishing Co., Inc., 2001.

[66] W. R. Stevens. *UNIX Network Programming*, volume 1. Prentice Hall PTR, Upper Saddle River, New Jersey, second edition.

[67] J. Tardo and L. Valente. "Mobile Agent Security and Telescript". In *Proceedings of IEEE CompCon*, pages 58–63, 1996.

[68] Y.-C. Tseng, S.-Y. Ni, Y.-S. Chen, and J.-P. Sheu. "The Broadcast Storm Problem in a Mobile ad hoc Network". In *Proceedings of the 5th ACM MobiCom Conference*, pages 151–162, Seattle, WA, August 1999.

[69] D. Wall. *"Mechanisms for Broadcast and Selective Broadcast"*. PhD thesis, Stanford University, 1980.

[70] M. Weiser. "Some Computer Science Issues in Ubiquitous Computing". *Communications of the ACM*, 36(7):75–85, July 1993.

[71] Y. Zhang and W. Li. "An Integrated Environment for Testing Mobile Ad-Hoc Networks". In *Proceedings of the 3rd ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, Lausanne, Switzerland, June 2002.

[72] ZigBee Alliance. URL. `http://www.zigbee.org`.

Biography

Prithwish Basu is a Ph.D. candidate in the Department of Electrical and Computer Engineering at Boston University (BU). He received the B.Tech. degree in Computer Science and Engineering from Indian Institute of Technology (IIT), New Delhi, India in 1996, and the M.S. degree in Computer Systems Engineering from Boston University in January 1999. He is currently a research assistant in the Multimedia Communications Laboratory at Boston University. His research interests include resource discovery, routing, and clustering in mobile ad hoc networks; algorithmic issues in mobile and pervasive computing; scalable network video delivery; and performance analysis and modeling. Prithwish has co-authored over ten conference and journal articles, and two invited book chapters during his stay at Boston University. He received the Student Travel Award at MobiCom 2002.

Prithwish is a student member of the IEEE and the ACM. He was a member of the organizing committee for MobiCom 2000 held in Boston in the capacity of the coordinator of student volunteers. He has also served as a reviewer for a number of conferences and journals over the past six years.