DISTRIBUTED SERVICES IN PERVASIVE SYSTEMS

by

SAGAR A. TAMHANE

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2010

Copyright © by SAGAR A. TAMHANE 2010

All Rights Reserved

To my parents Neela and Ashok, my sister Prachi, my niece Swara and my wife Aditee.

ACKNOWLEDGEMENTS

I would like to thank my supervising professor Dr. Mohan Kumar for his constant motivation, encouragement and his invaluable advice. Without his support, it would not have been possible to complete this dissertation. I would also like to thank Dr. Hao Che, Dr. David Kung, Dr. Donggang Liu and Dr. Bob Weems for their interest in my research and for taking time to serve in my dissertation committee.

I would like to thank the CSE department for the support and encouragement throughout my stay at UTA. I am grateful to the Graduate School for providing the STEM fellowship.

The research work presented in this dissertation was supported in part by the US National Science Foundation Grants ECCS-0824120 and CSR 0834493. I am grateful to the National Science Foundation for providing financial support during my doctoral studies.

I would also like to thank my colleagues at the Pervasive and Invisible Computing (PICo) lab for their feedback on my research work and their cooperation.

November 15, 2010

ABSTRACT

DISTRIBUTED SERVICES IN PERVASIVE SYSTEMS

SAGAR A. TAMHANE, Ph.D.

The University of Texas at Arlington, 2010

Supervising Professor: Mohan Kumar

Devices in pervasive systems are generally resource constrained, heterogeneous, personal, and mobile. These constraints limit the services and quality of service offered by the devices. For example, execution of such tasks as video and audio processing might not be possible or successful on cell phones, PDAs and sensors. Hence interactions between devices are exploited to perform collaborative task execution. Devices with better resource levels perform tasks on behalf of other devices. This dissertation provides algorithms for cooperative service executions in pervasive environments. The major contributions of this dissertation are:

- Cluster Based Scheduling (CBS) algorithm that performs service scheduling in smart spaces.
- Decentralized Grading Autonomous Selection (DGAS) algorithm that performs fault tolerant service execution in Mobile Ad hoc Networks.
- Mutual Exclusion for Opportunistic Networks (MEOP) algorithm that facilitates exclusive access to shared resources in opportunistic networks (OPNET).
- Middleware architecture for devices in OPNETs.
- An algorithm for performing service composition in OPNETs.

The main advantages of CBS over existing schemes are - reduced communication and storage overhead, and support for usage of multiple task scheduling algorithms. CBS factors such challenges as device heterogeneity, service availability and device mobility into the scheduling algorithm. DGAS provides fault tolerance by replicating service execution onto multiple devices. Decision about participation of devices in service executions are taken autonomously by each device. DGAS avoids usage of any central entity. When multiple devices need exclusive access to a shared resource, middleware installed on the devices should facilitate the access. MEOP is a token requesting algorithm that provides exclusive access to resources in OPNETs. MEOP has lower communication overhead as compared to token ring, permission based and centralized algorithms. A middleware architecture is proposed for devices in OPNETs. The middleware is modular. Each device can implement any subset of modules. This dissertation also presents service composition algorithms for OPNETs. Analysis of success probability of task execution and the length of compositions is presented and verified. A prototype of the service composition algorithms is implemented on Bluetooth enabled devices carried by students at the University of Texas at Arlington.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iv

ABSTRACT v

LIST OF FIGURES xi

LIST OF TABLES xiii

Chapter Page

1. INTRODUCTION 1

1.1 Motivation 2

1.1.1 Need for distributed service execution 4

1.1.2 Need for parallel processing 5

1.2 Challenges in distributed computing over pervasive systems 6

1.2.1 User mobility 6

1.2.2 Random failures 7

1.2.3 Device heterogeneity 7

1.2.4 Energy 7

1.2.5 Interaction between user and devices 8

1.2.6 Security 8

1.3 Major research contributions 8

1.3.1 Cluster Based Scheduling (CBS) 8

1.3.2 Distributed Grading Autonomous Selection (DGAS) 9

1.3.3 Mutual Exclusion in Opportunistic Networks (MEOP) 10

1.3.4 Service composition in OPNETs 11

1.4 Organization of the dissertation 12

2. BACKGROUND AND RELATED WORK 13
2.1 Offloading executions to high end servers 14
2.2 Offloading executions to mobile devices 16
2.3 Information sharing in vehicular ad hoc network 20
2.4 Information sharing in opportunistic networks 22
2.5 Mutual exclusion (M.E.) 25
2.5.1 Token based exclusion 26
2.5.2 Permission based exclusion 28
2.6 Service composition in pervasive systems 29
2.7 Task scheduling algorithms 31
2.8 Discussion on related work 32
3. DISTRIBUTED SERVICE EXECUTION IN SMART ENVIRONMENTS 34
3.1 Network model 34
3.2 Task model 35
3.3 Client - cluster communication 36
3.4 Intra-cluster scheduling 37
3.5 Simulations 41
3.5.1 Comparison metrics 41
3.5.2 Random graph generation 42
3.6 Summary and discussion 45
4. DISTRIBUTED SERVICE EXECUTION IN MANETS 47
4.1 Network model 47
4.2 Centralized Grading and Centralized Selection (CGCS) 48
4.3 Middleware modules 49
4.3.1 Agent 49
4.3.2 Participant 49

4.4 Decentralized grading 50

4.5 Decentralized Grading and Centralized Selection (DGCS) 54

4.6 Decentralized Grading and Autonomous Selection (DGAS) 56

4.7 Simulations 58

4.8 Summary and discussion 66

5. MUTUAL EXCLUSION FOR DISTRIBUTED SERVICES IN OPPORTUNISTIC NETWORKS 68

5.1 Network model 69

5.2 Mutual exclusion 69

5.2.1 Request generation 73

5.2.2 Request propagation 73

5.2.3 Token propagation 75

5.2.4 Fault detection 77

5.2.5 Correctness properties 79

5.3 Simulations for mutual exclusion 82

5.4 Summary and discussion 91

6. SERVICE COMPOSITION IN OPPORTUNISTIC NETWORKS 94

6.1 Introduction 94

6.2 Proposed middleware 96

6.3 Pair-wise contact 100

6.4 Service composition 101

6.4.1 Success probability 104

6.4.2 Expected length of composition 106

6.5 Fault tolerance 107

6.6 Simulation studies and implementation 109

6.6.1 Success probability 110

6.6.2 Length of composition 111
6.6.3 Time and number of devices required to complete service execution . . 113
6.6.4 Requests with multiple high level tasks 116
6.6.5 Fault tolerance 117
6.7 Prototype using mobile devices in university campus 118
6.8 Summary and discussion 121
7. CONCLUSIONS AND FUTURE WORK 124
7.1 Summary of Contributions 124
7.2 Future research directions 126
Appendix
A. PUBLICATIONS FROM THIS DISSERTATION 128
REFERENCES 130
BIOGRAPHICAL STATEMENT 139

LIST OF FIGURES

Figure		Page
1.1	Example of a pervasive system	3
3.1	Example task graph, device connectivity, availability of services and devices	35
3.2	Example: Selection of tasks into $\mathbb{P}$ based on value of λ	39
3.3	Effect of CCR on average NSL. (a) CCR = 0.1 (b) CCR = 1 (c) CCR = 10	43
3.4	Efficiency comparison ($CCR = 0.01, n = 20$)	44
3.5	Ratios of energy spent in communication overhead of A: algorithm in [1], B: MPI-IOS algorithm with respect to C: CBS	45
3.6	Effect of device failure on NSL of CBS	45
3.7	Ratio of energy consumption of CBS compared to a centralized algorithm	46
4.1	State diagrams of Agent in DGAS	49
4.2	State diagram of Participant in DGAS	50
4.3	Number of service providers having success probability > 0	59
4.4	Number of replicas executed when service providers follow a distribution	62
4.5	Number of replicas executed with random number of service providers	63
4.6	Number of replies received without and with opportunistic spreading of replication request (using DGAS)	64
4.7	Effect of selected parameters on number of replicas executed using DGAS	65
4.8	Number of replicas executed using PDAs and netbooks	66
5.1	State transition diagram of application requiring mutual exclusion	70
5.2	Example of opportunistic data transfer. (a) n_i, n_j have partial information. (b) n_i, n_j opportunistically exchange information	71
5.3	Example of updates to the logical DAG	74
5.4	Effect of varying λ. (a) Average number of messages transmitted. (b) Average	

waiting time per CS request 84

5.5 Comparison of MEOP with permission based and centralized algorithms 86

5.6 Effect of varying average time required to execute CS 87

5.7 Effect of varying percentage of nodes that generate token request 88

5.8 Effect for device failures on MEOP. (a) Average number of messages transmitted per CS. (b) Average number of CS executed 89

5.9 Time and community based selection for token destination. (a) Average number of messages transmitted per CS. (b) Average waiting time per CS 90

5.10 Effect of varying λ, on the Haggle dataset 92

5.11 Effect of varying λ, while using trace from UTA 93

6.1 Middleware Architecture 97

6.2 Sample graphs at nodes n_a and n_b created during pairwise contacts.
(a) G_a^1: paths via n_a (b) G_b^1: paths via n_b (c) $G_{a,b}^2$: paths via n_a and n_b 101

6.3 Service Composition 102

6.4 Sample combined graph of replicated composite service 108

6.5 Probability of successfully finding a transformation between the requested formats. (a) $M = 2$. (b) $M = 5$. (c) $M = 10$. (d) $M = 15$ 112

6.6 Probability of length of composition. (a) $M = 2$. (b) $M = 5$.
(c) $M = 10$. (d) $M = 15$ 113

6.7 Simulation: Average delay in completing the composite service 114

6.8 Simulation: Average number of devices in composition. (a) 1 to 12 number of services per device. (b) 25 to 29 number of services per device 114

6.9 Implementation: Average delay while using Haggle dataset 115

6.10 Implementation: Average number of devices in composition
while using Haggle dataset 116

6.11 Probability of success when request length is 3 117

6.12 Probability of length of composition 117

6.13 Probability of success with and without fault tolerance 118

6.14 Probability of length of composition 120

LIST OF TABLES

Table Page

2.1 Comparison between mutual exclusion algorithms applied to OPNETs 30

4.1 Example Journeys 53

4.2 Time (sec) required to start playing audio, with small number of devices and two agents in the network 60

4.3 Time (sec) required to start playing audio, with large number of devices and 25 agents in the network 60

6.1 Probability of successfully composing a service 120

6.2 Average intercontact time (hours) between devices. 121

6.3 Average delay and number of devices involved per composition 122

CHAPTER 1

INTRODUCTION

The concept of pervasive computing was prophesied by Mark Weiser in [2]. A pervasive environment provides services to its users in a transparent way such that users get services whenever, wherever and however they want [3]. Pervasive services have found applications in entertainment industry, healthcare, military, and other areas. Examples of such applications include monitoring of patients in homes for the elderly, monitoring of vehicular traffic and avoidance of traffic congestion, proactive personalization of environments according to users' preferences, peer-to-peer information caching, and sharing and cooperation among unmanned robots. Pervasive services have generally been deployed over following types of networks: Wireless Sensor Networks (WSN), smart space networks (such as intelligent home or classroom), Mobile Ad hoc Networks (MANET), Vehicular Ad hoc Networks (VANET) and Opportunistic Networks (OPNET).

Pervasive systems typically comprise of such devices as sensors, PDAs, cell phones and laptops. A device can participate in different types of pervasive networks at different times. For example, consider a PDA to be initially in an intelligent home. When the owner of the device commutes to his work place, the PDA becomes part of a VANET. After reaching his destination, the PDA becomes part of a MANET created using devices in the building.

Each device in a pervasive environment possesses a set of hardware and software resources and serves as host to application services. The devices have limited computation capabilities, limited residual energy and limited memory. A user may wish to perform tasks that, if performed by the user's device alone, are not achievable or can be performed with low quality of service; but when multiple devices are used, are achievable or can be performed with better

QoS. For example, pervasive devices host software services that can compress/decompress, encode/decode data or perform audio/video/image processing. Devices that have better resource levels perform such services on behalf of resource constrained devices. Hence there is a need for sharing of resources among devices. The process of offloading computational tasks from one device to another is called as *Cyberforaging*. Cyberforaging is performed only if the offloading results in energy and/or time savings.

In the past, researchers working on cyberforaging have focused on offloading of computations to immobile and high end servers. In the literature, most of the algorithms proposed for cyberforaging use centralized servers to perform decision making. The assumption of presence of such high-end servers is not valid in all pervasive networks. Depending on users' location and other devices in the network, a device might not have access to high-end servers to perform service execution.

This dissertation presents algorithms that perform service execution in pervasive systems that are devoid of high end and static servers. Section 1.1 motivates the reader by giving examples and describing the need for decentralized and distributed service execution in pervasive environments. Section 1.2 lists some of the challenges that should be handled. The chapter ends with a description of contributions of this dissertation.

1.1 Motivation

Pervasive systems comprise an ad hoc network of such devices as cell phones, handheld computers, laptops, sensors and others. Pervasive systems are essentially parallel and distributed computing systems that are networked using wired and/or various wireless technologies. Figure 1.1 shows an example of some of the devices and their communication connectivity. Some of these devices are mobile while some are static and connected to infrastructure. Such a network can be used to implement applications such as target tracking, where

Figure 1.1. Example of a pervasive system.

sensors and cameras and autonomous vehicles are deployed to gather information which is sent to a laptop. The laptop performs object/target recognition and raises alarms on personal devices. Users with PDAs and cell phones can control the information gathered from sensors and specify features to be extracted.

An intelligent home may contain motion sensors, light sensors, security cameras, along with cell phones, PDAs, personal computers belonging to the residents. Additionally such appliances as coffee maker, TV, DVD player can be added to the network. The residents can then utilize the devices to perform such interesting applications as detect that a resident has woken up in the morning and start her coffee maker automatically. If a resident is watching TV in one room and she moves to another room, the TV program migrates onto device(s) closest to her.

Sensors mounted in vehicles can prevent collisions, provide guidance in parking and lane changing and warn the driver about vehicle part failures. GPS devices are widely used to know the location of the vehicle. Vehicle to vehicle communication is performed to spread

notifications and alerts about traffic and/or accidents on the road. Mobility of devices has been used to an advantage to perform service execution [4] and information dissemination [5].

Svensson et al [6] use services provided by multiple devices to create a useful application. The authors compose services provided by a camera, a GPS locator and a backend database service. The developed application is used by landscape architects. When an architect takes a picture, the image is automatically tagged with the architect's GPS coordinates. The image is then automatically transmitted and stored into a database. Thus interaction between devices is exploited to automate a process.

Aiello and Dustdar [7] and Chowdhury et al. [8] have implemented an application for monitoring of elders in an intelligent home. Services provided by devices such as cameras, accelerometers, context recognition service and a database service are composed together. The application recognizes whether a person has fallen down. The application then identifies the person, locates his medical information and informs care takers about location of the elderly person. The application involves compute intensive tasks such as image/video processing during object/person recognition, decision making during context recognition and feature extraction from streams of incoming data.

1.1.1 Need for distributed service execution

Cyberforaging enables sharing of residual energy as well as CPU amongst devices in the network. In addition to offering the CPU as a service, each device also offers software as services for remote invocation and execution. Such distributed service executions increase the range of tasks that a user can perform. For example, Alice needs to convert a text file into a .mp3 audio format. Suppose, her cell phone does not have the corresponding converter, but John's phone has the required software. When Alice and John are within communication range, Alice's device can automatically transfer the file to John's device and invoke the text to speech file converter.

In the literature, multiple services have been combined to satisfy user's request. For example, Liang et al. [9] perform service composition for the following multimedia processing application. The authors consider a user who wishes to watch a movie on his PDA. The movie file is available at a remote device. Since the remote device cannot store the movie for all possible types and configurations of client devices, transcoding is required to convert the movie into a format suitable for the PDA. Also, the movie does not have subtitles and the user needs subtitles in a language different than that in the movie. Hence video processing is required to extract the sound track, convert it into a language as required by the user, convert the speech to text and insert the text into the movie. Thus even though the user's PDA does not have capabilities to perform the necessary transformations, multiple services available at multiple devices can be combined to satisfy the requirements.

Yang et al. [10] consider a *follow-me* application. The authors consider that a user is watching newscast on his home TV. The user then pauses the newscast and travels to the airport to pickup a friend. While traveling, the newscast is converted into a audio-only stream and is played using speakers in the user's car. When the user reaches the airport, flight information is displayed on devices in the car. When the user parks his car and walks to the terminal gate, the newscast and flight information are streamed to his PDA. Such an application requires resource sharing between devices to achieve the data transformations and context recognition.

1.1.2 Need for parallel processing

A pervasive system can be viewed as a distributed system composed of multiple devices that offer CPU cycles as a service to other devices. Multiple devices can be used to execute a task in parallel. Consider that a user wishes to perform a computationally heavy task such as audio or video processing on his PDA. Parallelism among nearby devices can be exploited to obtain speed up for completion of task execution. The PDA can query neighboring devices and

receive information about their computing capabilities. The input audio/video can then be split amongst the participating devices and processed in parallel.

Consider that Alice's device invokes a service on John's device. Such events as John switching off his device, or the device exhausting its residual energy, or the two devices moving out of each other's communication range result in a low success probability for completion of Alice's task. If the required service is provided by multiple devices, Alice can invoke the service on multiple devices in parallel. This replicated invocation and execution increases the probability of receiving final results from atleast one service provider.

Parallel processing is a well-studied area for high-end machines. Parallel algorithms and programming techniques for several fundamental problems like matrix multiplication, sorting, searching, image analysis and several others have been described in the literature. Even though device parallelism exists in abundance in pervasive systems, exploiting parallelism using traditional algorithms is nontrivial due to such unique challenges as heterogeneity, resource constraints, mobility, and privacy and security. In particular, sensor and embedded systems are expected to be long running - meaning that once deployed they are expected to operate for long periods without human intervention. In pervasive systems, exploiting parallelism may increase the energy consumption due to increased communications. Therefore, it is necessary to develop novel techniques that can simultaneously meet time and energy constraints.

1.2 Challenges in distributed computing over pervasive systems

1.2.1 User mobility

Traditional parallel algorithms do not consider the participating processors to be mobile. In examples described in the Section 1.1, mobile and personal devices are be used for computation. Since the devices are mobile, there will be constant changes in the underlying network. Any algorithm that assumes the network to be static cannot be used in such mobile pervasive

systems. Sonnek et al. [11] propose a centralized algorithm to handle the mobility challenge. The algorithm requires complete and exact knowledge of all devices in a network and hence is not be suitable for dynamic environments.

1.2.2 Random failures

Devices in pervasive environments might disconnect or fail at any time. There is no guarantee on the availability of those devices. Such random failures can be due to user turning off the device, device losing its network connectivity or device being non-responsive due to software error. Litke et al. [12] have proposed a centralized algorithm to model the reliability of each device using Weibull distribution.

1.2.3 Device heterogeneity

In a pervasive computing environment, the devices have varying CPU speeds and memory capacities. In the past, few algorithms such as [13] and [14] have been proposed that consider only CPU heterogeneity. Other constraints such as mobility and random failures have not been considered.

1.2.4 Energy

Traditional parallel algorithms assume that there is sufficient energy to complete the assigned tasks. Mobile devices in a pervasive environment have limited residual energy and might not be able to complete the tasks. Transmission and reception of a message consumes considerable amount of energy. Hence algorithms should be energy-aware and communication amongst devices should be reduced. Yu et al. [15] propose an energy-balanced allocation algorithm for homogeneous sensor nodes equipped with dynamic voltage scaling. The authors do not consider such challenges as mobility and random failures.

1.2.5 Interaction between user and devices

Since personal devices are used for the parallel computations, the tasks should not inhibit the user from performing his daily activities. For example, on a cellphone, an incoming phone call receives higher priority than the current parallel task. Hence it might not be possible to reserve the CPU for entire amount of time of the parallel task. Eagle et al. [16] have performed modeling of interaction between users and a few applications on their respective devices.

1.2.6 Security

In a pervasive environment, when a client is using neighboring devices for execution of his tasks, the client should be protected from malicious devices. Also, devices that perform tasks on behalf of the client should trust the job submitted by client. If needed, the data should be encrypted to ensure privacy of the data. Research efforts such as [17] and [18] have provided algorithms to enhance security and trust in pervasive systems.

1.3 Major research contributions

This dissertation provides algorithms that enable distributed service execution in smart environments, MANETs, VANETs and OPNETs.

1.3.1 Cluster Based Scheduling (CBS)

The CBS algorithm [19] is designed for service execution in smart environments. Requests for task executions are modeled as directed acyclic graphs (DAG) with services as nodes and dependencies amongst services modeled as links of the DAG. The devices are assumed to be grouped into logical clusters. Each cluster has a cluster head. A *client* is a device that wishes to schedule a set of tasks. Scheduling is performed in iterations. The client communicates only with cluster heads. Cluster heads perform scheduling of the DAG onto devices in their respective clusters. Scheduling a set of dependent tasks onto a network of devices is a

NP-complete problem. To avoid generating and evaluating all possible schedules, CBS adopts a dynamic programming approach. CPU capability of each device is considered as a knapsack. A subset of tasks is added to the knapsack depending on how much computation is performed, how much communication is avoided and how much communication cannot be avoided by the subset. Each cluster head sends information about subsets selected to be scheduled within the cluster. After receiving such *bids* from cluster heads, client selects the cluster that can perform most work in minimum time. The next iteration is started to schedule any unselected tasks. Iterations are performed till all of the tasks are scheduled.

The main advantages of CBS as compared to other schemes are: (*i*) Communication overhead is reduced by considering the devices to be grouped into logical clusters. (*ii*) Schedules are generated in a distributed fashion. CBS does not require the client to have knowledge of all characteristics of each device in the environment. (*iii*) CBS allows usage of multiple task scheduling algorithms. (*iv*) Simulation results demonstrate time and energy efficient scheduling of tasks in heterogeneous environments. To handle mobility, CBS requires that the exact arrival time and departure time of each device is known.

1.3.2 Distributed Grading Autonomous Selection (DGAS)

In MANETs and VANETs, it is difficult to know the exact arrival and departure time of all devices. Distributed decision making incurs large overhead. Applications in such environments should perform autonomous decision making. The Distributed Grading Autonomous Selection (DGAS) algorithm [20] is suitable for MANETs and VANETs. A number of mobile software agents are deployed in the network. Communication between mobile devices is performed using a backbone created out of the deployed agents. Since the devices are mobile, software agents also move from one geographical location to another. When a *client* needs to execute services hosted on other devices in the network, the client sends the request to the nearest agent. The agent transmits a request to devices that are within its communication range. Each device

autonomously decides whether it is resourceful enough to participate in service execution. Such factors as residual energy, mobility, time required for communication are used to decide whether a device is suitable to participate in service execution. It may happen that too many devices are suitable to execute the service. To control the number of replicas, each agent stores a probability value, Θ. This value is transmitted by the agent along with the service execution request. If a device is resourceful enough to execute the requested service, the device decides whether to execute the service with probability Θ. After executing the service, results are routed back to the agent. The agent varies Θ depending on the number of results received.

The main advantages of DGAS over existing schemes are: (*i*) Communication overhead is reduced since each device autonomously decides whether to participate in service execution. (*ii*) Each device maintains its own history and need not share its private information with other devices in the network. (*iii*) DGAS allows each device to use a different algorithm for computing its success probability in executing a service. (*iv*) Simulation results show savings in time due to lower communication overhead. Through simulations studies it is show that, despite the autonomy at device level, the number of replicas are effectively controlled by varying Θ.

1.3.3 Mutual Exclusion in Opportunistic Networks (MEOP)

Opportunistic networks are essentially distributed networks in which devices infrequently come within each others' communication range. Applications in such a distributed system might require exclusive access to shared resources. The portion of code that requires exclusive access to the resource is called as a *critical section*. The MEOP algorithm [21] enables exclusive access to shared resources in an opportunistic network. MEOP uses a timeout based fault detection algorithm. The novelty of the timeout algorithm is that the algorithm exploits the inter contact time distributions. MEOP is independent of the underlying routing protocols and hence does not need continuous monitoring of the entire network. Each device maintains a request pending queue. When an application wishes to access the shared resource, a request is gener-

ated and added to the pending queue. When two devices opportunistically come in contact, the pending queues are exchanged and updated. MEOP is a token based algorithm. An application executes its critical section only when the application possesses the token. Token requests are propogated over a logical DAG.

MEOP is free from starvation, deadlock and allows only one application to access the shared resource at a time. To the best of our knowledge, MEOP is the first to enable mutual exclusion in opportunistic networks. Unlike existing mutual exclusion algorithms for MANETs [22], the proposed algorithm does not require continuous monitoring of the network topology. Simulation results show that MEOP is communication efficient as compared to other algorithms proposed for generic MANETs.

1.3.4 Service composition in OPNETs

Consider that a user wishes to perform an application level service that is not available anywhere in the network. However, such a service can be composed by a series of basic services that are likely to be available on devices in the network. In service composition, multiple services are concatenated together in such a sequence that the first service in the chain takes input generated at the application and last service produces the required output. Service composition has been studied in detail for traditional networks [23] and MANETs [9], [10], [24]. This dissertation is the first to propose service composition in OPNETs. Two algorithms: Minimum Length Composition (MLC) and Minimum Delay Composition (MDC) are proposed [25]. Analysis for success probability of service composition and length of compositions is presented and verified. A middleware architecture for opportunistic computing is also proposed. The middleware is modular and deployable onto any device. An extremely resource constrained device, such as a sensor, hosts basic modules, whereas resource rich can implement all of the modules. Service execution over devices in OPNETs is challenged by device and communication failures. To overcome this challenge, middleware support is required to mask discontinuites. One solu-

tion is to perform fault tolerant service composition by generating multiple composite services. A prototype of the service composition process has been implemented over Bluetooth enabled devices carried by students in the campus of University of Texas at Arlington. Service execution requests are exchanged when the devices opportunistically come within communication range.

1.4 Organization of the dissertation

Chapter 2 discusses the related work available in the literature. Chapter 3 describes the CBS algorithm for task scheduling in smart pervasive spaces. Chapter 4 describes the DGAS algorithm for service execution in MANETs and VANETs. Chapter 5 presents the MEOP algorithm. Chapter 6 describes the service composition algorithms and a middleware for opportunistic networks. Chapter 7 provides concluding discussions and directions to future research.

CHAPTER 2

BACKGROUND AND RELATED WORK

Devices such as cell phones, PDAs, laptops that are available today have much more processing capabilities, battery life and other facilities than those a few years back. The device capabilities will continue to grow in the future. Along with the increase in capabilities, user expectations also have grown by large and there is a demand to have more applications installed onto the devices. Since the devices in a pervasive system are mobile and resource constrained, ad hoc networking concepts have been used to perform cooperative execution of applications. Networking amongst devices helps to increase the computation power accessible to small devices. Cooperative task execution also aims to conserve resources on resource constrained devices by utilizing devices that have sufficient resources. In this chapter, we review some of the efforts that address the problem of performing heavy computations over handheld, resource constrained mobile devices.

Section 2.1 describes research work where computationally heavy tasks are offloaded to high end servers that are immobile. Section 2.2 describes architectures where tasks are offloaded to mobile devices in Mobile Ad hoc Networks. There have been publications that report algorithms for routing and data dissemination in VANETs. Section 2.3 describes few of those algorithms. Algorithms for routing and information dissemination in opportunistic networks are described in Section 2.4. Section 2.5 describes algorithms that enable mutual exclusion in traditional networks and MANETs. Section 2.6 discusses algorithms that perform service composition in MANETs. Section 2.7 lists some of the algorithms that perform task scheduling in networks consisting of immobile devices. The chapter ends with a discussion on the need of algorithms proposed in this dissertation.

2.1 Offloading executions to high end servers

The concept of cyberforaging in pervasive systems was introduced by Balan et al. [26]. Cyberforaging is the process of discovering and using resourceful server(s) in order to offload computationally heavy tasks from mobile devices to the server(s). The resourceful servers are called as *surrogates*. This chapter provides a discussion on many of the architectures proposed for cyberforaging in various types of environments.

Su et al. [27] consider an environment where a mobile device enters a wireless hotspot, connects to the Internet and offloads tasks onto a server on the Internet. Since the server might become a bottleneck for interactive applications, the authors propose an architecture, named Slingshot, that creates a replica of the server onto devices co-located with the wireless access points. Since the transmission time between mobile device and the replica is less than that between the mobile device and server, usage of the replica provides performance improvement for interactive applications. The server and its replica are synchronized continuously. When a mobile device moves from one access point to another, the replica of current execution state is moved to the latter access point. The authors have targeted cyberforaging of applications that consist of continuous interaction between user and his device. Slingshot requires surrogates to be available at each access point. In a pervasive system, devices might offer and require services that are not available at the surrogates. Hence Slingshot would require each service to be available in a downloadable format. Process migration from one surrogate to another requires homogeneity in the execution environment.

Palmer et al. [28] propose a middleware named Ibis for performing application deployment over a grid. The main features of Ibis are: (*i*) an extensive API for data exchange. (*ii*) *Smart Sockets* that are used to create connections through firewalls installed at the grid, and (*iii*) middleware modules for grid monitoring, resource management, RMI and MPI. Ibis does not consider challenges described in Section 1.2 and hence is not suitable for pervasive peer-to-peer service sharing.

Kristensen [29] has proposed a middleware named Locusts. The main modules of Locusts are - presence daemon, scheduler, execution environment, resource monitor, context manager and Locusts API. Locusts decomposes task into a directed acyclic graph of services available in the network. The services are then scheduled onto surrogates such that there is no direct communication between services executing at different surrogates. When the user moves from one surrogate to another, the user receives intermediate results of executions from the former surrogate and transmits these to the later surrogate. In a pervasive system, surrogates may host different sets of services. Hence, depending on the set of services to be executed, there might be a need for communication between surrogates. Since Locusts schedules services such that there is no direct communication between surrogates, Locusts is not suitable for service execution in pervasive systems. Locusts assumes that a service can be executed at any of the surrogates. This assumption is not valid for devices in pervasive system.

Gnawali et al. [30] propose a two-tier architecture, called Tenet, for data fusion over a static sensor network. Resource constrained motes form the lower tier. Comparatively resourceful devices form the upper tier and are called as masters. Masters perform processing on the data received from the mote layer, while motes only sense the environment. The master sensors are used for cyberforaging of tasks on behalf of other sensors. Tenet has been developed only for static sensor networks. Membership of the tiers is not changed dynamically. In a pervasive system, devices may not be easily classifiable into master-worker layers, due to resource constraints. For example, some devices might have lower residual battery and an almost idle CPU, whereas some devices might have higher residual battery but a slower CPU. Hence the membership of a device into the master layer should be made dynamic.

Ma et al. [31] propose a middleware, named JESSICA, that supports parallel execution of multi-threaded Java applications in a cluster of computers. The degree of parallel execution is dependent on the number of threads created in that application. JESSICA was designed for a cluster of resourceful machines. Threads are automatically redistributed across cluster to

exploit parallelism. The node on which the application is started is called as a console. Other participating nodes are called as worker nodes. When the utilization of worker node is more than double that of the console, the worker is considered to be under heavy load. Threads from that worker are then migrated to the console. If an under-loaded worker is found, the received thread can be migrated to it. When a running thread wants to migrate from console to a worker, it is split into two cooperating entities, one running on the console, called the master, and other running on the worker, called the slave. The execution context is divided into machine dependent sub-context and machine independent sub-context. The master thread remaining at the console performs any location-dependent operations such as I/O for the slave. Exploiting thread-level parallelism requires device level support for thread migration. Hence distribution of threads may not be suitable for pervasive networks comprising of heterogeneous devices. JESSICA does not consider the energy consumption in the devices. JESSICA does not consider recovery solutions for sudden worker crash, which is rare in a resourceful cluster and is common in mobile and pervasive environments.

The main disadvantage of research work described in this section is the dependence on static and/or high end servers. In this dissertation, we consider pervasive environments in which such resourceful surrogates are not available.

2.2 Offloading executions to mobile devices

This section describes architectures where tasks are distributed to mobile devices in Mobile Ad hoc Networks. Execution of services on devices in MANETs is subject to failures due to resource constraints and mobility. Hence instead of executing a service on only one device, multiple devices execute the required service in parallel. This increases the probability of receiving results from at least one device.

Sonnek et al. [11] use reputation of devices to decide which devices will execute a particular service. When a mobile device wishes to perform a task, the device sends the request to a central server. The central server stores reliability information about each mobile device. The reliability of a device is computed as the ratio of number of correct responses to the total number of tasks attempted by the device. The central server groups mobile devices such that the combined reliability of atleast one group is sufficiently high. This work cannot be directly used in highly dynamic environments due to two reasons: first, a central server stores reliability of each device. Storing characteristics of each device in a large area such as a city, will have a huge footprint. Secondly, there are many factors that influence whether the service provider device will send the results within a certain time interval. The success of a replica depends not only on its resource levels or the whether the device is malicious, but also on its mobility. These factors have not been considered in [11].

Litke et al. [12] consider an environment similar to that in [11]. Instead of reputation of a device, the authors of [12] consider crash, omission and timing failures as factors that contribute to the reliability of mobile devices. The reliability of each device is modeled using a Weibull distribution. The authors assume that all devices have the same success probability distribution. This is a simple assumption, not valid in current dynamic environments replete with heterogeneous devices. The authors consider a central server which gathers information about mobile devices and computes the number of replicas required for a successful execution. Challenges described in Section 1.2 have not been considered.

Subramani et al. [32] consider a "bag of independent tasks" case. The architecture employs multiple schedulers that are distributed in the network. For each task, all the schedulers place that task onto the most lightly loaded devices. The schedulers act as local servers to gather information about other devices in the network. The devices in the network are assumed to be static and not prone to failures. This assumption is not valid for devices in pervasive environments. Since each scheduler executes every task on some device, there are always exactly the

same number of replicas of each task. In a mobile, ad hoc environment, different tasks at different times require variable number of replicas, making this scheme unsuitable for pervasive systems.

Alsalih et al. [33] propose a task scheduling algorithm for MANETs. The tasks and their interdependencies are represented as a directed acyclic graph. The task scheduling algorithm proposed in [33] is energy-aware and attempts to minimize the energy as well as makespan of the generated schedule. Makespan is defined as the total amount of time required to complete all tasks in the DAG. The algorithm first sorts tasks in descending order of their priority, where priority of a task is defined as the maximum length of path from the task to any of the final tasks. The tasks are then placed onto devices such that the cost is minimized. The MANET is considered to be composed of heterogeneous devices. Though the authors perform scheduling over a MANET, mobility of devices is not factored in the proposed algorithm. Yu et al [15] propose an energy-balanced allocation algorithm for homogeneous sensor nodes equipped with dynamic voltage scaling. All devices are assumed to be connected with each other and available till the execution of all tasks is over. [33] and [15] require atleast one device to gather information of all devices in the network and perform the scheduling. Thus, though the scheduler device might be resource constrained, it acts as a central server and incurs huge communication overhead.

Liu et al. [34] propose a distributed operating system layer, named MagnetOS, that makes the entire network appear to applications as a single virtual machine. It automatically and transparently partitions applications into components and dynamically places the components onto devices. MagnetOS makes the entire network operate as an extended Java virtual machine. The implementation of MagnetOS consists of a static application partitioning service that injects new code into the network, a process (on each node) that performs dynamic monitoring and component migration, and a set of policies to guide object placement at runtime. For static partitioning, granularity of partitioning is a Java object. Using an object as the unit of mobility helps in preserving existing object interfaces. The inter-object invocations become events

in MagnetOS. To find energy efficient placement of objects, the communication pattern between objects is observed. Objects that have higher interaction are kept closer to each other on neighboring devices or on the same device. Users have no control over the initial layout of components in the ad hoc network. The algorithm for finding energy-efficient placement of objects suffers from thrashing. If two objects use a third object in same access pattern, the third object will keep moving between the two nodes on which the two objects are placed. This results in excessive waste of energy. Hence MagnetOS is not suitable for pervasive systems.

Maghraoui et al. [35] demonstrate how to achieve process migration in applications that use Message Passing Interface (MPI). The authors propose Internet Operating System (IOS) that provides load balancing capabilities. MPI/IOS is a system that integrates IOS middleware strategies with existing MPI applications. MPI/IOS provides checkpointing and migration mechanisms. IOS middleware analyzes the underlying physical network resources, application communication patterns, data accesses and memory usage to decide how applications should be reconfigured to accomplish load balancing. When a node becomes underutilized, it generates work stealing requests. Each such request has the performance-related information about that node. Upon receiving such request, each node decides whether there are any processes that can be migrated. When a migration decision is taken, all the nodes are informed about the decision. This causes all processes to halt in the next iteration until the migration is complete. The migrating process then checkpoints its state and spawns an image of itself in the remote processor. This mechanism causes synchronization delay. Monitoring of communication patterns incurs huge overhead. Process migration requires homogeneity of execution environments across all devices. The assumption that any process can be migrated to and executed on any device is not applicable for pervasive environments, making MPI/IOS unsuitable for pervasive systems.

DynaMP [36] is a message passing architecture used for parallel computation in mobile systems. DynaMP uses all its neighbors in a Bluetooth piconet. Work is distributed evenly

amongst all the neighbors. The authors assume homogeneity in service execution capabilities of all the devices in the network. Challenges described in Section 1.2 are not handled.

2.3 Information sharing in vehicular ad hoc network

McNamara et al. [37] propose an algorithm for P2P file sharing amongst users traveling by the same train. Each device stores the average intercontact time along with the time at which the contact was made with other devices. When a user needs a file, his device filters out devices that do not have the required file and then sorts the remaining devices in descending order of the intercontact time. A file is transferred only if time required for the transfer is less than the average contact time between the two devices. The authors do not consider security and energy constraints. Fault tolerance and recovery is not addressed. The similarity between [37] and our work is the assumption that there is a pattern in which a person commutes daily.

Zhang et al. [38] propose a P2P content sharing scheme called *Roadcast* for VANETs. The scheme consists of two components - popularity aware content retrieval and popularity aware data replacement. A particular file or information can be stored (replicated) at multiple locations in the network. When a user requires some specific data, the user sends a query to all devices within its communication range. Popularity aware content retrieval module ranks the relevance of data to the queries by using popularity factor of the data. When a device receives a file, the file is stored onto the device and is made available to other devices in the network. The device might utilize all of its memory for storing files. In such cases, some of the files are removed. The popularity aware data replacement module decides which files are to be removed. Files are retained such that the number of instances of each file is proportional to square-root of their popularity. Thus Roadcast maintains more instances of frequently requested files. Roadcast requires each device to have complete knowledge of data and devices in the network. Content sharing in pervasive environments might benefit from using social networking

concepts. Roadcast does not implement any social networking heuristics. The authors do not discuss any fault tolerance issues in Roadcast.

Lee et al. have implemented a P2P file sharing software, named CarTorrent [39], onto devices in actual vehicles. The main modules of CarTorrent are - File Splitter, Gossip Thread, and File Manager. The file splitter module splits a file into multiple pieces that can be used by the torrent. Gossip threads send and receive information about file pieces available in the network. The file manager stores information about file pieces and devices. When a user needs a file, CarTorrent perform discovery of pieces and selects the best set of devices that can provide the file pieces. Device selection is performed by determining the rarest piece of a file and the closest node that has it. AODV is used for routing of packets from one vehicle to another. CarTorrent does not consider the speed, direction and mobility patterns of vehicles. These factor greatly affect the success of communication and data sharing. Simulations performed in [39] are extremely simple and do not reflect real world cases. The authors have not studied simulations using a city wide network.

Depending on the devices, their mobility and resource levels and application requirements, multiple middleware architectures might be needed to mask heterogeneity of VANETs. Delicato et al. [40] argue that a single middleware is not sufficient for all possible configurations and applications in VANETs. The authors propose a Software Product Line (SPL) approach to design a middleware family. Middlewares are generated depending on the variability of the VANET. SPL consists of two phases - Domain Engineering (DE) and Application Engineering (AE). Common features and utilities are modeled as mandatory components of a middleware. Application specific features are modeled as alternative or optional modules. In the DE phase, a software architect designs the SPL considering all possible middleware components and their connections. In the AE phase, module selection is performed to get multiple middlewares. Factors such as device type, location, mobility, sensing capability, role of the device in applications, security and fault tolerance are used to perform module selection.

2.4 Information sharing in opportunistic networks

Opportunistic networks have received considerable attention in recent years. Many challenging problems such as routing, modeling of social interaction have been addressed. Passarella et al. [4] propose an algorithm that performs service provisioning in opportunistic networks. A device might host multiple services that can be invoked by other devices in the network. When two devices come within communication range, they get an opportunity to initiate service execution on each other. A device that initiates execution on other devices is called as a seeker. The device which executes the required service is called as a provider. Results are returned back to the seeker only when the execution has completed and the two devices (seeker and provider) come within communication range. Since the devices meet each other opportunistically and the intercontact time is large, the seeker initiates execution of the required service on multiple providers. This reduces the expected time to receive the results back at the seeker. The authors provide analysis of the time delay and the number of executions that should be started. Since multiple devices might be utilizing services provided by a particular device, and a seeker might request execution of multiple services on the provider, the service execution is modeled as a $M^X/M/1$ queue.

Pantazopoulos et al. [41] use social networking concepts to place shared data efficiently in an opportunistic network. Devices in the network might access a shared data resource. The authors introduce the concept of *conditional betweenness centrality* which measures the cost of accessing data stored on a particular device from any other device that is interested in the data. Data is initially placed at a random location. The proposed algorithm computes shortest paths from all devices to the location of the data. Some of the devices might have multiple shortest paths passing through them. Top few percent of such devices are selected. For each of those device, the centrality value is computed. The device with the lowest cost of data access is selected to be the location of the data. This process is repeated till there is no more movement of the data. The authors do not consider such challenges as random failures, trust and reputation

of devices. Data is not replicated over multiple devices. Replication of data will decrease communication overhead and increase the availability in presence of device failures.

Ott and Kutscher [42] propose protocols for implementing HTTP over delay tolerant networks. Publications such as [5], [43] and [44] provide algorithms for routing in opportunistic networks. Hui et al. [5] groups users' devices into communities and sub-communities depending on interactions among the devices. The algorithm exploits peoples' social behavior by using the following heuristic: since people have different roles and popularity, their devices' also would have been in communication range of different number of devices. Each device is associated with one or more communities. Degree centrality of a particular device is defined as the number of devices that have been within communication range of that particular device. Forwarding is performed based on the communities and popularity, exploiting the opportunistic connections among devices. Devices that have higher value of degree centrality are more frequently chosen as the data forwarders.

Boldrini et al [43] perform context based routing in opportunistic networks. The authors provide a framework, named HiBOp, for managing and factoring context into data forwarding decisions. HiBOp exploits the current context as well as recorded (history) context. HiBOp provides fault tolerance by transmitting multiple copies of the data. The sender is allowed to create multiple copies. Intermediate devices perform only forwarding. When a device wishes to send data to another device, it broadcasts not only the data but also information about destination device. Intermediate devices that receive this broadcast evaluate the probability of delivery and propagate the data to other devices that have higher delivery probability.

Costa et al. [45] propose SocialCast which is a routing algorithm for applications that have publish-subscribe method of data exchange. The algorithm exploits information about users' interest in certain data, social ties of people and attempts to predict users' movements. The authors assume that users' with common interests meet more frequently with each other than with other users. To perform the prediction, the authors use Kalman filters and use pre-

viously observed movements as input to the filters. The predictions are useful in deciding which hosts will be better message forwarders. Devices that have a higher probability of being co-located with the destination are chosen to be the data forwarders. To avoid flooding of messages in the network, each message has a time-to-live based on hop counts. To provide fault tolerance, the sender generates and transmits multiple copies of the data.

Spyropoulos et al [46] propose two algorithms: (*i*) spray and wait, and (*ii*) spray and focus, for routing of data in opportunistic networks. In the spray and wait algorithm, the sender creates a fixed number of copies of the message. When the sender comes within communication range of another device, the sender sends one copy of the message to the device. The device that receives the message stores the message and forwards it to the destination upon direct contact with the destination. That is, the intermediate device does not route the message to any other device other than the destination. Hence message is delivered in atmost two hops. It might happen that due to restricted mobility or localized mobility pattern, all the copies might remain in the vicinity of the sender and the message might never reaches the destination. In the spray and focus algorithm, upon receiving the message copy, an intermediate node uses utility based forwarding heuristics to forward the data towards the destination. Hence the probability that the message is never delivered to the destination is reduced.

Wang et al. [44] use erasure coding for data forwarding in opportunistic networks. Every message to be transmitted, is converted into multiple blocks of data such that even if only a few blocks are received by the destination, the original message can be reconstructed using the received blocks. Abdulla et al. [47] perform modeling of user mobility. The authors present a detailed simulation study of intercontact times under Random Waypoint and Random Direction mobility models. Though many past research works have proposed middlewares for particular problems, there have been no publications investigating middleware support for opportunistic networks.

Service execution algorithms proposed in this dissertation are independent of routing algorithms. Any of the routing algorithms described above can used for routing of data in the proposed algorithms.

2.5 Mutual exclusion (M.E.)

Mutual exclusion (M.E.) is one of the fundamental challenges in distributed systems. Consider that multiple devices share a resource. Need for mutual exclusion arises when two or more nodes wish to perform (or enter) their *critical section (CS)*. To execute critical section, a device requires exclusive access to the shared resource. Examples of problems that require mutual exclusion are: assigning unique values such as IDs and IP addresses to mobile nodes in dynamic networks [48], ensuring total ordering of message delivery in mobile networks [49] and updating a shared file in a peer to peer environment. Any algorithm that provides mutual exclusion should satisfy the following correctness properties [50]:

1. Safety: *At most one node must be executing its critical section at any given time.*
2. Liveness: Liveness implies freedom from starvation and deadlock. Starvation is defined as: *any node that wishes to execute its CS, will eventually do so*. Deadlock is defined as: *if there is any node that wishes to execute its CS, then some node will eventually enter its own CS*.

Ordering is a desirable but not mandatory mutual exclusion property. Ordering requires that if one node requested access to CS before another node, the access is granted in that order. Very limited amount of work has been done to provide mutual exclusion in generic mobile ad hoc networks (MANETs). Existing algorithms for mutual exclusion can be classified as permission based and token based.

2.5.1 Token based exclusion

In token based algorithms, a token is circulated in the network. A node can enter its CS only if it holds the token. Token based algorithms can be further classified into token passing and token requesting algorithms. In such token passing algorithms as [51, 52], the nodes are arranged into a static or dynamic ring and the token is passed to all the nodes on that ring. In token requesting algorithms such as [22], a directed acyclic graph (DAG) is constructed such that each node points towards its neighbor through which the token holder can be reached. When the token is transferred from one node to another, the pointers are updated accordingly. For a static network, the DAG can be updated easily without much overhead. When a node wishes to enter CS, it sends a request message to its neighbor in the direction of the token holding node. The message is thus propagated to the token holder via multiple hops. Once the token holder exits its CS, the token is sent to the requester using the reverse path of the request message.

2.5.1.1 Token passing algorithms

In *Baldoni-Virgilito (BV)* algorithm [51], the token is passed over a dynamically created logical ring. Each round over the ring has a coordinator. Initially, the coordinator of the first round holds the token. When a node wishes to enter its CS, it sends a request to the coordinator, thus initiating token circulation. At the beginning of every round, the role of coordinator is transferred to the next node in the ring. Hence in each round, ID of the next coordinator is passed along with the token. Thus token requests need not be passed along the ring and can be passed using any routing protocol. When a node exits its CS, the node passes the token to a node that has not been visited by the token and is at a minimum number of hops from the token holder. The algorithm avoids starvation by passing the token to all the nodes by creating a ring structure. The authors make the following assumptions: (*i*) Nodes do not fail and the network is not subject to permanent partitions. Hence the algorithm can correctly avoid starvation and

deadlock only in the absence of node failures. (*ii*) A protocol that can provide such metrics as number of hops between two nodes is available. (*iii*) Topology does not change during the multi hop transmission of token. These assumptions are not valid for opportunistic networks. The token is circulated over the entire ring even if only a single node wishes to enter the CS. Hence the BV algorithm has large overhead if the request generation rate is low or only a few nodes generate requests. Hence the BV algorithm is not suitable for opportunistic networks.

Wu et al. [52] propose an algorithm, Mobile Dual-Token MUTEX (MDTM), for detection of token loss. The algorithm detects token loss due to communication failure only. The authors assume that the nodes do not fail. Two tokens are circulated in the network. Each token has its own dynamic ring and a coordinator. Only one token, called as primary token, gives permission to enter CS. The two tokens monitor presence of each other. When a token visits a node, the token records its ID onto the node. If a token completes a cycle over the ring and does not find the ID of other token on any of the nodes, loss of the latter is declared and the lost token is regenerated. Since MDTM is a token ring algorithm, it deadlocks if the ring is broken due to node failures. Hence MDTM algorithm is not suitable for opportunistic networks.

Malpani et al. [49] propose six algorithms that continually circulate the token over all nodes in a ring structure. In the *Local-Recency (LR)* and *Local-Frequency (LF)* algorithms, the token holder forwards token only to its neighbors. If there are multiple neighbors, the *LR* algorithm chooses the node that was least recently visited by the token, whereas the *LF* algorithm chooses the node that was least frequently visited. In *Global-Recency (GR)* algorithm, token is forwarded to any node in the network that was least recently visited. This node may not necessarily be the neighbor of the token holder. If the token is passed using multihop communication, the intermediate nodes do not enter their CS. The *Global-Recency with Next (GRN)* algorithm is similar to the *GR* algorithm except that when the token is transferred using multihop communication, intermediate nodes can enter their CS. In *Global-Frequency (GF)* algorithm, the token is passed to any node in the network that was least frequently visited by

the token. If multihop communication is used, intermediate nodes do not enter their CS. The *Global-Frequency with Next (GFN)* algorithm is similar to *GF* except that the intermediate nodes can enter their CS. Simulation results in [49] show that the *LR* algorithm performs better than other five algorithms. Starvation and deadlock are avoided only in the absence of failures, making the algorithms unsuitable for opportunistic networks.

2.5.1.2 Token requesting algorithms

In *Derhad-Badache (DB)* algorithm [22], a DAG is created for token requesting and forwarding. Edges in the DAG represent the route to be used by token requests and the token. When a token holder wishes to release the token, it chooses its successor such that (*i*) there is a path from token holder to successor, (*ii*) the successor has been waiting for longest period of time and (*iii*) routing cost between token holder and its successor is minimum. To obtain the routing cost, the algorithm requires a network topology monitor that can continuously update information about current network topology. The algorithm assumes that the network does not get partitioned permanently. To avoid starvation, algorithm requires each requester node to connect to token holder's partition for a time duration of atleast $T_{CS} + T_{discovery} + T_{Token}$, where T_{CS}, $T_{discovery}$ and T_{Token} are execution time of CS, route discovery delay and time required to deliver token from holder to requestor respectively. Due to node mobility, this assumption is not valid in an opportunistic network. Since the topology changes frequently, the DAG needs to be updated continuously, even if the token is not transferred between nodes. This updation has a huge overhead. Hence the DB algorithm is not suitable for opportunistic networks.

2.5.2 Permission based exclusion

In permission based algorithms such as [53] and [54], a node that wishes to enter CS sends messages to a set of nodes. If all of the nodes in that set grant permission, the requesting node enters CS. The algorithms assume that the nodes do not fail and the network topology is

assumed to be static. These algorithms do not tolerate loss of messages. In an opportunistic network, since any two nodes rarely come in contact, seeking permission from a set of nodes incurs heavy communication overhead and a large delay.

Wu et al. [55] propose a permission based algorithm for MANETs. Each node maintains an *info_set* and a *status_set*. When a node wishes to enter CS, it sends request to all the nodes in the info_set. A node receives request messages from nodes in its status_set. When a node is executing its CS and receives a request, the requestor node is added to a pending queue. After completing its CS, the node sends permission to all the nodes in the pending queue. If a node receives a request and does not need itself to enter CS (or wishes to enter CS with a lower priority), the node sends permission to the requestor and moves the requestor node to the info_set. Priority of a request is computed using the time at which the request was generated. Requests that were generated earlier receive higher priority. Consider that node n_a sends permission to node n_b. After receiving the permission, n_b moves n_a to the status_set. Since a node requires permissions from multiple nodes in the network, [55] has a huge communication overhead. When a node wishes to disconnect from the network, the node sends messages such as *DISCONNECT* or *DOZE*. An opportunistic network is partitioned for almost the entire lifetime. Hence disconnection messages are not guaranteed to reach other nodes in the network. Thus algorithm proposed in [55] is not entirely suitable for opportunistic networks.

Table 2.1 summarizes the above described algorithms. Simulation results in [49] show that the *LR* algorithm performs better than other five algorithms proposed in [49]. Hence the following table includes only *LR*.

2.6 Service composition in pervasive systems

Kalasapur et al [56] propose a service composition protocol for pervasive environments with underlying MANETs. Each device is assigned a level according to the device type. When

Table 2.1. Comparison between mutual exclusion algorithms applied to OPNETs

	BV [51]	LR [49]	MDTM [52]	DB [22]	[53]	[55]
Permission Based					✓	✓
Token Ring	✓	✓	✓			
DAG Based				✓		
Handles Link Failures			✓			
Handles Node Crash				✓		
Handles Node Mobility	✓	✓	✓	✓		✓

two devices come within communication range, they perform handshake to create a parent-child relation between them. The parent device receives information about services hosted on the child. When a device needs a particular service, it first checks if the request can be satisfied by itself. If not, the request is sent to the parent device. This process is continued till the best device or a composition is found. Information about services hosted on devices is combined into a two-level graph. In an opportunistic network, devices are disconnected for most of the time. Due to lack of hierarchy, as imposed by the protocol described in [56], such service composition scheme is not suitable for opportunistic networks.

Chakraborty et al [57] propose a middleware for service composition in semi-structured MANETs. Device on which the composition request is generated is called as *Request Source (RS)*. Each request consists of a sequence of high level tasks. RS evaluates other devices for the role of *Composition Manager (CM)*. CM performs service discovery, integration and execution of the services requested. If the CM does not have knowledge of a required service, a controlled flooding is performed to find the service. If the service is still not found, the service composition fails. The difference between [57] and service composition presented in this dissertation is that, if a service is not found, we try to chain services such that a composite service is created for the required service.

Service composition middleware model (SCM) [58], consists of 4 components - Translator, Generator, Evaluator and Builder. The translator reads the user request. The generator creates all possible service composition paths. The evaluator chooses the best path that satisfies given QoS constraints. The builder executes the selected composition path. Ibrahim et al. [58] also present a survey of existing service composition algorithms and map the components of the surveyed algorithms to components of SCM. Service composition in opportunistic networks is influenced by such factors as routing, privacy, context and resources available with each device. Such factors are not considered in SCM.

2.7 Task scheduling algorithms

Dhodhi et al. [14] use a problem-space genetic algorithm for task scheduling on a heterogeneous set of processors. An initial schedule is obtained by sorting tasks according to their B-levels [59]. Selection, crossover and mutations are performed to obtain next generation schedules. Schedules that reduce the makespan of the schedule are retained and used to generate new schedules. The authors do not consider mobility, energy constraints while generating schedules. All the devices are assumed to be available for infinite amount of time. Phan et al. [60] consider mobile devices for grid computing. However, all of the challenges presented by a pervasive computing environment are not handled. In [61], the authors perform task scheduling over a heterogeneous collection of homogeneous clusters. Devices in a cluster are similar in their characteristics. Devices in different clusters might be different.

Scheduling tasks on many devices increases parallelism, but might also increase the communication cost. Scheduling tasks on fewer devices or just a single device decreases the communication cost but increases the time taken to finish all the tasks. Kim et al. [62] exploit this observation by proposing an iterative algorithm that uses two methods: push and pull. Push distributes tasks onto multiple devices whereas pull brings tasks onto the same device. The au-

thors use Heterogeneous Earliest Finish Time (HEFT) [13] to obtain a schedule that is used as a base or the first schedule. The push and pull methods are executed alternately to obtain variations in the base schedule. Schedules that are worse than the previously obtained schedules are discarded and the better schedules are used for future iterations. The push-pull cycle is repeated till no improvement is seen for a fixed number of cycles. The push method aims at distributing independent tasks onto multiple devices. The pull method checks if tasks on the critical path use more than one device, if yes, the pull method tries to put the tasks onto a single device. The authors require a central server to perform scheduling. The central server gathers information about all devices in the network and has exact knowledge of the device characteristics at every instance of time. Hence such a centralized algorithm is not suitable for pervasive environments.

Chatterjee et al. [63] propose a distributed clustering algorithm for MANETs. The algorithm finds a dominating set of devices such that each device in the network is within communication range of atleast one device in the dominating set. Devices in the dominating set act as cluster heads. Clusters are created such that each cluster head is connected to equal number of other devices. Such factors as mobility, battery power, transmission power are considered during clustering.

2.8 Discussion on related work

Section 2.1 described architectures and algorithms research works where computationally heavy tasks are offloaded to high end servers that are immobile. Such resourceful and static servers are not available in every mobile pervasive network. In this dissertation, we consider networks where the resourceful and static servers are absent. Algorithms such as [27] require all mobile devices to have connectivity to the Internet. This requirement also is unrealistic in the environments such as MANETs, VANETs and opportunistic networks. Efforts such as [34] and [31] propose algorithms that achieve parallelism and concurrency by dividing the application at

the level of objects and threads. In this dissertation, we exploit the parallelism amongst services hosted by devices. A device may further divide a hosted service into threads and objects to create parallel execution flows within the service.

Though algorithms described in Section 2.2 offload tasks to mobile devices, these algorithms use central servers for decision making. The research work presented in this dissertation focuses on environments that are devoid of such central servers and environments where iterative communication with a central server has a huge overhead. There have been publications that report algorithms for routing and data dissemination in VANETs. Section 2.3 describes few of those algorithms. For opportunistic networks, the research community is mainly focused on such challenges as routing and information dissemination. Few of such algorithms are described in Section 2.4. Service execution over opportunistic networks has not been analyzed in great detail. Distributed applications might require mutually exclusive access to shared resources. Though distributed mutual exclusion has been studied for traditional networks and MANETs, it has not been studied with challenges posed by opportunistic networks. Section 2.5 describes algorithms that enable mutual exclusion in traditional networks and MANETs. In this dissertation, we propose an algorithm for facilitating mutual exclusion in opportunistic networks. In the past, several algorithms have been proposed for service composition in MANETs. Section 2.6 discusses few of those algorithms. In this dissertation, we present a middleware that is suitable for service executions in opportunistic networks. We study the process of service composition under conditions where devices infrequently come within each others' communication range. Section 2.7 lists some of the algorithms that perform task scheduling in networks consisting of immobile devices. These algorithms do not factor the challenges presented by pervasive systems.

CHAPTER 3

DISTRIBUTED SERVICE EXECUTION IN SMART ENVIRONMENTS

This chapter describes the CBS algorithm for task scheduling in smart pervasive environments such as intelligent homes or classrooms. When a user wishes to perform some tasks, a request is generated on his device. The request is considered to be in form of a Directed Acyclic Graph (DAG), where tasks (or services) that are to be executed are denoted as nodes of the DAG. The interdependencies between the tasks are represented as edges in the DAG. The network of available devices also are modeled as another DAG, where each device is represented by a node in the DAG and connections between the devices are represented as edges. Due to mobility of devices, each device is available for a certain period of time. Hence the services hosted by the devices also are available for during corresponding time windows. CBS performs task scheduling under the constraints of mobility and device heterogeneity.

3.1 Network model

The proposed scheme requires devices to be grouped into logical clusters. Creation and maintenance of a cluster is provided by the middleware. Each cluster has a contact point or a cluster head that performs all housekeeping jobs for the cluster [24]. For the sake of simplicity, we assume a single cluster head per cluster. The cluster head can be distributed in space and/or time in order to address issues of load balance and fault tolerance. The creation of clusters ensures smooth execution of tasks in mobile and uncertain environments, where availability of a single node cannot be guaranteed for long periods of time.

Each device d_i is represented by the tuple $(\mu_i, tda_i, tdu_i, tdm_i, l_i)$ where μ_i: CPU speed of the device, tda_i: Time at which the device is available, tdu_i: Time at which the device

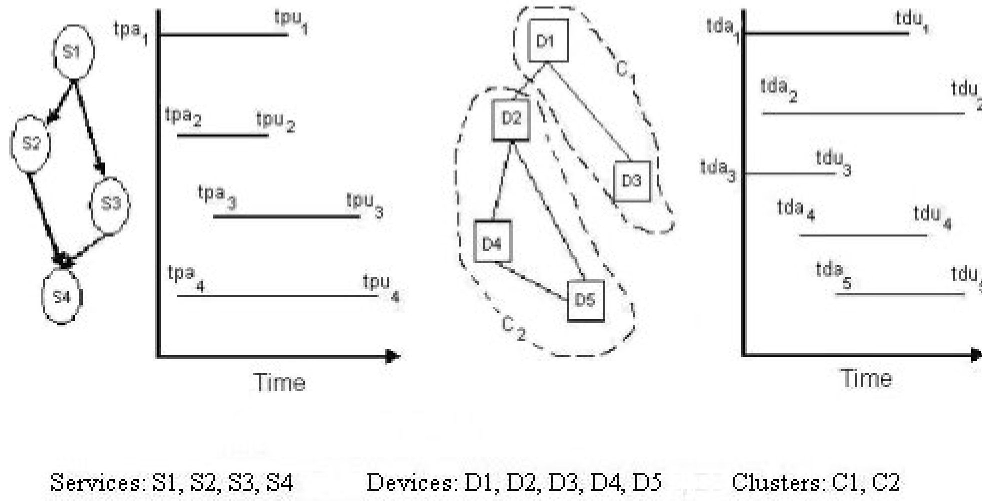


Figure 3.1. Example task graph, device connectivity, availability of services and devices.

becomes unavailable, tdm_i: Time at which the device finished its previous task. Hence the device is available in the time interval $[tda_i, tdu_i]$, and $tda_i \leq tdm_i \leq tdu_i$. It is assumed that these values are known. l_i is the number of different voltage levels supported by the device for dynamic voltage scaling. Devices are grouped together into logical clusters based on some criteria. For example, the grouping would be on connectivity or services offered by the device or resource level of each device. Let C_x be a cluster of devices and represented as $\{d_1, d_2, ...\}$. Let $\mathbb{C} = \{C_1, C_2, ...\}$ be the set of all clusters.

3.2 Task model

Let $G = (P, E)$ be the directed task graph with $P = \{p_1, p_2, ...\}$ as the set of nodes or tasks and $e_{i,j} \epsilon E$ is an edge incident from p_i onto p_j. Each task p_j is represented by the tuple $(c_j, tpa_j, tpu_j, tps_j, tpf_j)$, where c_j: cpu requirement, tpa_j: time at which task p_j becomes available, tpu_j: time at which task p_j becomes unavailable, tps_j: time at which task p_j will start executing, tpf_j: time at which task p_j will stop executing. Let $r_{i,j}$ be the time required to pass data between the two tasks. When the two tasks are scheduled on same device, $r_{i,j} = 0$, oth-

erwise $r_{i,j} \neq 0$. Let T be the deadline by which the entire task graph is to be executed. tpa_j and tpu_j are used to handle the service availability challenge. The pervasive environment might be able to provide these values. Figure 3.1 shows a sample task graph, device graph and availability windows of each service and device. The scheduling is performed in two levels: client - cluster communication and intra-cluster scheduling.

3.3 Client - cluster communication

A device that initiates the task scheduling is called as the *client*. Algorithm 1 gives the communication protocol between the client and the cluster heads.

Algorithm 1 Client-Cluster communication

1: $\mathbb{T} = \{T/|\mathbb{C}|, 2T/|\mathbb{C}|, 3T/|\mathbb{C}|, ..., T\}$
2: **for all** t in $\mathbb{T}$ **do**
3: send (winner of previous slot, request for bids for current slot) to all cluster heads.
4: receive (bids from cluster heads)
5: **end for**

The client divides the deadline, T, into $|\mathbb{C}|$ equal intermediate deadlines. Let $\mathbb{T}$ be the set of these deadlines. Hence, $\mathbb{T} = \{T/|\mathbb{C}|, 2T/|\mathbb{C}|, 3T/|\mathbb{C}|, ..., T\}$. The client informs all the clusters of the task graph and the intermediate deadlines. Each cluster head performs task scheduling using devices in that cluster only. Depending on the resources available within the cluster, the cluster head uses a suitable task scheduling algorithm. Greedy and list based algorithms require less CPU and energy to generate a schedule whereas genetic algorithms can generate better quality schedules in more time.

Each cluster returns the number of tasks that can be completed by the intermediate deadline and the start time and finish time of the tasks selected in a particular time slot. This is called as a *bid*. The cluster that performs the maximum number of tasks by the intermediate deadline is declared as the winner of that time slot. If two clusters select same number of tasks, then

the cluster that will execute the tasks in the least time is selected. If multiple clusters finish the same number of task at the same time, the tie is broken randomly. The result along with the start and finish time of the selected tasks is then sent to all clusters and the bidding process for the next time slot begins. Hence the number of messages received (rx) and transmitted (tx) by the client are: $rx = |\mathbb{C}|^2$, $tx = rx + 1$ respectively.

3.4 Intra-cluster scheduling

Due to the user mobility patterns and energy remaining on each device, each device will have a time window during which it is available for parallel computing. We divide the intermediate deadline provided by the client into more intervals depending on the arrival and departure time of the devices. The tasks are sorted on their bottom level (b-level) [59] to get the preferred sequence in which the tasks should be scheduled. Algorithm 2 describes an intra-cluster scheduling algorithm. Let τ_a be the set of device available times, and τ_u be the set of device unavailable times. Hence t is the sorted sequence of events of device becoming available or unavailable within a particular cluster, where t_v is the vth element of t and $v < w \Leftrightarrow t_v \leq t_w$. Hence each device is either available during the entire interval or is unavailable during the entire interval. $\mathbb{D}$ contains devices that can be used till time t_v. Step 6 calculates ϱ, the total computing power available till time t_v. Since each iteration performs scheduling in only one interval, we can reduce the task graph such that tasks that require more computing power than ϱ can be ignored. Let $\mathbb{P}$ be the set of unscheduled tasks such that sum of CPU requirements of tasks in $\mathbb{P}$ is less than or equal to $\varrho.\lambda$, where λ is the selection factor. Thus $\mathbb{P}$ contains tasks that could be scheduled till time t_v. Since the tasks are sorted according to their distance from the exit node, λ controls the number of edges that have both ends in $\mathbb{P}$. If the value of λ is small, we might get only disconnected tasks in $\mathbb{P}$. Scheduling both ends of the edges onto same device reduces the communication cost. Figure 3.2 shows an example of the effect of values of λ. Let ρ^j

Algorithm 2 Intracluster scheduling

1: $\tau_a = \{tda_1, tda_2, ...\}$ and $\tau_u = \{tdu_1, tdu_2, ...\}$
2: $t = \{\tau_a \cup \tau_u\}$
3: **for all** $t_v \le \mathbb{T}_h$ **do**
4: $\mathbb{D} \subseteq C_x$ such that $d_i \in \mathbb{D} \Leftrightarrow tdm_i < t_v$
5: sort $\mathbb{D}$ in descending order of μ_i
6: $\varrho = \Sigma_{d_i \in \mathbb{D}}[\mu_i(t_v - tdm_i)]$
7: $\mathbb{P} \subseteq P$ such that $\Sigma_{p_j \in \mathbb{P}} c_j \le \varrho.\lambda$
8: $\rho^j \subset \mathbb{P}$ such that $\rho_k^j \in \rho^j \Rightarrow \rho_k^j$ is an unscheduled parent of p_j
9: **for all** $d_i \in \mathbb{D}$ **do**
10: $\psi \subseteq \mathbb{P}$ where $\psi_j \in \psi \Rightarrow (\Sigma_{p_k \in \rho^j} c_k) + c_j \le [\min(tpu_j, t_v) - \max(tpa_j, tdm_i)].\mu_i$
11: Consider the device to be a knapsack of capacity $\mathbb{r}$. Form a table of $\mathbb{c} = |\psi|$ columns and $\mathbb{r}$ rows
12: Solve knapsack using recurrence relation: $[\mathbb{r}, \mathbb{c}] = \max\{[\mathbb{r}, \mathbb{c}-1], ([\mathbb{r}-\psi_j, \mathbb{c}-1] + \psi_j$ if $\rho^j \in [\mathbb{r}-\psi_j, \mathbb{c}-1] \bigwedge \max_{p_k \in \rho^j}(tpf_k + r_{k,j}) + c_j/\mu_i \le \min(tpu_j, t_v))\}$
13: $\phi_{r,c} = (\Sigma_{e_{k,j} \in Es_{r,c}}(r_{k,j}) + \Sigma_{p_j \in [\mathbb{r},\mathbb{c}]}(c_j))/\Sigma_{e_{k,j} \in Ec_{r,c}}(r_{k,j})$
14: Choose $[\mathbb{r}, \mathbb{c}]$ with highest rank. Schedule corresponding tasks onto d_i
15: Remove those tasks from $P, \mathbb{P}$
16: Update tps_j, tpf_j of the tasks and tdm_i of the device
17: Send the tps_j, tpf_j values to the client
18: **end for**
19: **end for**

be the set of unscheduled parents of p_j. Steps 10 to 18 are repeated for each device available till time t_v. A task might be scheduled onto d_i if the sum of the CPU requirement of the task and its unscheduled parents is less than or equal to the computation power available with the device in the interval $[tdm_i, t_v]$. Let ψ be the set of such tasks and ψ_j be the jth element in ψ. Hence $\psi_j \in \psi \Rightarrow (\Sigma_{p_k \in \rho^j} c_k) + c_j \le [\min(tpu_j, t_v) - \max(tpa_j, tdm_i)].\mu_i$. To select tasks for d_i, the computation power offered by the device till t_v is considered as the capacity of the knapsack. Hence capacity of the knapsack is $(t_v - tdm_i)\mu_i$. A dynamic programming table for the knapsack problem gives better results when the rows of the table are numbered consecutively, starting from 0. We create a dynamic programming table of $\mathbb{c} = |\psi|$ columns and $\mathbb{r}$ rows. For simulation purpose we considered $\mathbb{r} = 10$. Since the capacity of knapsack is $(t_v - tdm_i)\mu_i$, each row in the table will correspond to a multiple of $((t_v - tdm_i)\mu_i)/\mathbb{r}$. Since the knapsack capacity

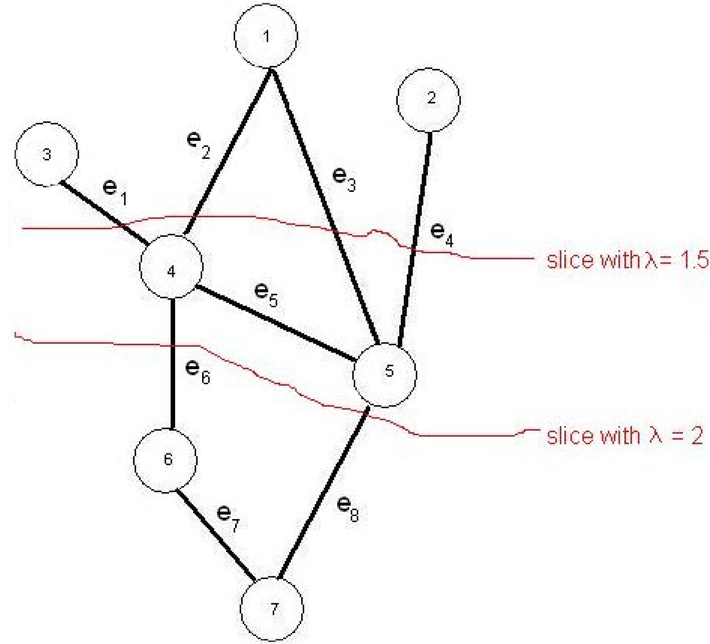


Figure 3.2. Example: Selection of tasks into $\mathbb{P}$ based on value of λ.

has been converted into $\mathbb{r}$, the CPU requirement of each task in ψ should be converted in the corresponding ratio. Hence divide $c_j \in \psi$ by $((t_v - tdm_i)\mu_i)/\mathbb{r}$. Step 12 gives the recurrence relation for the dynamic programming table. Let $Ec_{r,c}$ be the cutset of the combination $[\mathbb{r}, \mathbb{c}]$. Let $Es_{r,c}$ be the edges that have both ends in $[\mathbb{r}, \mathbb{c}]$. The rank of combination of tasks in $[\mathbb{r}, \mathbb{c}]$ is given by

$$\phi_{r,c} = \frac{\sum_{e_{k,j} \in Es_{r,c}} (r_{k,j}) + \sum_{p_j \in [\mathbb{r},\mathbb{c}]} (c_j)}{\sum_{e_{k,j} \in Ec_{r,c}} (r_{k,j})} \quad . \tag{3.1}$$

The combination with highest rank is scheduled onto device d_i. The tps_j, tpf_j and the number of tasks in the selected combination are sent to the client.

Consider the example task graph of Figure 3.2. Let $\lambda = 2$, $\varrho = 8$, $c_j = j$ and $e_j = j$, that is, $c_1 = 1$, $c_2 = 2$, $c_3 = 3$, $\ldots$, and $e_1 = 1$, $e_2 = 2$, $e_3 = 3$, $\ldots$. Hence rank of combination $(1, 2, 3) = 0.6$ and rank of combination $(1, 3, 4) = 0.785$. Thus varying the value of λ might give us better combinations. Consider that a schedule is obtained using the above method. For

each device, the sequence of tasks assigned to the device is checked for any idle time between two tasks. If an idle time is found and the corresponding device supports dynamic voltage scaling (DVS), the previous task may be executed on a lower voltage such that the idle time is minimized. A task is not considered for DVS if its output is sent to another task on a different device. Consider an idle time between tasks p_i and p_j. Let the device support k different voltage levels. The following steps are repeated till the idle time is minimized:

1. Let τ_k be the total time required to execute p_i at voltage level, k.
2. If $\tau_k \leq (tps_j - tps_i)$, decrement k by 1. Else execute p_i using voltage level $k + 1$.

Let E_j be the energy savings obtained by executing task p_j on a lower voltage level. The total energy savings for a schedule is $\sum_j E_j$.

In Algorithm 1, consider that T is not divided into slots. Hence bidding will be done only once and the obtained schedule consists of all tasks mapped onto devices in a single cluster. Depending on the device characteristics, a better solution might be obtained if the tasks are scheduled across multiple clusters. Hence instead of having just one slot, we divide T into multiple slots. Since bidding is performed for every slot, the communication overhead increases with increasing number of slots. Hence we use $|\mathbb{C}|$ number of slots as a tradeoff. Algorithm 2 is executed once for each slot of Algorithm 1. Hence Algorithm 2 is executed $|\mathbb{C}|$ times. Let there be n devices and m tasks. Algorithm 2 sorts the arrival and departure time of devices in ascending. Devices that are available in each interval are found and the innermost loop is repeated for each of that device. We get maximum number of intervals when only 1 device arrives or departs at an instance. Hence if there are x devices in a cluster, the innermost loop is executed x^2 times for the worst case. The inner loop creates a dynamic programming table of size y rows $*|\psi|$ columns. The worst case will be when $\psi = \mathbb{P}$, that is $|\psi| = m$. Hence it will take $2ym$ computations for each iteration of the inner loop. After a schedule is obtained from algorithm 2, each task might be executed on a lower voltage level. Hence it takes $\sum_j l_{\delta(j)}$ computations, where $\delta(j)$ is the device on which p_j is scheduled. Let $l_{max} = \max_i(l_i)$. Hence

worst case complexity of the algorithm is $O(y.m.|\mathbb{C}|.x^2 + m.l_{max})$. The client transmits and receives a total number of $2|\mathbb{C}|^2 + 1$ messages. Thus we see that if keeping n as a constant, if we increase $|\mathbb{C}|$, the communication cost increases and the computation cost decreases. The total number of messages sent by all devices in a work stealing algorithm like [35] is given by $(n^2).f$, where f is the frequency of advertisement messages sent. Thus we see that even though the computation complexity is higher in our algorithm, the energy savings obtained due to reduced communication will be much higher and the higher computation complexity is worth the energy savings.

3.5 Simulations

We compare CBS with the HEFT algorithm [13] used over all devices. To evaluate the communication overhead and energy consumption, we compare CBS with algorithms given in [1] and [35]. The simulations were performed for random task graphs and randomly generated device characteristics and connectivity.

3.5.1 Comparison metrics

- Normalized Schedule Length (NSL): NSL is defined as the makespan divided by the minimum time required to execute the critical path. $NSL \geq 1$ since the denominator is the lower bound of the makespan. We use average NSL over 25 sets of task and device graphs.

$$NSL = \frac{makespan}{\min_{i=1,\ldots,n}\left(tda_i + \frac{\sum_{p_j \epsilon cp} c_j}{\mu_i}\right)} \ . \tag{3.2}$$

where cp is the critical path of task graph.

- Efficiency: Speedup is defined as the ratio of sequential execution time of the entire task graph on a single device to the parallel execution time. Efficiency is defined as speedup divided by the number of devices used in the schedule.

$$speedup = \frac{\min_{i=1,\dots,n}\left(tda_i + \frac{\sum_{p_j \in P} c_j}{\mu_i}\right)}{makespan} \quad . \tag{3.3}$$

- Energy consumed in communication overhead: We find the ratio of energy consumed by algorithm in [1] to that consumed by CBS and the ratio of energy consumed by MPI-IOS [35] to that consumed by CBS.

3.5.2 Random graph generation

Random graphs are generated with the following parameters:

- Number of tasks (n): The number of tasks is selected from the set $\{20, 40, 60, 80, 100\}$
- Computation cost (c): Each task in the task graph has CPU requirement in the range of 1 to 200 units.
- Communication to computation ratio (CCR): The average value of CCR is selected from the set $\{0.01, 0.1, 1, 10, 100\}$. Communication is lightweight if the ratio is 0.01 and is most heavy when the ratio is 100.
- Number of devices (m): The number of devices considered are 4, 8, 16, 32 or 64.
- CPU speed of each device (μ): The CPU speed is chosen randomly from the range 1 to 600 units.

Figure 3.3 shows the effect of CCR on average NSL for $CCR = 0.1, 1, 10$. We see that irrespective of the CCR, our scheme performs better than HEFT and gives a lower average NSL. For many samples, HEFT did not generate a schedule, but our scheme did. Such cases were not considered for the average NSL in figure 3.3. In these samples, there was a task with multiple parents scheduled onto multiple devices. The communication cost between devices caused

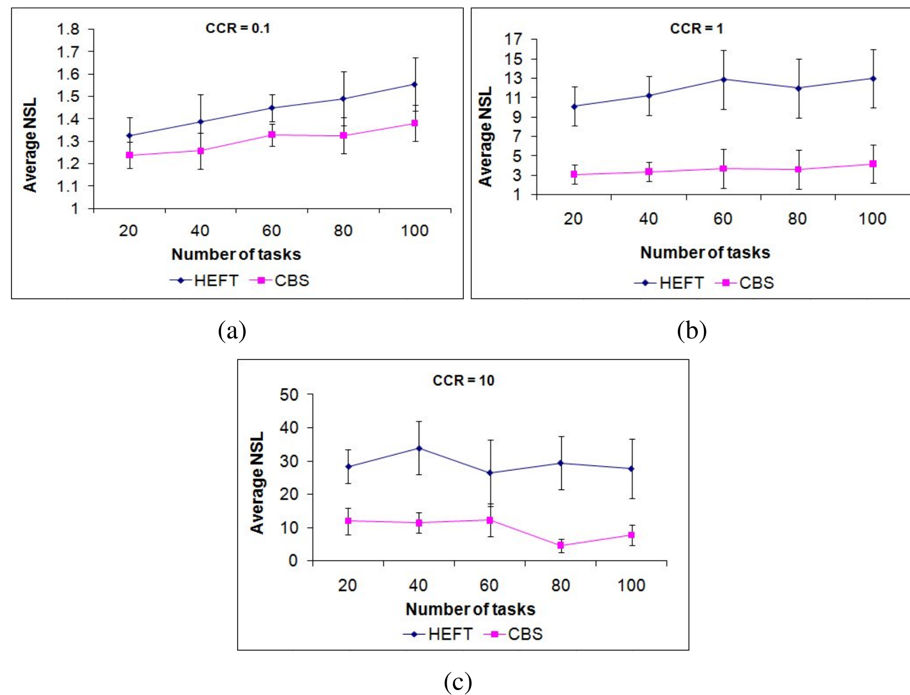


Figure 3.3. Effect of CCR on average NSL. (a) CCR = 0.1 (b) CCR = 1 (c) CCR = 10.

finish time of the child task to become greater than availability of all devices. Our scheme uses a knapsack dynamic programming table and is not based on the earliest finish time. Hence less number of devices were used for the parents and hence the finish time of the child task was within the device availability window.

Figure 3.4 shows the efficiency comparison between HEFT and our scheme. We see that our scheme consistently produces higher efficiency. The number of devices used ranged from 4 to 64. We compared the communication overhead cost in CBS with that in [1] and MPI-IOS. In [1], a device that executes a particular task becomes responsible for scheduling the successors of that task. This requires communication between a task and all other devices. Due to clustering, we were able to reduce the communication required. For figure 3.5, we find the ratio of energy consumption in MPI-IOS and the algorithm in [1] to that of CBS. Figure 3.5 shows the ratios for $CCR = 0.1$ and $CCR = 10$ with $n = 20$ and m ranging from 4 to 64. Number of messages exchanged in [1] and CBS are dependent on device connectivity and

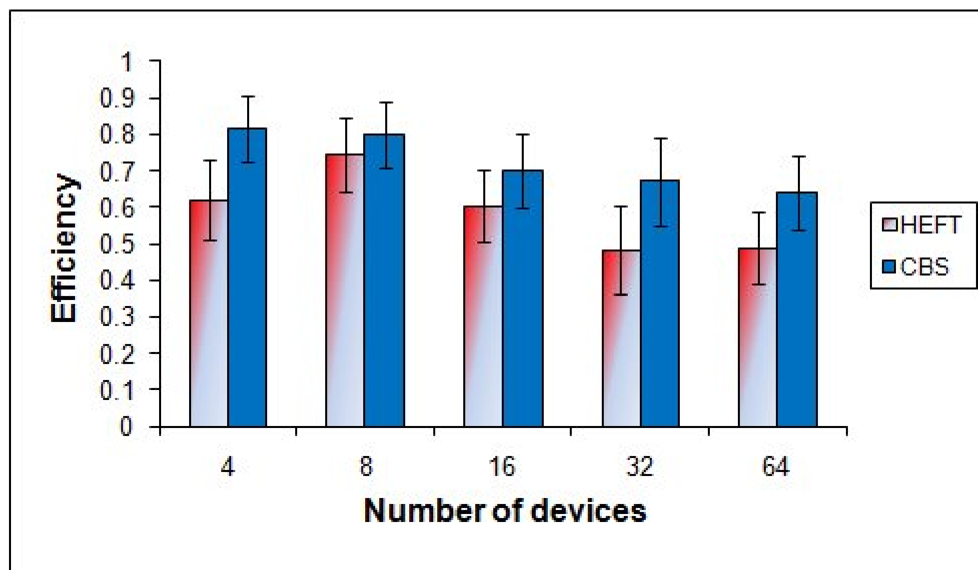


Figure 3.4. Efficiency comparison ($CCR = 0.01, n = 20$).

number of clusters respectively. Number of messages exchanged in MPI-IOS depends not only on device characteristics but also on the time taken to finish the task graph. Hence we observe that energy consumed by MPI-IOS is much larger than that in [1] and CBS.

Figure 3.6 shows the effect of device failures on NSL of CBS. A device is assumed to have failed if the device moves away from the cluster earlier than the previously specified departure time. If a device fails, the corresponding cluster head reschedules the failed task onto some other device. If the cluster head is unable to reschedule the failed task, the client restarts scheduling iterations using the failed service as the starting/first service. Such a failure recovery mechanism incurs huge delay. Hence the NSL increases sharply due to failures.

Figure 3.7 shows the ratio of energy consumption of a centralized algorithm to that of CBS. Energy consumption of CBS is normalized to 1. In a centralized algorithm, each device has to maintain information about all other devices in the network. Hence after regular intervals, each device transfers its information to all other devices in the network. Hence we see that energy consumption of a centralized algorithm is higher than that required by CBS.

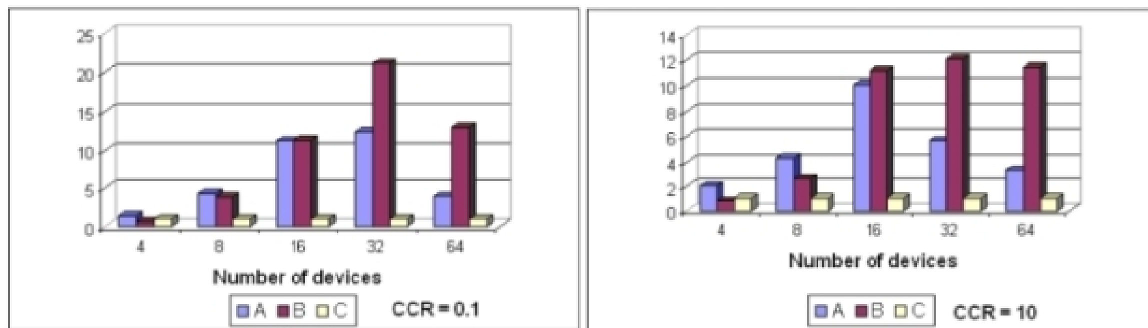


Figure 3.5. Ratios of energy spent in communication overhead of A: algorithm in [1], B: MPI-IOS algorithm with respect to C: CBS.

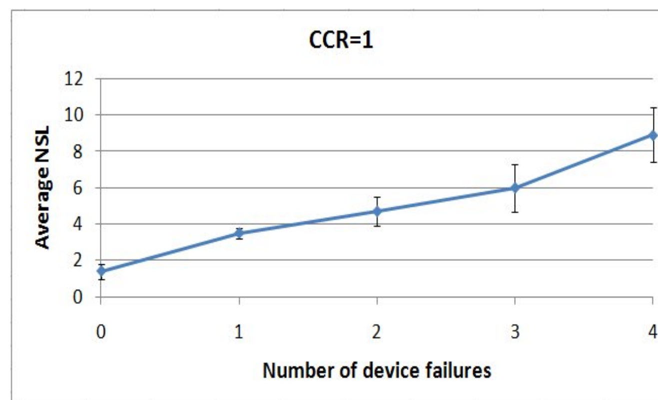


Figure 3.6. Effect of device failure on NSL of CBS.

3.6 Summary and discussion

CBS is a novel cluster based algorithm for task scheduling in parallel pervasive systems. We have shown that the algorithm has less communication overhead for task scheduling. Using simulation results, CBS was compared with two other schemes. The proposed cluster based scheme overcomes some of the challenges - device heterogeneity, service availability and device mobility common to mobile ad hoc networks and pervasive environments. CBS allows the use of multiple scheduling heuristics depending on the capabilities of the devices and cluster head.

In future, an algorithm for handling the user-device interaction challenge should be investigated. Above simulations used clustering based on device connectivity. Algorithms that

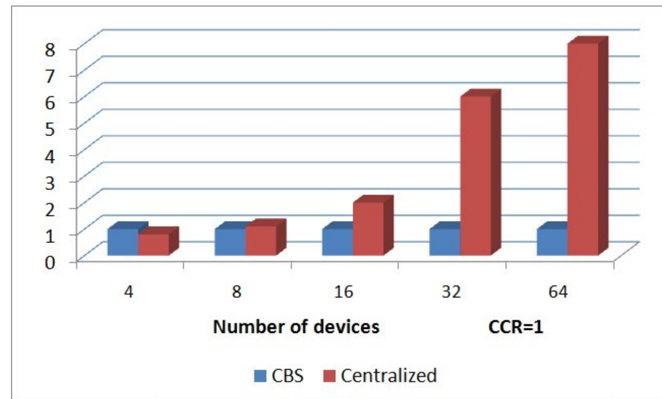


Figure 3.7. Ratio of energy consumption of CBS compared to a centralized algorithm.

perform clustering based on device functionality and the offered services should be exploited to increase efficiency of task scheduling.

CHAPTER 4

DISTRIBUTED SERVICE EXECUTION IN MANETS

Algorithms discussed in Sections 2.2 and 2.3 require either a centralized server or static devices to act as backbone for communication between mobile devices. The algorithms proposed in this chapter aim to relax such requirements. Two algorithms: Decentralized Grading Centralized Selection (DGCS) and Decentralized Grading Autonomous Selection (DGAS) are described. The performance of these algorithms has been compared with [12].

4.1 Network model

The proposed scheme deploys a number of mobile software agents in the network. Mobile agents are installed in the application layer. Since the devices are mobile, the software agents also move from one geographical location to another. Moreover, if the resource level on a device that hosts an agent becomes low, the agent moves from that device to another device with better resources. Each agent maintains, and periodically updates, the cost of communication to all other agents. Devices that host agents do not participate in service execution. Communication between any two devices is done using the network of agents. Consider that the network is comprised of a set of devices $D = \{d_1, d_2, \ldots, d_m\}$. Let $S \subset D$ be the set of devices that host the mobile agents. The devices that do not host agents, henceforth called as participating devices, connect to their closest agent. The classification of devices into agent hosting and participating types is for simplicity. The extension to devices that host agents and provide services is trivial and depends on the capability of device itself. For example, a laptop PC can host an agent as well as participate in execution, whereas a sensor node may be hard-pressed to host agents. The participating devices maintain history of the agents they were connected to while moving

from one place to another. We assume that devices move in a repeatable fashion. For example: people traveling daily for work or college.

4.2 Centralized Grading and Centralized Selection (CGCS)

The algorithms referred to in Section 2.2, use a central server that stores information about all devices in the network. This information is used to computes the probability of success of each device, referred to henceforth as *Grading*. The central server calculates the probability of receiving at least one reply from a subset of devices. The subset is chosen such that the cardinality is minimum, while providing sufficiently high probability of success. The process of identifying the subset of devices to host replicas is referred to henceforth as *Selection*. Hence the existing algorithms can be labeled as Centralized Grading Centralized Selection (CGCS) algorithms. In a CGCS algorithm, the following interactions contribute to the time required to receive a reply:

- Client sends replication request to the agent
- Transmission of service specifications by agent
- Mobile devices send their history to the agent. The agent waits until it receives history of all mobile devices in its vicinity and then computes the success probability of each device
- Transmission of data to selected mobile devices
- Execution of each replica on corresponding device
- Transmission of the result by each replica to the agent
- Forwarding result from the agent to the client

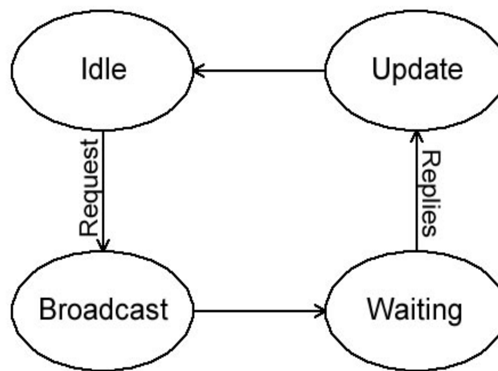


Figure 4.1. State diagrams of Agent in DGAS.

4.3 Middleware modules

4.3.1 Agent

Figure 4.1 shows the state transition diagram of a mobile software agent. In the *idle* state, the agent waits for receiving a replication request. It periodically updates the cost of communication between itself and all other agents. Upon receiving a service replication request, the agent moves into the *Broadcast* state. In this state, the agent transmits the service specifications and data to devices. After finishing the transmission, the agent shifts into *Waiting* state where it waits until a deadline to receive the results. After receiving the first reply, the agent forwards the result to the client. After the deadline expires, the agent goes into *Update* state where it counts the number of replies it had received and updates its internal variables as explained in the Section 4.6. The agent then reenters the *Idle* state.

4.3.2 Participant

Figure 4.2 shows the state diagram of a participating device. In the *Idle* state, the device waits for service replication request. Upon receiving a replication request, the device will move to the *Grading* state where it computes its success probability. The device autonomously decides whether to *Execute* the replica or move back to *Idle* state. This decision is made using the

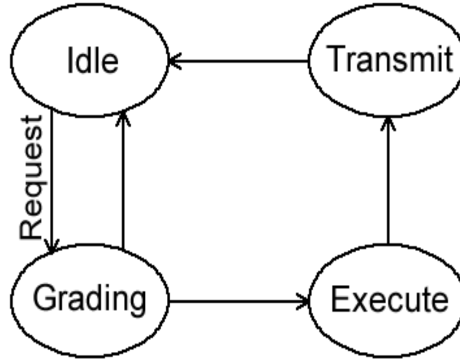


Figure 4.2. State diagram of Participant in DGAS.

Decentralized Grading Autonomous Selection (DGAS) algorithm. In the *Execute* state, the device executes the replica. After the execution succeeds, the device moves into *Transmit* state where it transmits the results to agents within its 1-hop vicinity. The device then reenters the *Idle* state.

4.4 Decentralized grading

Consider a client that requires service J. Since the devices are prone to failures, this service needs to be executed using replication. The client sends the service request to the closest agent hosted on device d_j. The client also provides a time deadline λ by which results should reach back. The agent broadcasts the request to all the participating devices in its vicinity. If a device d_i can provide the required service, it receives data from the agent. Let τ_i be the time taken to transfer data from the agent to d_i, and ε_i be the time required to execute service J on device d_i. Let t be the time at which the client had sent service execution request. Hence the results of the execution will be available at d_i at time $(t + \tau_i + \varepsilon_i)$. We assume that d_i does not disconnect from d_j during the time interval $[t, t + \tau_i]$. If it does, d_i does not participate in replicated execution of service J. Let γ_i^j be the time taken to return back the results from d_i when it is in the vicinity of d_j. During the time interval $[t, t+\tau_i+\varepsilon_i]$, d_i might have moved away

from d_j. Hence at time $(t + \tau_i + \varepsilon_i)$, d_i might be connected to some other agent in the network or might not be connected to any agent. If d_i is connected to an agent at time $(t + \tau_i + \varepsilon_i)$, d_i gives result to the agent which forwards the result to the client using the overlay agent network. If d_i is not connected to any agent, it will require an additional time θ_j to reach agent on device d_j and start sending the result. Hence $\gamma_i^j = t + \tau_i + \varepsilon_i + \theta_j + \delta_j$, where δ_j is the time required to transfer results from agent at d_j to the client. Let $X_i(t)$ be the state of d_i, that is the closest agent that is within the communication range of d_i at time t. Hence $P(X_i(t + \tau_i + \varepsilon_i + \theta_j) = d_j)$ is the probability that d_j is the first device (that hosts an agent), that d_i can use to send results back to the client. Hence:

$$E(\gamma_i) = \sum_{\forall j \in S} \gamma_i^j . P(X_i(t + \tau_i + \varepsilon_i + \theta_j) = d_j) \tag{4.1}$$

that is, the expected value of time taken for the results to reach back the client from device d_i is the sum of product of the time taken for the results to reach from d_i while it is connected to d_j and the probability that d_i will be in communication range with d_j at time $(t + \tau_i + \varepsilon_i + \theta_j)$. Let $\mathbb{N}$ be the set of devices that received service specifications from the agent. The expected time at which the agent receives first reply is given by:

$$E(\gamma) = \min_{\forall i \in \mathbb{N}} E(\gamma_i) \tag{4.2}$$

We now find the probability of getting the result back from d_i within a time deadline λ. If there is only one agent, hosted on a device d_j, such that $\gamma_i^j < \lambda$:

$$P(\gamma_i < \lambda) = P(X_i(t + \tau_i + \varepsilon_i + \theta_j) = d_j) \tag{4.3}$$

If there are multiple possible agents, say on devices $d_j, \ldots, d_m$, to which d_i could connect at time $(t+\tau_i+\varepsilon_i+\theta_j)$ and $\gamma_i^j, \ldots, \gamma_i^m < \lambda$, the probability of receiving result back from d_i within time λ is given by:

$$\begin{aligned} P(\gamma_i < \lambda) &= P(X_i(t+\tau_i+\varepsilon_i+\theta_j) = d_j) \\ &+ \ldots \\ &+ P(X_i(t+\tau_i+\varepsilon_i+\theta_m) = d_m) \end{aligned} \tag{4.4}$$

Hence in general, the probability of getting the result back from d_i within a time deadline λ is given by:

$$P(\gamma_i < \lambda) = \sum_{j \in S, \gamma_i^j < \lambda} P(X_i(t+\tau_i+\varepsilon_i+\theta_j) = d_j) \tag{4.5}$$

Henceforth, we refer to $P(\gamma_i < \lambda)$ as the success probability. For distributed grading, each device records the information about agents it has connected to while moving from one location to another. This history is used to calculate $P(X_i(t+\tau_i+\varepsilon_i+\theta_j) = d_j)$, the probability that d_i will be connected to d_j at time $(t+\tau_i+\varepsilon_i+\theta_j)$. Hence when the mobile device connects to an agent, it acquires:

1. the ID of agent (ID)
2. cost of communication to the agent (t_a)
3. time at which the two devices came in contact (t_{in})
4. time at which the mobile device moved out of communication range of that agent (t_{out}).

Since the devices in pervasive systems may be personal devices, the owner of a service provider device might interrupt the execution to perform a higher priority service. Information like call frequency and call duration can be included to decide the value of $P(X_i(t+\tau_i+\varepsilon_i+\theta_j) = d_j)$. [16] studies daily interaction between user and his cellphone to perform context recogni-

tion. We envisage including such information in future work. In DGCS and DGAS, devices autonomously compute their own success probability. Hence depending on configuration of each device, owner of the device can decide the factors to be used while computing the probability. Since security measures are beyond the scope of this dissertation, we have not implemented security during replica distribution. Table 4.1 shows example mobility history maintained by device d_i. Consider that d_i receives replication request from an agent with $ID = 4$, $\varepsilon_i = 50$ and $\lambda = 175$. d_i parses its history and finds that at time $(t + \tau_i + \varepsilon_i)$ it might be connected to agent with $ID = 2$ or 1 or might not be connected to any agent. At time $(t + \tau_i + \varepsilon_i + \theta_j)$, d_i might be connected to agents 1, 2 and 8. Hence $P(X_i(t + \tau_i + 50 + \theta_1) = 1) = P(X_i(t + \tau_i + 50 + \theta_2) = 2) = P(X_i(t + \tau_i + 50 + \theta_8) = 8) = \frac{1}{3}$. Let $t + \tau_i = 75$, $\delta_1 = 45$, $\delta_2 = 30$ and $\delta_8 = 33$. Hence, from Equation 4, $P(\gamma_i < \lambda) = P(X_i(t + \tau_i + 50 + \theta_2) = 2) + P(X_i(t + \tau_i + 50 + \theta_8) = 8) = \frac{2}{3}$

Table 4.1. Example Journeys

Journey 1				Journey 2				Journey 3			
ID	t_a	t_{in}	t_{out}	ID	t_a	t_{in}	t_{out}	ID	t_a	t_{in}	t_{out}
3	10	20	30	4	12	25	33	6	2	1	8
6	3	46	50	10	22	43	51	4	14	11	21
4	13	55	65	1	21	56	70	10	25	30	49
1	23	83	89	8	9	81	89	1	11	58	72
8	8	90	97	12	18	91	99	2	14	77	88
2	13	101	110	9	8	110	118	8	6	95	103
9	6	113	139	23	12	130	148	9	7	120	140
5	34	158	164	5	31	156	167	5	36	159	169
7	27	174	183	7	23	173	185	7	24	176	182

Lemma 4.1: $P(\gamma_i < \lambda) \leq 1$, that is the success probability of device d_i cannot exceed 1.

Proof: Device d_i computes $P(\gamma_i < \lambda)$ for every service request. d_i creates a set containing IDs of agents that d_i has met $(\tau_i + \varepsilon_i + \theta_j)$ time units after meeting the current agent. $P(X_i(t + \tau_i + \varepsilon_i + \theta_j) = d_j)$ is the number of times d_j appears in the set, divided by cardinality

of the set. Equation 4.5 considers only those agents that have $\gamma_i^j < \lambda$. Hence $P(\gamma_i < \lambda)$ will always be ≤ 1.

4.5 Decentralized Grading and Centralized Selection (DGCS)

In DGCS, each device computes its success probability and transmits the value to the agent. After receiving success probabilities from the devices, the agent selects n devices to replicate the task and sends the input data or service input parameters to the selected devices. It might happen that the remainder contact time between the agent and a device is smaller than the time required to transfer the parameters. Hence all the n devices selected by the agent might not receive input parameters. It is well known that the contact time follows a heavy tailed distribution [64]. In DGCS each device also computes $P(r_i^j > \tau_i)$: the probability of successfully receiving the input parameters, where r_i^j is the residual contact time between device d_i and the agent on device d_j. Thus for DGCS the success probability is computed as:

$$P(\gamma_i < \lambda) = P(r_i^j > \tau_i) * \sum_{j \in S, \gamma_i^j < \lambda} P(X_i(t + \tau_i + \varepsilon_i + \theta_j)) = d_j \tag{4.6}$$

Each device computes its own success probability using equation 4.6 and sends the value to the agent. The agent sorts the received success probabilities in descending order and chooses first n devices to host replicas of the task. The probability of receiving at least one result within the deadline λ, from the set of n replicas, is given by:

$$P(J) = 1 - \prod_{n} (1 - P(\gamma_i < \lambda)) \tag{4.7}$$

The value of n is determined such that $P(J)$ becomes acceptable, that is $P(J) > \psi$, where ψ is a user specified value. In DGCS, the following interactions contribute to the time required to receive the results:

- Client sends replication request to agent.
- Transmission of task specifications by agent.
- Mobile devices send success probability. The agent waits until it receives success probability values of all mobile devices in its vicinity.
- Transmission of data to selected mobile devices.
- Execution of each replica on the corresponding device.
- Transmission of the result by each replica to the agent.
- Forwarding result from the agent to the client.

Lemma 4.2: DGCS has lower communication overhead as compared to CGCS.

Proof: In a CGCS algorithm, the participating devices send their history to the agent. The agent waits till it receives history from all the participating devices in its vicinity. Hence, the selection process starts earlier in DGCS as compared to CGCS. Let η be the number of devices that receive task specification, t_s be the time taken to transmit task specifications, $\acute{h}$ be the average time taken to receive history from one device, t_g be the time taken to compute reliability of one device and t_d be the time taken to transmit data to selected devices. For a CGCS algorithm, the expected time taken to receive the reply from a particular replica is given by Equation 4.1 with $\tau_i = t_s + \eta.\acute{h} + \eta.t_g + t_d$. For DGCS, the expected time is calculated using $\tau_i = t_s + t_g + \eta.\acute{g} + t_d$, where $\acute{g}$ is the average time taken to receive success probability values from all η devices. Since the success probability value is one floating point number per device, $\acute{g} \ll \acute{h}$. Hence it is obvious that DGCS has lower communication overhead as compared to CGCS.

Algorithm 3 DGAS

1: Initialise $\Theta = \Theta_{init}$ at each mobile agent
2: **if** Device hosts agent and service is to be replicated **then**
3: Broadcast service request to all mobile devices in 1-hop
4: Broadcast Θ and deadline λ along with the request
5: Receive replies
6: **if** r_{min} < Number of replies < r_{max} **then**
7: $\Theta = \Theta - v_s$
8: **if** Number of replies > r_{max} **then**
9: $\Theta = \Theta - v_l$
10: **if** $\Theta < \Theta_{min}$ **then**
11: $\Theta = \Theta_{min}$
12: **if** Number of replies < r_{min} **then**
13: $\Theta = \Theta + v_i$
14: **if** $\Theta > 1$ **then**
15: $\Theta = 1$
16: **if** Number of replies = 0 and $\Theta = 1$ **then**
17: $k = k + 1$
18: send replication request to neighboring agents in k-hops
19: receive replies
20: **if** Number of replies > r_{min} **then**
21: $k = k - 1$
22: **if** Device does not host agent and request is received **then**
23: Compute $P(\gamma_i < \lambda)$ using Equation 4.5
24: **if** $P(\gamma_i < \lambda) > 0$ **then**
25: Decide participation with probability Θ
26: **if** Device is going to replicate service **then**
27: Execute service
28: **if** $\Theta = 1$ **then**
29: Opportunistically forward request to mobile devices

Each agent stores a probability value, Θ. This value is used to decide the number of service providers that will execute the replica. Algorithm 3 shows the decentralized solution. The algorithm starts with $\Theta = \Theta_{init}$. This value is sent along with the service replication request to all the participating devices in the 1-hop distance of the agent. When a request is received,

the devices check whether they can provide the required service. The matched service providers then find their corresponding success probability using Equation 4.5. If success probability > 0, each provider decides with a probability Θ, whether to execute a service replica. If the provider decides to execute the replica, the service is executed and the device tries to return the results. Let $E \subset D$ be the set of devices that decide to execute a service replica. Each agent stores three threshold values: r_{min}, r_{max} and Θ_{min}. If an agent receives more than r_{max} replies, Θ is reduced by a large value, v_l. If the number of replies received is between r_{max} and r_{min}, Θ is reduced by a smaller value, v_s. If the value of Θ is too low, the agent might not get any replies. Hence Θ_{min} is the lower limit on the value of Θ. If the number of replies received is less than r_{min}, Θ is increased by a value v_i. This updated value is used during the next service execution request. If $\Theta = 1$ and no replies are received, the agent forwards the service execution to its neighboring mobile agents within k-hops. The neighbors broadcast the request to service provider devices in their vicinity and follow the above procedure. If the software agent does not receive a result, k is increased. If many copies of the result are received, k is decreased.

Opportunistic networks [65] are networks consisting of mobile devices where the devices are not connected to each other for most of the time. When two devices come within communication range, they exchange messages. Such opportunistic communication is used to propagate service replication request. The devices do not maintain a record of intercontact times and contact times. The agents transmit the replication request to devices within 1-hop range. Therefore some devices may not be connected to any agent at time the replication request was sent. Let $C \subset D$ be the set of such devices. Let $d_c \in C$. Hence d_c has not received the replication request. d_c might opportunistically come in contact with a device $d_e \in E$. d_c then calculates the value of $P(\gamma_c < \lambda)$, that is the probability that d_c will completely execute service request and return results back to client within deadline λ. If $P(\gamma_c < \lambda) > P(\gamma_e < \lambda)$, that is, if d_c is more probable to succeed than d_e, d_c also starts executing a replica. Thus a device that is executing a replica, opportunistically spreads the replication request to other devices in the network.

Lemma 4.3: DGAS has lower time overhead than CGCS.

Proof: In DGAS, following interactions contribute to the time required to receive a reply:

- Client sends replication request to agent
- Transmission of service specifications by agent
- Transmission of data by agent
- Execution of each replica on the corresponding device
- Transmission of the result by each replica to the agent
- Forwarding result from the agent to the client

DGAS avoids the synchronization phase in CGCS and the success probability is computed using $\tau_i = t_s + t_g + t_d$. Thus DGAS has lower time overhead than CGCS.

4.7 Simulations

For all the simulations, we consider $\Theta_{init} = 1$. The simulations consider a 300m * 300m area. Each device has average speed of 25 meters per hour and average communication radius of 12.5m. Simulations for results shown in Figures 4.3 and 4.4 are performed using the city section mobility model [66]. The simulation for results shown in Figure 4.5 uses random waypoint mobility model [67]. Each device stores history of 50 journeys. We simulate replicated execution of an audio noise removal application. The client performs cyberforaging due to absence of the required software on the client. Time required to execute service on each device, that is ε_i, is randomly chosen from the range $[50, 300]$ seconds. We consider the corresponding service to be hosted by 50% of devices. [12] is used as the CGCS algorithm.

In our proposed algorithm, communication of results between any two devices is done using the network of agents. Hence, we study the effect of varying the number of agents. Figure 4.3 shows the variations in the number of service providers that have success probability

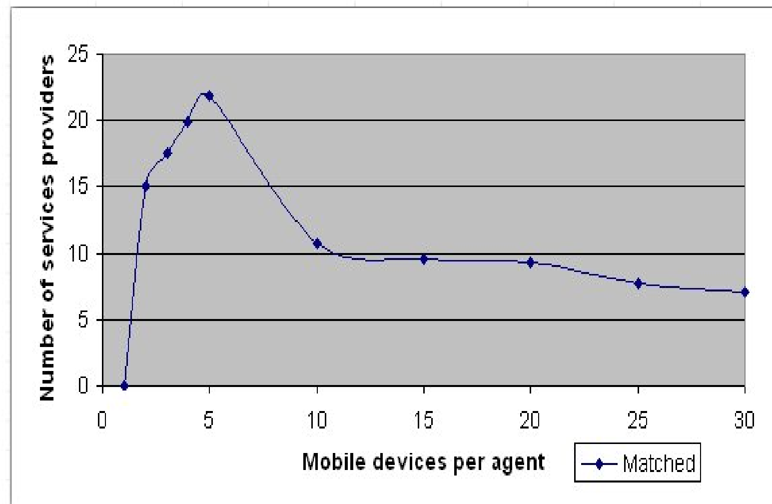


Figure 4.3. Number of service providers having success probability > 0.

> 0 with respect to the number of mobile devices per agent. We used 400 devices and varied the number of agents such that number of mobile devices per agent was in the range [1, 30]. Each reading is an average of 50 service executions. If x is the number of devices per agent, it does not imply that an agent has a maximum of x neighbors. The number of neighbors an agent has at a particular time depends on the mobility of devices. With 400 devices, and say $x = 5$, there are 80 devices that host agents and the remaining 320 can offer services. When the number of mobile devices per agent is 1, an agent is deployed onto every device. Since the device that hosts an agent does not execute replicas, there are no service providers available for replica execution. As the number of devices per agent increases, some of the devices host agents while some participate in service provisioning. Hence the number of services that have success probability $>$ 0 increases till a certain limit. When the number of devices per agent increases even more, probability that a service provider device will be in vicinity of an agent decreases. Hence when service providers want to return back results, only a small number of those providers have success probability $>$ 0. Thus, in Figure 4.3, number of matched service providers first increases and then decreases with increasing value of x.

Table 4.2. Time (sec) required to start playing audio, with small number of devices and two agents in the network

Number of Devices in Network	CGCS	DGCS	DGAS
10	484	406	405
20	498	413	412
30	514	418	416
40	537	421	420
50	573	427	426

Table 4.3. Time (sec) required to start playing audio, with large number of devices and 25 agents in the network

Number of Devices in Network	CGCS	DGCS	DGAS
100	651	456	449
200	699	467	462
300	968	480	469
400	1121	484	473
500	1192	491	483

Tables 4.2 and 4.3 show the time required for receiving the first reply, using the CGCS [12], DGCS and DGAS algorithms. In Table 4.2, we vary the number of mobile devices in the network from 10 to 50. There were 2 mobile agents. In Table 4.3, we vary the number of mobile devices in the network from 100 to 500. There were 25 mobile agents. For both of the simulations, $r_{min} = 1$, $r_{max} = 4$, $\Theta_{min} = 0.01$, $v_l = 0.4$, $v_s = 0.15$ and $v_i = 0.1$. Each reading is an average over 50 task executions. In a CGCS algorithm, the participating devices send their history to the agent. The agent waits till it receives history from all the participating devices in its vicinity. In DGCS the devices send only the success probability. The agent waits till it receives the probability values from all the participating devices in its vicinity. Hence, in DGCS, the agent has to wait for a smaller amount of time. This synchronization is absent in the DGAS algorithm. Hence Tables 4.2 and 4.3 show that the difference of time required to receive the first reply increases with the number of mobile devices. The difference in time requirement

is due to two situations. First, as explained in Section 4.5 and Section 4.6, τ_i in CGCS, DGCS and DGAS is given by $(t_s + \eta.\acute{h} + \eta.t_g + t_d)$, $(t_s + t_g + \eta.\acute{g} + t_d)$ and $(t_s + t_g + t_d)$ respectively. Hence the start time of execution of service J on device d_i using DGAS < start time in DGCS < start time in CGCS. Hence d_i might send back results earlier in DGAS and DGCS. Secondly, since the communication overhead is higher in CGCS, there might be devices that send their history to the agent, get selected to execute the service replica, and move out of range before the agent sends them the data. Hence the actual probability of receiving at least one reply might be lesser than that computed by the agent. Due to lower communication overhead in DGCS, there is a smaller probability that the above might occur in DGCS.

In a VANET, the devices on vehicles interact with roadside equipment. Hence for VANETs, the agents were placed on immobile devices at the intersections. Thus the communication cost among agents can easily be updated periodically. The participating mobile devices are either on some vehicle or with some pedestrian. Communication between mobile devices is done using the network created by agents. Hence whenever a client wishes to execute a service, the request is sent to the nearest agent instead of the client itself broadcasting the request directly to its neighbors. The mobile devices move in a repeatable fashion. Each device has a fixed starting point and a fixed destination. Although the devices move towards their corresponding destinations, the direction they choose at every intersection is not fixed. The number of matched service providers is considered to be proportional to the number of devices at the intersection. The number of vehicles entering an intersection is known to have normal distribution with two peaks indicating the two peak periods in morning and evening. Hence each agent also encounters mobile devices with the same distribution.

Figure 4.4 shows the number of replicas executed when the number of service providers follows a distribution. In Figure 4.4, the first data series indicates the number of service providers that had success probability > 0. The second series shows the number of replicas used in DGCS and CGCS algorithms. The third trend shows the number of replicas used in

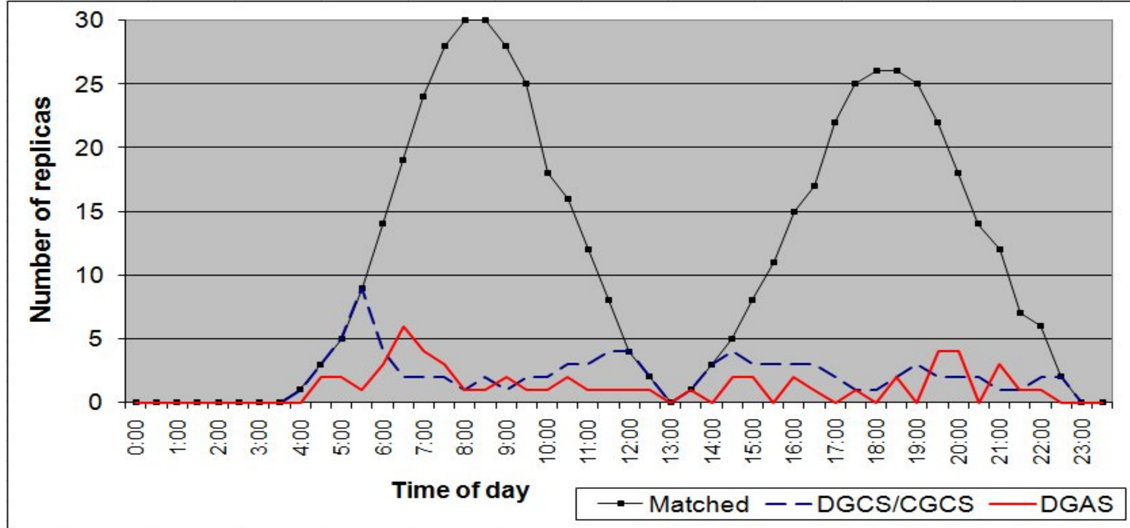


Figure 4.4. Number of replicas executed when service providers follow a distribution.

DGAS. $r_{min} = 1$, $r_{max} = 4$, $\Theta_{min} = 0.01$, $v_l = 0.4$, $v_s = 0.15$ and $v_i = 0.1$. We consider the peak periods to be between 7 to 10 a.m. and 5:30 to 8:30 p.m. The x-axis shows 48 intervals, that is, half hour intervals of the entire day. During each interval, a service replication request is generated. Hence, the number of matched service providers available at each request also follows the distribution. As explained earlier, due to the higher communication overhead in CGCS, there might be some devices that were selected to execute service replica, but moved out before receiving the data. In our simulations, the number of devices that moved out in CGCS but received data in DGCS is very small. Also note that each reading in Tables 4.2 and 4.3 is an average of 50 service executions, whereas in Figure 4.4, each reading is a single execution. Thus series for number of replicas using CGCS and DGCS are almost identical. Hence for Figure 4.4 we consider only one trend for DGCS. Since initially $\Theta = 1$ in DGAS, the number of replicas executed is same as number of providers available. When the agent receives replies, it follows Algorithm 3 and changes the probability accordingly. Thus Θ reduces during peak periods and increases during afternoon to accommodate fewer number of devices available on the roads. If the number of service providers follows a certain distribution, the number of providers

available at the time of replication request can be predicted, and hence instead of incrementing or decrementing Θ as explained above, Θ can be equated to $1/predictedValue$. Though this would give lesser number of replicas being executed, having $\Theta = 1/predictedValue$ will not give good results if the number of matched providers do not follow any specific distribution. Thus it is necessary to increase or decrease Θ according to Algorithm 3.

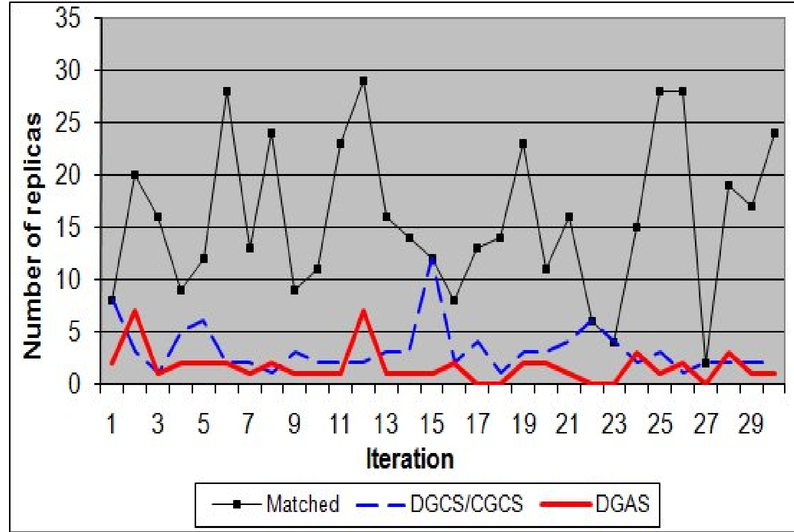


Figure 4.5. Number of replicas executed with random number of service providers.

Figure 4.5 shows the effect of variations in Θ when the number of matched service providers does not follow a distribution. This experiment does not consider any specific scenario such as a road intersection. We use the random waypoint mobility model [67]. We generate random number of service providers for each service replication request. $r_{min} = 1$, $r_{max} = 4$, $\Theta_{min} = 0.01$, $v_l = 0.4$, $v_s = 0.15$ and $v_i = 0.1$. As explained above, the number of replies executed using CGCS and DGCS are considered equal. Algorithm 3 tries to keep the number of replies received to a minimum of r_{min}. The parameters v_l, v_s and v_i are used to update the value of Θ. Hence if there are same number of replies for consecutive replication requests, the value of Θ will eventually converge to Θ_{min}. Comparing Figure 4.4 with Figure 4.5, we can see

that the algorithm gives better convergence and is more effective when the number of service providers follow a distribution.

During simulations corresponding to Figures 4.4 and 4.5, there were instances when none of the devices executed the requested service. Since CGCS has complete knowledge of the devices in the vicinity, CGCS was able to select devices to execute the replica. Hence even though there were devices that could have executed devices, no replicas were initiated due to the moderation added by Θ. Hence we see that the trend for DGAS has a value of zero at many more iterations as compared to CGCS. Hence even though CGCS can choose the optimal set of devices for replication, Figures 4.4 and 4.5 appear to show DGAS performing better than CGCS.

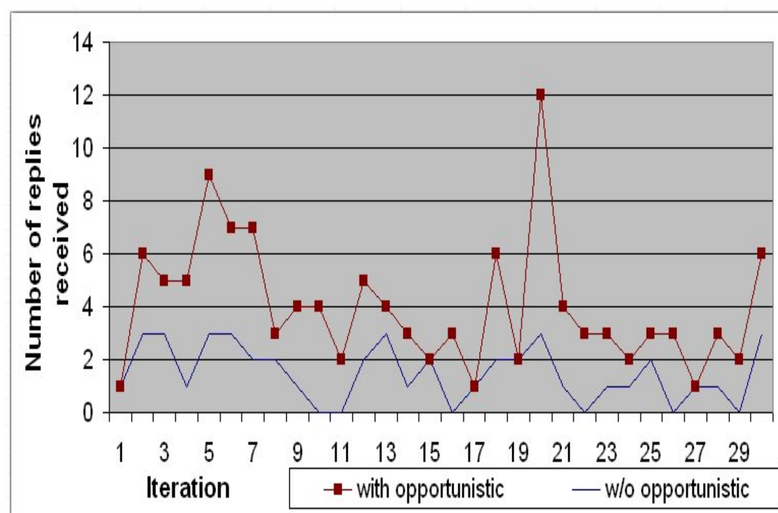


Figure 4.6. Number of replies received without and with opportunistic spreading of replication request (using DGAS).

Figure 4.6 shows the effect of using opportunistic forwarding of replication request. We use the following parameters: $r_{min} = 3$, $r_{max} = 10$, $\Theta_{min} = 0.1$, $v_l = 0.2$, $v_s = 0.05$ and $v_i = 0.3$. 100 devices and 5 agents were used. Few of the 100 devices were near the same mobile agent as

the client. Hence when Θ became 1, the mobile devices started spreading replication requests to devices that have not received the request. As a result, the client received more number of replies. In Figure 4.6 the first data series shows the number of replies received by the client received when opportunistic spreading was used. The second series shows the number of replies when $\Theta = 1$ and opportunistic spreading was not used.

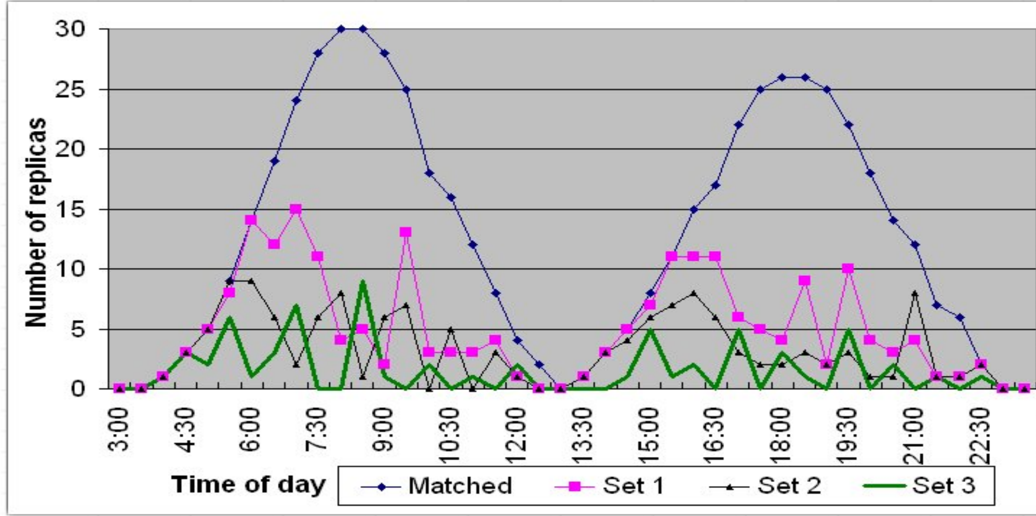


Figure 4.7. Effect of selected parameters on number of replicas executed using DGAS.

Figure 4.7 shows the effect of selected values for r_{min}, r_{max}, Θ_{min}, v_l, v_s and v_i on the number of replicas executed. The simulation setup for the diurnal pattern is same as that for Figure 4.4. Set 1 refers to: $r_{min} = 3$, $r_{max} = 10$, $\Theta_{min} = 0.1$, $v_l = 0.2$, $v_s = 0.05$ and $v_i = 0.3$. Set 2 refers to: $r_{min} = 2$, $r_{max} = 6$, $\Theta_{min} = 0.05$, $v_l = 0.3$, $v_s = 0.1$ and $v_i = 0.2$. Set 3 refers to: $r_{min} = 1$, $r_{max} = 4$, $\Theta_{min} = 0.01$, $v_l = 0.4$, $v_s = 0.15$ and $v_i = 0.1$. Set 3 tries to keep Θ at a lower value to get lower number of replicas executed. Thus parameters in Set 3 are more restrictive than that in Set 2 which are more restrictive than values in Set 1. As the parameters become more restrictive, lesser number of replicas are executed, but zero replies are obtained more often.

We have also implemented DGAS onto Sharp Zaurus PDAs and Toshiba netbooks. The devices communicate using an ad hoc network. We simulate the mobility of these devices as follows. One netbook simulated the mobility of nodes by running the Random Waypoint mobility model corresponding to all the network nodes. When two devices in the simulation come within communication range, we establish a communication link between them. Figure 4.8 shows the number of replicas executed when random number of service providers were within communication range of an agent.

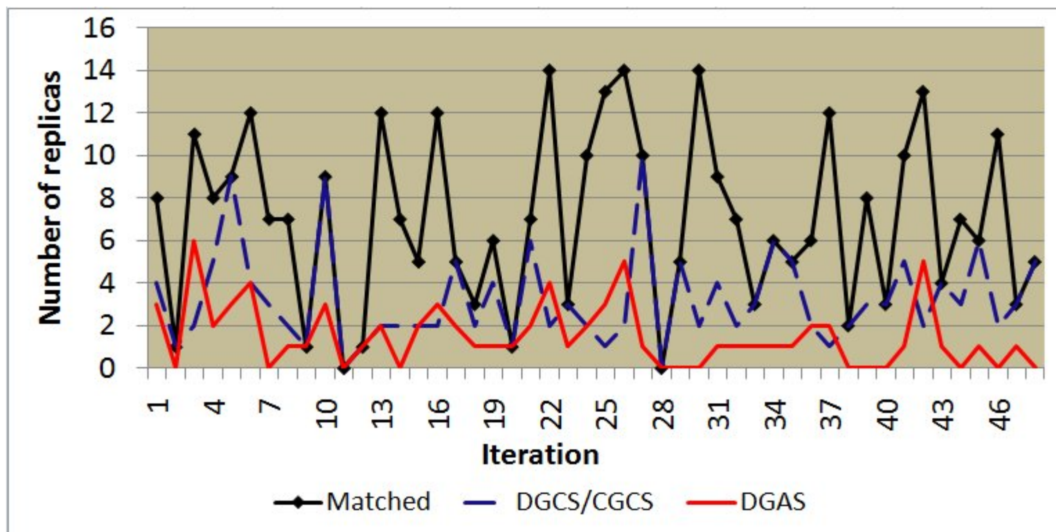


Figure 4.8. Number of replicas executed using PDAs and netbooks.

4.8 Summary and discussion

We presented a novel algorithm that enables decentralized decision making for fault tolerant service execution. Software agents are used to mask the network dynamics, enhance autonomy and increase efficiency in the dynamic environments. In DGAS, each device computes its success probability and decides whether to execute the service replica. As shown using simulations, the proposed scheme can deploy minimum number of replicas if the number

of available service providers follows a distribution. The major advantage of DGAS is that it avoids the synchronization phase required by the CGCS algorithm.

In DGAS, since the selection process is performed autonomously, the agent does not have any means to compute the probability of receiving at least one reply. Hence making the value of Θ dependent on the number of replies received in previous iteration is the correct approach to the situation. DGAS is more suitable for replicated execution of services used in non-emergency situations or when the user does not pay for the fault tolerant service execution. For example, students moving around a college campus can share services with other students. Even though DGCS has more communication overhead, it is more suitable for situations where the client requires previously agreed upon quality of service. If the devices in the network are highly unreliable, that is, have very low success probabilities, the agent (in DGCS) will select almost all of the devices to satisfy $P(J) > \psi$. For such networks, in DGAS, the value of Θ will quickly reach 1. Hence DGAS is better suited for such networks with highly unreliable devices.

In the proposed algorithms, we considered mobility as the factor for failure. Other factors such as residual energy, random hardware faults can be added more easily to DGAS than to CGCS algorithms. Agents broadcast replication request within 1-hop distance. The algorithms can be easily modified to search for service providers within multiple hop distance. The analysis and algorithms presented in this chapter would remain the same. We are working on algorithms that can learn the variations of Θ and can predict the optimal value. Prediction of the optimal value of Θ would maintain the number of replies received back by client to a minimum. For future work, we also envisage to consider networks that do not have agents deployed in them. In such networks, using a decentralized algorithm to control the number of replicas is a challenging problem.

CHAPTER 5

MUTUAL EXCLUSION FOR DISTRIBUTED SERVICES IN OPPORTUNISTIC NETWORKS

Opportunistic networks are essentially distributed networks with transient connectivity among nodes. Nodes in opportunistic networks are resource constrained, mobile and infrequently come in contact with each other. A series of opportunistic contacts in time and space leads to opportunistic paths. An opportunistic network comprises a set of opportunistic contacts and paths, distributed in space and time. The topology of such an opportunistic network changes frequently due to mobility, link and node failures. In the near future, several applications are expected to exploit opportunistic networks.

In such a distributed network, nodes may require exclusive access to a shared object or resource. Examples of problems that require mutual exclusion are: assigning unique values such as IDs and IP addresses to mobile nodes in dynamic networks [48], ensuring total ordering of message delivery in mobile networks [49] and updating a shared file in a peer to peer environment. Ensuring freedom from starvation is a challenging problem in opportunistic networks due to the infrequent contacts and node failures. This chapter describes a novel token based algorithm and proves that it is free from starvation and deadlock, and satisfies the safety property. Unlike existing algorithms for MANETs, the proposed algorithm does not require continuous monitoring of the network topology. Simulation results show that MEOP is communication efficient as compared to other algorithms proposed for generic mobile ad hoc networks. We also propose a timeout based fault detection algorithm that exploits the inter contact time distributions. To the best knowledge, MEOP is the first to support mutual exclusion in opportunistic networks.

5.1 Network model

Let N be the total number of nodes in the network. Nodes are represented as n_1, n_2, ..., n_N. Let $E[T_{i,j}]$ be the average intercontact time between two nodes n_i and n_j, where $i, j \in P$. Let $E[T_i]$ be the average inter-any-contact time between node n_i and all other nodes that it connects to. We first list the assumptions:

1. Inter contact time between two nodes has either a heavy tailed or an exponential distribution with average $E[T_{i,j}]$.
2. The distributions of inter contact times and inter-any-contact times have reached a steady state before the algorithm for mutual exclusion is executed.
3. A node n_i might come in communication range of only a subset of nodes in the network.
4. When two nodes are in communication range of each other, messages are transferred in FIFO order. The nodes communicate synchronously and hence any message loss can be detected by both the nodes.
5. Multihop communication between two nodes need not preserve FIFO order of the messages.
6. Communication channel between neighboring nodes is bidirectional.
7. Though the network is partitioned most of the time, there eventually will be a path between any two nodes.
8. Clock drift is negligible.

5.2 Mutual exclusion

Figure 5.1 shows the state transition diagram of applications that need mutual exclusion. Each node maintains a list of pending requests. When a node wishes to enter its CS, it generates a request and adds the request to the list. The request is propagated, when the requestor opportunistically comes within communication range of other nodes. The requestor then waits

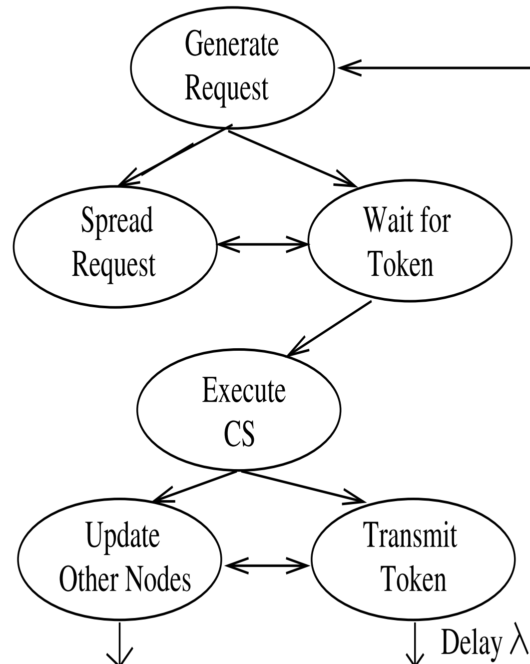


Figure 5.1. State transition diagram of application requiring mutual exclusion.

to receive the token. A node does not generate another request if its current request has not been satisfied. The requestor enters CS as soon as the token is received. For the period of CS execution, a node may turn off its radio and stop communicating with other nodes. After completing the CS, whenever the token holder opportunistically comes in contact with other nodes, the token holder informs other nodes about the location of the token. If there are any pending requests in the request list, the token holder selects the request that was generated earliest and tries to route the token to the corresponding node. After exiting the CS, the node generates another request after an average of λ time units. A node, n_i, that receives the token might have multiple requests (including its own) pending in its request list. If its own request was not generated earlier than all of the other requests, the node may again enter its CS only if $E[T_i]$ is greater than the expected time required to finish the CS. Else, n_i waits and transfers the token to other nodes.

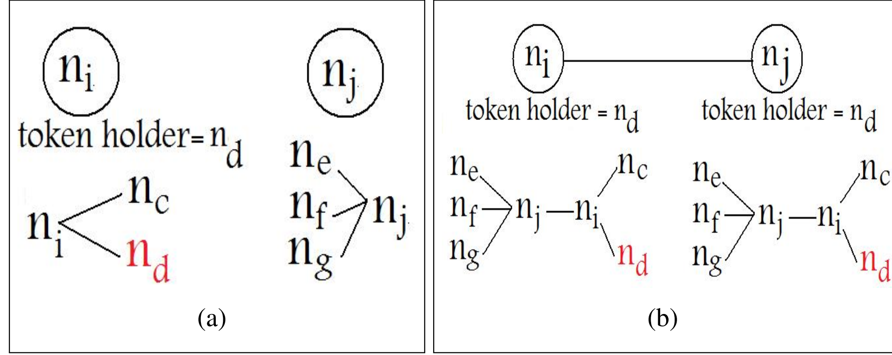


Figure 5.2. Example of opportunistic data transfer. (a) n_i, n_j have partial information. (b) n_i, n_j opportunistically exchange information.

When two nodes opportunistically come within range, they exchange such information as requests for CS, token location and routing tables recorded by both the nodes till that time. Figure 5.2 shows communication exchange between two nodes of the network. In Figure 5.2(a), nodes n_i and n_j have partial information about the network. Let nodes n_c, n_d, n_e, n_f and n_g, where $c, d, e, f, g \in P$, be some of the other nodes in the network. Node n_i is aware that node n_d is the token holder. Node n_j does not have information about the location of the token. Figure 5.2(b) shows the combined graph of intercontact times. Since information is spread only upon opportunistic contacts, some nodes in the network will have older information than others. When such nodes come in contact, it is necessary to decide which node has the recent information. Hence such information as generated request and ID of the last known token holder carries a timestamp. Using the timestamp, the older information is updated with the recent. For example, if node n_i knows n_d as the last known token holder and node n_j has n_e as the token holder, a timestamp associated with the information helps identify recent most information. Depending on the node mobility patterns, a node might meet only a subset of nodes in the network. For example, in Figure 5.2, n_c might never meet n_e. For such cases average intercontact time is computed as the minimum sum of intercontact times over all paths between the two nodes. Thus $E[T_{c,e}] = E[T_{c,i}] + E[T_{i,j}] + E[T_{j,e}]$.

Algorithm 4 Mutual Exclusion in OPportunistic Networks

{id: variable that stores ID of the node}

Procedure generateRequest() {Generate Request and Start Request Propagation}

```
1: if tokenHolder is known then
2:     Request_id = (T_id, id)
3:     routeRequestTo(tokenHolder)
4: end if
```

Procedure routeRequestTo(destination) {Forward Request to Token Holder}

```
1: if tokenHolder == id then
2:     Add request to list of pending requests
3: else
4:     Store local copy of the request
5:     exchangeInfo()
6:     Use routing protocol to find next hop towards tokenHolder
7:     Transfer request to next hop
8: end if
```

Procedure consumeToken() {Execute CS and forward token to successor}

```
1: Execute critical section
2: if request list is not empty then
3:     successor = get next high priority node from request list
4:     forwardToken(successor, request list)
5: else
6:     exchangeInfo()
7: end if
```

Procedure forwardToken(destination, request list) {Forward token and list of pending requests to destination}

```
1: if destination == id then
2:     consumeToken()
3:     generateRequest()
4: else
5:     exchangeInfo()
6:     Use routing protocol and transfer token to next hop towards destination
7: end if
```

Procedure exchangeInfo() {Update information with neighbors}

```
1: Transmit network topology, ID of token holder to all neighbors
```

Algorithm 4 shows the proposed algorithm, Mutual Exclusion for OPportunistic network (MEOP). The algorithm is explained in detail in the following subsections.

5.2.1 Request generation

During the initialization of the system, ID of token holder node is not known to other nodes. Hence the token holder floods the network informing its ID to all other nodes. Once the location of the token becomes known, nodes can start generating requests. When node n_i wishes to enter its CS and does not have the token, it creates a request. Each request consists of a timestamp representing the time at which the request was generated and the ID of the corresponding node. The node executes its CS only after receiving the token. The node can generate another request for token only after exiting its CS, that is, the node does not generate a request if its previously generated request is not satisfied.

5.2.2 Request propagation

In traditional DAG based algorithms, the edges of the DAG represent the path along which token request is routed. The DAG needs to be updated if the network topology changes due to node mobility. This continuous monitoring of the network introduces a large overhead. The DAG is also updated when the token is transferred from one node to another. The proposed MEOP algorithm is a DAG based algorithm. Nodes of the DAG in MEOP point to the last known token holder. The DAG need not be updated when the topology changes. Thus the DAG in MEOP is a logical DAG, as compared to the routing based DAG used in traditional algorithms. The logical DAG is independent of the underlying node mobilities. The token request messages and the token might be propagated over a multihop route. The actual multihop routing of the token requests and the token is performed by underlying routing protocols. Since routing protocol is not the focus of this dissertation, we adopt such existing routing algorithms as [5]. MEOP abstracts the routing algorithms as follows: using some routing criteria, each

node is assigned a rating. When node n_i wishes to send a message to n_j, and they are not in the communication range, n_i transfers the message to some other node n_k only if n_k has a higher rating than n_i, for the given criteria. The process is repeated till message reaches n_j. For example: in BubbleRap [5], node popularity and community membership are used as the routing criteria. In our simulations, we use intercontact time as the routing criteria. When n_i wishes to transfer a message to n_j, and they are not in the communication range, n_i transfers the message to another node n_k only if $E[T_{k,j}] < E[T_{i,j}]$.

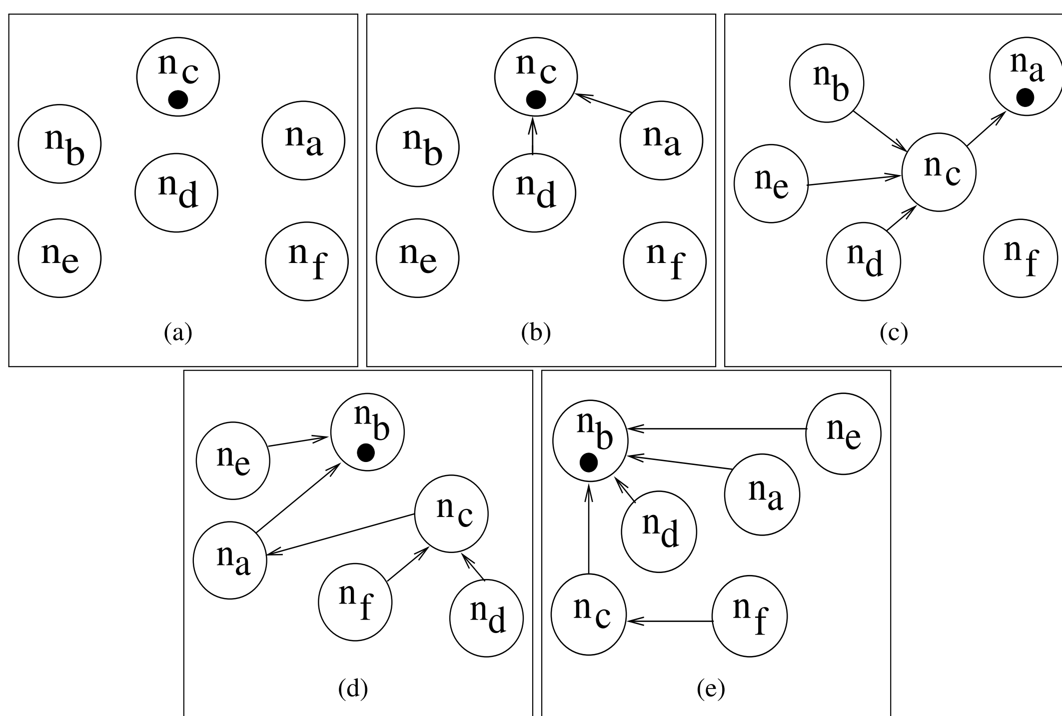


Figure 5.3. Example of updates to the logical DAG.
(a) n_c holds token. (b) n_c updates n_a and n_d. (c) n_a receives token from n_c. (d) n_b receives token from n_a. (e) n_b updates n_c and n_d.

When a node, that is not the token holder, receives a token request, the node inserts the new request in its local list of pending requests. Upon next opportunistic contact, the list is forwarded by the underlying routing protocol. If the node is the token holder, the request is

inserted into the request list. Figure 5.3 shows an example of creation and updates to the logical DAG. For simplicity, only six nodes are shown in Figure 5.3(a). Let n_c be the initial token holder and n_a, n_b, n_d, n_e and n_f be the requestors, where $a, b, c, d, e, f \in P$. Figure 5.3(b) shows that n_a and n_d consider n_c as the last known token holder. The token request message from the requestors might reach n_c at different times. If n_c is not executing its CS and it receives the first request, say from n_a, it will route the token to n_a. As mentioned above, the actual routing is performed by an underlying routing protocol.

5.2.3 Token propagation

Unlike traditional DAG based algorithms, in opportunistic networks the reverse path of request may not be the path taken by the token to reach the requestor, n_a. In Figure 5.3(c), n_a receives the token from n_c and hence n_c now points to n_a. After receiving the token, n_a executes its CS. After completing its CS, the token holder checks if its list of pending requests is empty. If the list is empty, the token is kept idle till the arrival of next request. If the list is not empty, the token holder selects the request that was generated earliest and tries to route the token to the corresponding node. Token request made by n_b is routed to n_c. n_c stores a copy of request made by n_b and then forwards the request to n_a. Figure 5.3(d) shows that, after leaving the CS, n_a routes the token to n_b. It might happen that n_c does not receive the token while it is being routed from n_a to n_b. Hence n_c still considers n_a to be the last known token holder. Similarly, n_d still points to n_c.

Intermediate nodes retain a copy of the token request. In the worst case, the token request reaches each and every node in the network. Hence in the worst case, N number of messages are required for reaching the token holder. In order to reduce the number of hops taken by the token request, other nodes in the network need to be informed about the change in location of the token. On one extreme, the network can be flooded each time the token is transferred. On the other extreme, no update messages are passed and the token request follows the path

along which the token was passed. Thus there is a tradeoff between the amount of flooding performed and the number of hops the token request is transmitted. MEOP uses the following limited flooding mechanism: when a node n_i transfers the token to another node n_j, only those two nodes have the accurate information about the location of the token. When n_i comes in contact with another node n_k, n_k is informed about the change in token location. n_k does not propagate the update to other nodes in the network. After receiving the token, the token holder enters its CS and may stop communicating with other nodes till the end of CS. After exiting the CS, the token holder also informs its neighbors about the location of the token. Only the sender and receiver of the token are allowed to update other nodes. It may happen that n_i and n_j never come in communication range of some nodes. These nodes will not receive the update message. Requests sent by these nodes, will either reach n_i (and then will be redirected to n_j) or some intermediate node, that has been updated, might redirect the request to n_j. Since only the sender and receiver of the token are allowed to update other nodes, a large amount of time is required to flood the entire network. Before the entire network gets flooded, n_j might transfer the token to some other node. Nodes that receive the recent update messages do not receive the older update messages. Hence entire network is flooded only in the worst case.

Figure 5.3(d) shows that n_b and n_e come in communication range and hence n_e points to n_b. Figure 5.3(e) shows that n_b still holds the token, but the topology has changed such that the token holder n_b comes in communication range of n_c and n_d. Hence n_c and n_d now consider n_b as the last known token holder.

If there are more than one pending requests in the request queue, the token holder chooses the requestor that had generated the request at the earliest. Hence requests that were made earlier in time, receive higher priority. An opportunistic network may not have a global clock or global time. Hence comparing two timestamps directly may not guarantee happened-before relation. Since clock synchronization is not the focus of this dissertation, we adopt the algorithm proposed in [68] to find the difference in clocks, and the clock drift. When a token request is

received, the receiver adjusts the timestamp of the request with the difference in clocks. Thus timestamps of requests received from multiple nodes are adjusted to the local clock.

Since token requests may be sent over multiple hops, the next highest priority node chosen by the token holder need not be a neighbor. The token may be transmitted over multiple hops to the next highest priority node. Since contacts between nodes are infrequent, the intermediate nodes are allowed to enter their CS only if the average inter-any-contact time of the intermediate node is more than the average time required to execute the CS on that node. Hence even though the token requests made by intermediate nodes do not have the highest priority, the time spent in waiting for an opportunistic connection is exploited to execute CS on intermediate nodes.

5.2.4 Fault detection

We consider two types of failures: (*i*) node n_i is waiting to transfer the token to a specific node n_j. n_i and n_j do not come in communication range due to their mobility patterns or due to crash at n_j. (*ii*) n_j receives the token from n_i and moves away from other nodes for a very long time or it has crashed. In both the cases, n_i performs the fault detection of n_j. We assume that n_i and n_j do not fail simultaneously. This assumption is similar to the one made in [52]. According to assumption 4, communication channel between two neighboring nodes is FIFO. If the devices move away from each others' communication range while the token request or token was being transferred, the error in transmission can be detected by both the nodes. Hence any incomplete transmission is cancelled by both the nodes. Thus token loss does not occur due to link failures.

Consider that node n_i is waiting to transfer the token to n_j. This case occurs when the routing protocol computes n_i to be the best router to n_j. n_i sets a timeout, $\tau_{i,j}$, for meeting with n_j. If n_i comes within communication range of n_j or a node that is a better router to n_j, n_j transfers the token and cancels the timeout. Else, if the timeout expires, n_i chooses the next highest priority request and sends the token to the corresponding node. A *SENDBACK*()

message is piggybacked along with the token. After the next high priority node completes its CS, the $SENDBACK()$ message makes the token come back to n_i. When the token comes back to n_i the process of waiting for n_j is repeated.

Consider that n_i transfers the token to n_j at time t. n_i sets a timeout, $\tau_{i,j}$, for meeting with n_j. If n_i and n_j come within communication range, and n_j still has the token, the timeout is reset to $\tau_{i,j}$. If n_j does not have the token, the timeout is cancelled and n_i stops performing fault detection of n_j. If n_i and n_j do not come within communication range before the timeout expires, n_i floods the network with $PING(n_j, t)$ messages. After receiving the $PING(n_j, t)$ message, a node n_x that has met n_j or any other node that was carrying the token (after time t) sends an $ISALIVE(n_j)$ message to n_i, else n_x sets a timeout $\tau_{x,j}$ for n_j. If node n_x meets n_j (or any other node that is carrying the token) before the timeout expires, n_x sends $ISALIVE(n_j)$ message to n_i, else n_x sends a $NOTALIVE(n_j)$ to n_i. If n_i receives at least one $ISALIVE(n_j)$ message, it resets the timeout to $\tau_{i,j}$ and again starts waiting for n_j. If n_i receives $NOTALIVE(n_j)$ messages from all other nodes, it considers n_j to have failed and generates a new token. Due to the opportunistic nature of the network, it may happen that n_i has started a timeout for n_j and n_i meets another node n_k that currently has the token or n_i might receive the token from some other node. In such cases, n_i cancels the timeout for n_j.

We set $\tau_{i,j}$ to be the maximum of observed intercontact time between the two nodes. According to assumptions 1, 2 and 7, the inter contact time between two nodes follows a distribution and the distribution has reached steady state. Hence the probability that n_j is alive and does not meet n_i during multiple timeouts becomes almost negligible. If a token loss is detected, n_i generates a new token.

It could happen that a node n_i generates a token request, propagates the request to another node n_j and n_j crashes. After transfering the request to n_j, n_i sets a timeout of some fixed value. If n_i does not receive the token before the timeout expires, n_i retransmits the same request to

other nodes. Hence the token request will eventually reach the token holder even in presence of node failures.

5.2.5 Correctness properties

5.2.5.1 Safety

MEOP is a token based algorithm. Hence safety property is guaranteed if there is at most one token in the network. Failures can occur in two situations: node n_i transfers the token to n_j and n_j faces software or hardware failure and crashes, or n_j moves away from all other devices for a very long time. In both the cases, n_i experiences timeout and generates a new token. We set the timeout $\tau_{i,j}$ to be the maximum of observed intercontact times between the two nodes. Since the distribution has stabilized, the probability that n_j meets n_i after time greater than $\tau_{i,j}$ is very small. Moreover, after the timeout expires, n_i floods network looking for n_j and the token. If other nodes also have not met n_j or the token, they set timeouts with corresponding maximum intercontact time values. n_i declares n_j to have failed only when all nodes have not met n_j and the token for the maximum observed intercontact time. The probability that n_j is alive and does not meet any node during corresponding timeouts is extremely small and so n_j is considered to have failed.

Consider that n_i starts waiting for n_j at time t. Let $T_{i,j}$ be the random variable that denotes the intercontact times between nodes n_i and n_j and $max(T_{i,j})$ be the maximum observed value. The distribution of $T_{i,j}$ might be non zero till infinity. Since the intercontact time distribution is either heavy tailed or exponential, the probability that n_i and n_j meet after time $max(T_{i,j})$ is very small. Let $\delta_{i,j} = P(T_{i,j} > max(T_{i,j}))$ be the probability that n_i and n_j do not meet before time $max(T_{i,j})$. If the two nodes do not meet in $max(T_{i,j})$, n_i broadcasts $PING(n_j, t)$ message. Let $\theta_{i,x}$ be the time taken for the $PING()$ message to reach from node n_i to node n_x. n_x sends an $ISALIVE(n_j)$ message if n_x has met either n_j or any token holding node after time

$[t, t + max(T_{i,j}) + \theta_{i,x}]$. Otherwise, n_x starts a timeout with value $max(T_{x,j})$. If n_x does not meet either n_j or any other node with the token in $max(T_{x,j})$, n_x sends $NOTALIVE(n_j)$ message to n_i. The $NOTALIVE(n_j)$ message transmitted by n_x can be a false negative only if n_j has not crashed, has the token and does not meet n_x in time interval $[t, t + max(T_{i,j}) + \theta_{i,x} + max(T_{x,j})]$. Let $P(FAIL_{x,i,j})$ be the probability of a $NOTALIVE(n_j)$ message from n_x to n_i. This probability is given by:

$$P(FAIL_{x,i,j}) = \delta_{i,j} P(T_{x,j} > max(T_{i,j}) + \theta_{i,x} + max(T_{x,j})) \tag{5.1}$$

If node n_i receives atleast one $ISALIVE(n_j)$ message, it repeats the process of waiting for n_j. n_i decides that n_j has failed only after receiving $NOTALIVE()$ from all other nodes. Let $\theta_{x,i}$ be the time taken by the reply, sent by n_x to reach n_i. Hence at time $(t + t_d)$, n_i will decide that n_j has failed, where t_d is given by:

$$t_d = max(T_{i,j}) + \max_{\forall x \neq i, x \neq j} (\theta_{i,x} + max(T_{x,j}) + \theta_{x,i}) \tag{5.2}$$

Consider that n_i has received $NOTALIVE(n_j)$ message from all other nodes in the network. If n_j has not crashed, has the token and meets any of the nodes after the corresponding maximum intercontact time, all the received $NOTALIVE(n_j)$ messages would become false negatives. Hence the probability that n_i incorrectly declares n_j to have failed is given by:

$$\begin{aligned} P(FAIL_{i,j}) &= P(T_{i,j} > t_d) \\ &\prod_{\forall x \neq i, x \neq j} P(T_{x,j} > max(T_{i,j}) + \theta_{i,x} + max(T_{x,j})) \end{aligned} \tag{5.3}$$

In order to recover from the crash, n_i generates a token. If n_j has not crashed and has the token, n_j itself can observe that it has not met any node during the corresponding maximum intercontact times, and hence can infer that n_i has regenerated a token. Hence n_j deletes its

token. Thus there is only one token in the system. Thus our algorithm satisfies the safety property.

5.2.5.2 Starvation

Each token request is propagated in the network till it reaches the token holder. In the worst case every node will have a copy of the request. Hence the token request is guaranteed to reach the token. Each token request has a timestamp. The request that was made earlier than other request, receives a higher priority. As explained above, there is always atmost one token in the network. Thus the algorithm is free from starvation.

5.2.5.3 Deadlock

When a node n_i is waiting to transfer token to n_j, no other node in the network can enter its CS. To show that our algorithm is free from deadlock, it is sufficient to show that n_i does not indefinitely wait for n_j. Infinite waiting time is avoided by using time $\tau_{i,j}$ as explained in previous section. After the timeout, n_i sends the token to the next high priority node and receives the token back. n_i then repeats the process of waiting. Hence at least one node in the network can enter its CS. Thus our algorithm is free from deadlock.

5.2.5.4 Ordering

In an opportunistic network, messages are transmitted over multiple hops. Due to location and mobility of nodes, even if two token requests are generated at the same time, the requests might reach token holder at different times. Hence total ordering cannot be guaranteed. When token holder has multiple pending requests, it chooses the node that generated the request earlier. Hence there is ordering between received requests. However, requests that have not been received by the token holder cannot be ordered according to their timestamps. In MEOP, when

the token holder selects the next high priority node, say n_i, it transfers the token to that node, possibly using multi hop communication. Intermediate nodes might have generated token request after n_i had generated the request. To maintain strict ordering, intermediate nodes should not be allowed to enter their CS. In order to decrease the average waiting time, MEOP allows intermediate node, say n_j, to enter its CS only if $E[T_j] >$ average time required to execute the CS at intermediate node. Thus MEOP satisfies the safety and liveness properties.

5.3 Simulations for mutual exclusion

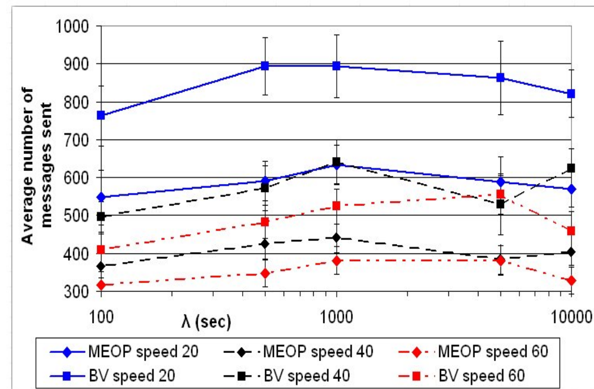
To simulate an opportunistic network, we use Random Waypoint mobility model for results shown in Figures 5.4, 5.6 to 5.9. For results shown in Figure 5.10, the *cambridge/haggle* dataset [69] was used. For the random waypoint mobility model, we use the *mobigen-ss* tool [70] to generate the steady state of node location and speed. Hence our simulations start in a steady state. The simulations use 10 nodes moving in an area of 300m x 300m. The average communication range is 10m. We compare MEOP with algorithms proposed in *BV* and [55]. Performance of MEOP is also compared with that of a centralized algorithm.

We also have collected Bluetooth sightings between 5 students at the Computer Science and Engineering department of University of Texas at Arlington (UTA). The students move around in the campus according to their respective daily schedules. The students move in an area of 1.6 Km x 1.6 Km. The students were chosen randomly. Students and hence their devices came within each other's communication range at such places as labs, classrooms, cafes and dinning areas. Each student carried a Bluetooth enabled device (a Toshiba netbook or Fujitsu laptop). The program for collection of Bluetooth sightings is implemented using Java and the Bluecove 2.1.0 bluetooth library [71]. The students were allowed to start, stop the process manually. If a device is low on residual battery, the student may stop the process. Also, if the student does not expect his device to be within communication range of other participants,

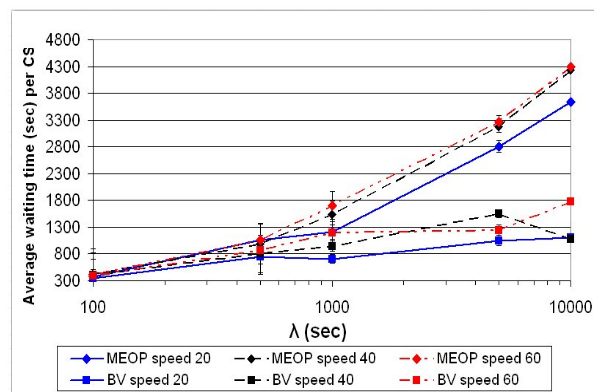
the student may stop the process to avoid performing unnecessary device discovery attempts. Allowing students to manually start and stop the process is realistic due to the fact that while using personal devices, the users might need control over their devices. Hence there were sightings where the process was not running, but the devices were within communication range.

The simulations use intercontact time as routing criteria. When n_i wishes to transfer a message to n_j, and they are not in the communication range, n_i transfers the message to another node n_x only if $E[T_{x,j}] < E[T_{i,j}]$. The routing protocol can be changed to any existing protocol without affecting MEOP algorithm. For request generation, we use variations similar to that used in [22]: after a node has executed its CS, the node generates a new request after an average delay of λ seconds. We vary λ from 100 seconds (heavy load) to 10000 seconds (light load). *BV* and [55] algorithms do not handle node failures and hence do not avoid starvation and deadlock in case of failures. Hence, to perform fair comparisons, node failures are not simulated. In following simulations, each reading is an average of 50 executions. Confidence intervals are computed with 95% confidence level.

Figures 5.4(a) and 5.4(b) show the average number of messages transmitted in the network and the average waiting time per CS with varying λ. Average time required to execute the critical section is 30 seconds. The average speed is varied from 20 m/s (low speed) to 60 m/s (high speed). All of the nodes generate requests. We see that the number of messages transferred first increases and then decreases with λ, whereas the average waiting time per CS continuously increases. This behavior is explained as follows. When $\lambda = 100$, after completing a CS, the token holder generates the next request much faster than that when $\lambda = 10000$. Nodes in an opportunistic network infrequently come in contact with each other. Hence when $\lambda = 100$, the token holder can satisfy many of its own CS requests before meeting another node. Thus, while waiting for an opportunistic contact, the token holder can satisfy requests (generated by itself) that do not have the highest priority in the pending queue. It may happen that a node comes in contact with the token holder when the token holder is executing its non-highest priority CS,



(a)



(b)

Figure 5.4. Effect of varying λ. (a) Average number of messages transmitted. (b) Average waiting time per CS request.

and thus the opportunity to forward the token to the actual highest priority node is lost. Thus the number of messages sent and the average waiting time per CS is less. When the delay between end of CS and generation of next request is $\lambda = 1000$, there is a lesser probability of losing an opportunity to forward the token. Hence number of messages transmitted increases. As the token is transferred to another node, next token request generated on previous token holder has to wait a larger amount of time. Hence average waiting time also increases. When $\lambda = 10000$, requests are generated at large time intervals. In MEOP, the token is less frequently transferred to other nodes due to possibly empty pending queue at the token holder. BV uses a token ring. Hence, even if a node does not need to execute CS, it still gets the token. In traditional token

rings over a static network, the token would have been immediately passed onto the next node. In an opportunistic network, the token holder has to wait for an opportunistic contact. Hence the average waiting time increases. Trends in Figures 5.6(a) and 5.6(c) also show the same behavior - average waiting time increases with increasing λ and the number of messages transmitted first increases and then decreases with increasing λ.

MEOP is based on a logical DAG. Token is transferred to a node, say n_i, only when its request has reached the token holder (and when the node is an intermediate node on the route to n_i). BV is a token ring. Token circulation is started as soon as the token holder (or coordinator) receives the first token request. Hence a node n_i might receive token even though its request has not reached the coordinator. Hence the average waiting time of BV is less than that of MEOP. In BV, the token is circulated over all nodes, even if some of the nodes do not wish to execute CS. Hence the number of messages transmitted in BV is more than that in MEOP.

Figure 5.5 shows the comparison of MEOP with a permission based mutex algorithm [55] and a centralized algorithm. We first describe the centralized algorithm implemented for the simulations. In the centralized algorithm, at the start of simulations, a node is assigned the role of central server. All other nodes that need to enter CS send request to the server. The server selects the earliest request from the server's queue for pending requests. The server then sends a *permission* message to the corresponding node. The requestor node receives the permission and enters the CS. After completion of the CS, the requestor node sends a *release* message to the server. After receiving *release* message, the server selects the another pending request from the queue. Hence the *permission* and *release* messages cannot be in-transit simultaneously in the network. Communication between server and other nodes may be single or multihop. Consider a node n_i that is being used as a relay during mutihop communication of the *permission* or *release* message. If n_i also needs the to enter its CS, n_i may execute its CS only if $E[T_i]$ is more than the CS execution time. Consider that n_i was being used as a relay for *permission* message destined for node n_j. If n_i executes its CS, upon completion of the CS, n_i forwards the

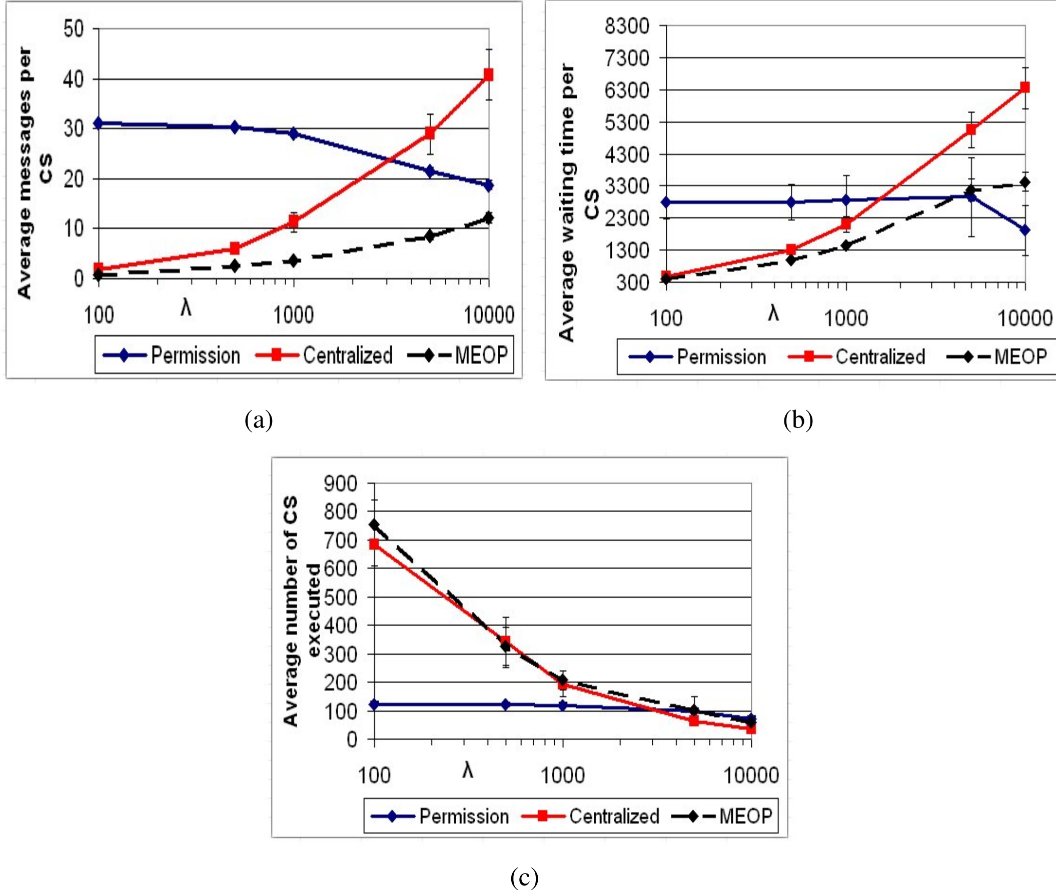


Figure 5.5. Comparison of MEOP with permission based and centralized algorithms. (a) Average number of messages transmitted per CS. (b) Average waiting time per CS request. (c) Average number of CS executed.

permission message towards n_j. The information that n_i has completed its CS is piggybacked on the *permission* message. When n_j sends a *release* message to the server, information about which intermediate nodes (such as n_i) have completed their CS is piggybacked on the *release* message. Consider that n_i is being used as a relay for the *release* message. If n_i executes its CS, n_i adds to the information sent along with the *release* message and sends the message towards the server. In [55], a node that needs to enter CS, sends request to nodes in its *info_set* and waits for permission from each node in the set. Intermediate nodes used during routing of permissions cannot enter their CS. That is only the node that is chosen to receive all permission

messages can enter its CS. Hence in Figure 5.5(c), we see that as λ increases, the number of CS executed decreases drastically in MEOP and centralized algorithms as compared to [55]. Figure 5.5(a) shows that as λ increases, number of messages required per CS increases in MEOP and centralized scheme, whereas the number of messages decreases in [55]. Figure 5.5(b) shows that as λ increases, average delay per CS increases in MEOP and centralized algorithms and decreases in [55].

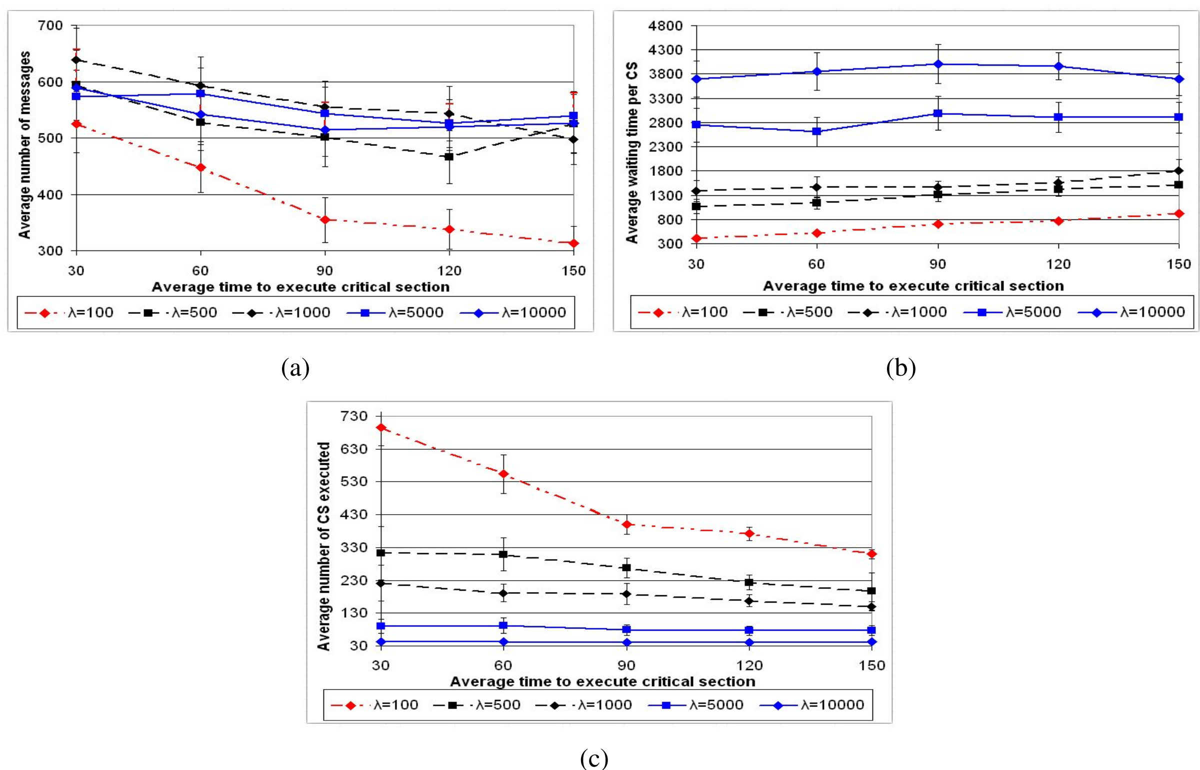


Figure 5.6. Effect of varying average time required to execute CS.
(a) Average number of messages transmitted. (b) Average waiting time per CS per node. (c) Average number of CS executed.

Figures 5.6(a), 5.6(b) and 5.6(c) show the effect of varying the average time required to execute CS on number of transmissions, average waiting time and number of executed CS respectively. The effect of varying λ is as explained above in Figure 5.4. When the time

required to execute CS is increased, the token holder spends more time executing CS and lesser time waiting for an opportunistic contact. Hence the number of messages transmitted decreases, average waiting time increases and the number of executed CS decreases with increase in CS.

The routing protocol used for above simulations can be changed without affecting the MEOP algorithm. We performed simulations to study the effect of varying λ when the PRoPHET [72] routing protocol is used and observed that the trends remained same as obtained in previous simulations.

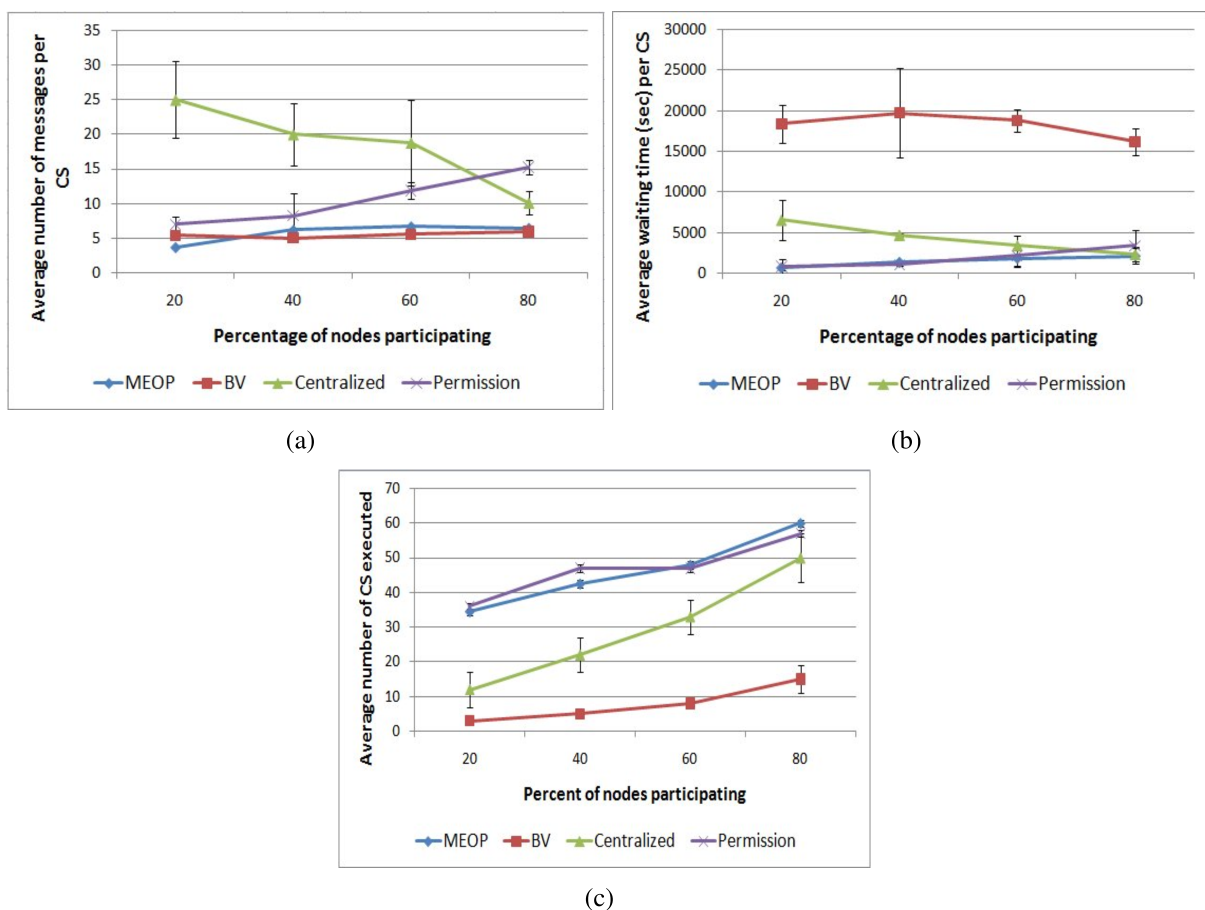
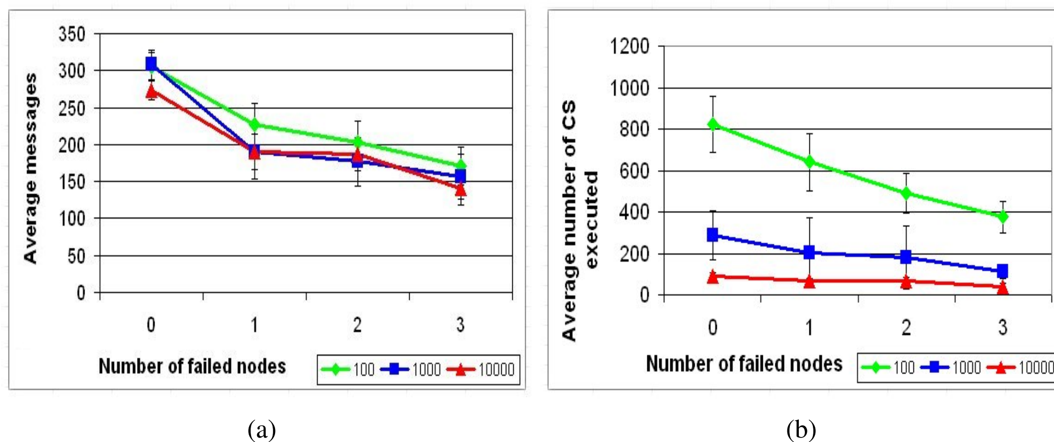


Figure 5.7. Effect of varying percentage of nodes that generate token request. (a) Average number of messages transmitted per CS. (b) Average waiting time per CS. (c) Average number of CS executed.

In the above simulations, all of the nodes in the network made token requests. Figure 5.7 shows the effect of varying percentage of participation. We see that MEOP has lower communication overhead irrespective of the number of nodes making token requests. In a token ring algorithm, the token is sent to all the nodes even if only a few nodes require the token. In MEOP, only those nodes that generate token requests and intermediate nodes used as relay for multihop communication receive the token. Hence the average waiting time per CS request is lower in MEOP when few of the nodes generate token requests. When the participation increases, each of the node either generates a token or is involved in multihop communication. In the BV algorithm, if the token has started to propagate over the ring, nodes receive and enter their CS even if the token request has not reached the coordinator. In the MEOP algorithm, a node that is not used as a relay during multihop communication does not enter its CS till the corresponding token request reaches the token. Hence BV has a lower average delay per CS when the participation is higher.

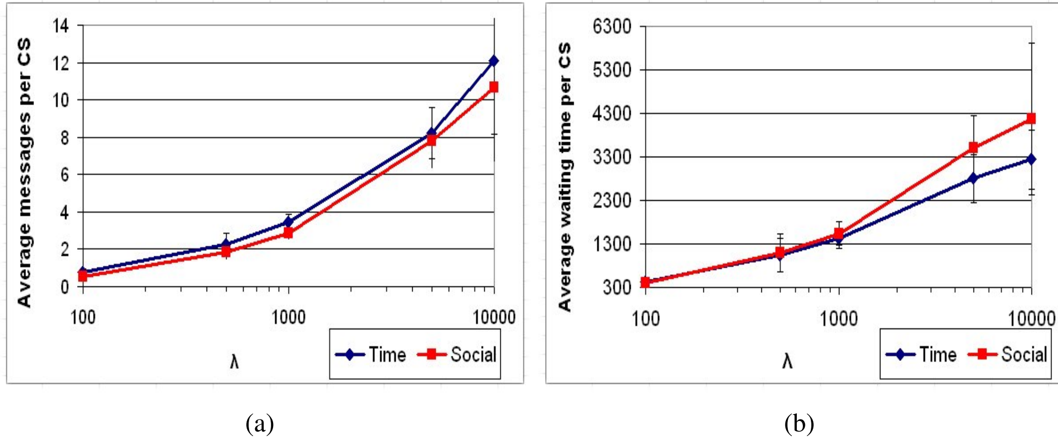


(a) (b)

Figure 5.8. Effect for device failures on MEOP. (a) Average number of messages transmitted per CS. (b) Average number of CS executed.

In the above simulations, the devices were considered to be reliable, that is, there were no device failures. In the simulations for Figure 5.8, we see the effect of device failures and

the fault tolerance algorithm described in Section 5.2.4. Figure 5.8(a) shows that the number messages transmitted in the network decreases with increase in failures. Figure 5.8(b) shows that the number of CS executed also decreases with increase in failures. The decrease in work done, that is number of CS executed, is due to the time spent by nodes performing fault detection of the failed node. No node can enter its CS from the time a token holder fails till the time the token is regenerated.



(a) (b)

Figure 5.9. Time and community based selection for token destination. (a) Average number of messages transmitted per CS. (b) Average waiting time per CS.

As explained in Section 5.2, each node stores a list of pending requests. When a token holder completes its CS, the token is passed (using possibly a multihop route) to a node that had made the earliest request. Let ζ_a and ζ_b be the time at which nodes n_a and n_b generated request for token. $\zeta_a < \zeta_b$ implies that n_a made the request before n_b. The priority of the requests was computed as $1/\zeta_a$ and $1/\zeta_b$. In simulations performed for Figure 5.9, we consider social aspects for selecting the next node to enter CS. Since social computing is not the focus of this dissertation, we adopt the following simple approach. Each node maintains a set, S_0, named *friends*. A node is selected as a friend if the intercontact time between the two nodes is less than a threshold. Nodes that are not in the friends list of a node n_a are classified into lists

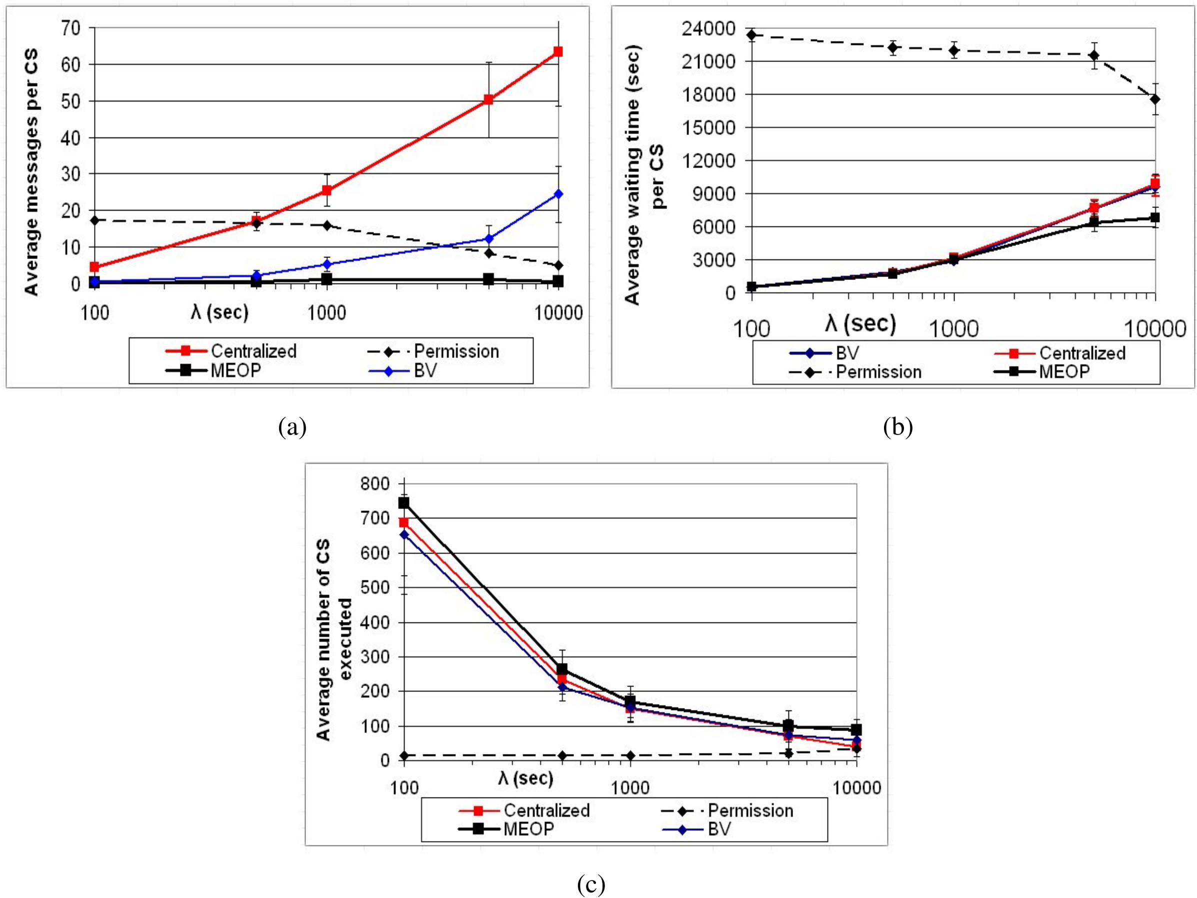
S_1: *friend-of-friend*, S_2: *friend-of-friend-of-friend*, and so on. Nodes that are not in the above lists, but have been in communication range of n_i are classified into list S_a: *acquaintances*. Nodes that are not present in any of the list and have been in communication range are classfied into list S_w: *world* list. Token requests from friends are given higher priority than those from friend-of-friends. Token requests from friend-of-friend are given higher priority than friend-of-friend-of friend and so on. Token requests from nodes in acquaintances and world are given lesser priority with world receiving least priority. To avoid starvation, the priority is computed as $1/(\alpha_x \zeta_i)$, where ζ_i is the time at which request was generated by node n_i and $1/\alpha_x < 1$ is the importance of list to which n_i belongs. $\alpha_0 < \alpha_1 < \alpha_2 < ... < \alpha_a < \alpha_w$. Hence eventually each request will become the request with highest priority. From Figure 5.9, we see that social networking concepts can be used to decrease communication overhead and average delay per CS.

We have also compared performance of MEOP with that of other algorithms while using real world traces. For simulation results shown in Figure 5.10, the *cambridge/haggle* dataset [69] was used. This dataset includes bluetooth sightings made by 12 students carrying iMotes for six days. We see that MEOP has much less communication overhead as compared to other algorithms.

Simulations performed for Figure 5.11 use the trace of Bluetooth enabled devices at UTA. We see that MEOP has lower communication and time overhead as compared to other algorithms.

5.4 Summary and discussion

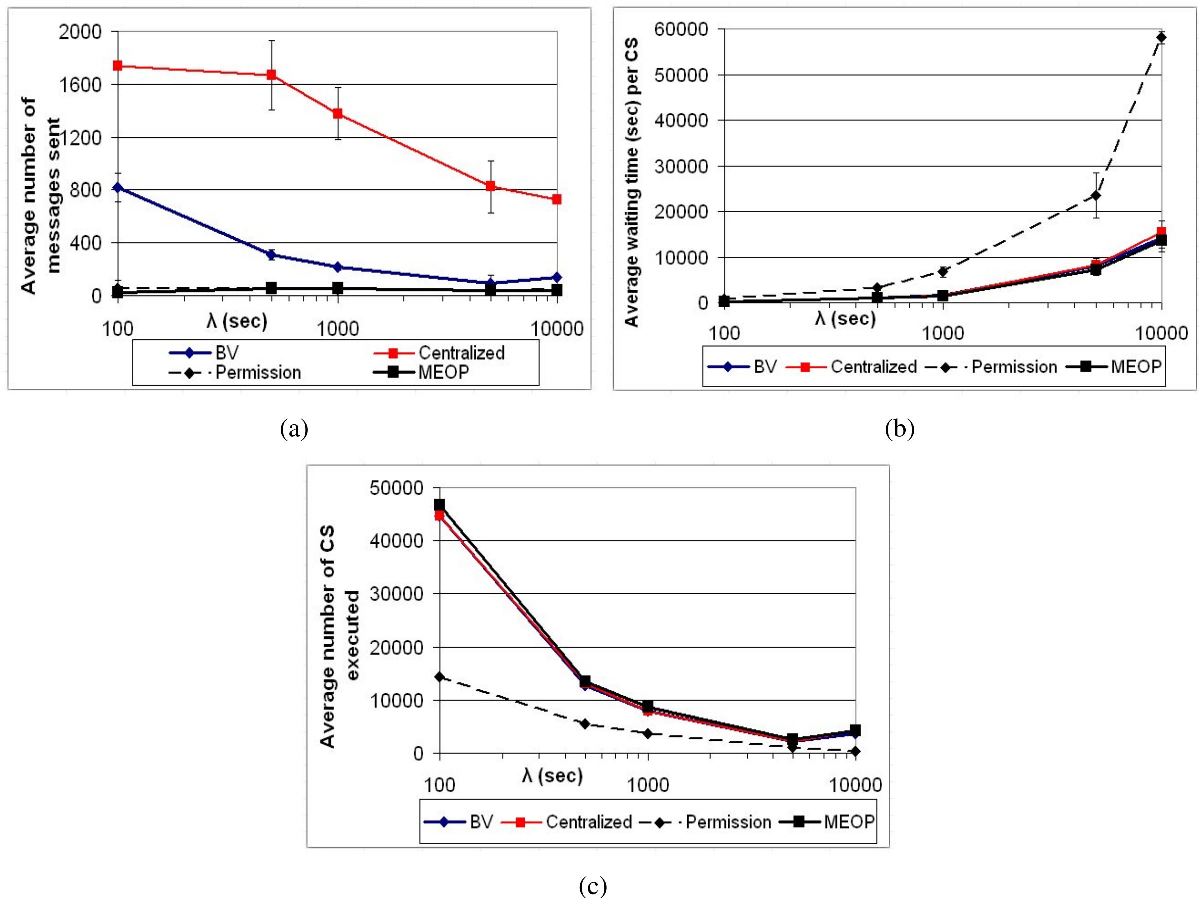
Mutual Exclusion is a fundamental problem in distributed systems. Nodes in an opportunistic network meet infrequently and are resource constrained. In such a dynamic and distributed environment, nodes requesting exclusive access to a shared resource might face star-



(a) (b) (c)

Figure 5.10. Effect of varying λ, on the Haggle dataset.
(a) Average number of messages transmitted. (b) Average waiting time per CS request. (c) Average number of CS executed.

vation and deadlock due to node mobilities and failures. In this chapter, we first discussed the applicability of some of the existing algorithms available for MANETs. We presented a novel algorithm, Mutual Exclusion for OPportunistic networks (MEOP) that satisfies safety and liveness properties. The algorithm uses a logical DAG for passing the token requests. MEOP is independent of the underlying routing protocols and hence does not need continuous monitoring of the entire network. We proposed a timeout based fault detection mechanism in which the distribution of intercontact times is exploited to decide the timeout values. Simulation results show that MEOP is communication efficient.



(a)

(b)

(c)

Figure 5.11. Effect of varying λ, while using trace from UTA.
(a) Average number of messages transmitted. (b) Average waiting time per CS request. (c) Average number of CS executed.

Social networking concepts have been used to route messages between nodes in an opportunistic network [5]. We have implemented as simple heuristic to exploit social relations between people. We envisage adding more heuristics in order to reduce the average waiting time per critical section request. Similar to the assumption made in [52], MEOP assumes that when a node is tracking the presence of a particular token holder, at most one of those two nodes fails. We are extending MEOP to handle multiple simultaneous failures.

CHAPTER 6

SERVICE COMPOSITION IN OPPORTUNISTIC NETWORKS

An opportunistic contact between two mobile devices takes place when they are within communication range of each other. A series of opportunistic contacts in time and space lead to an opportunistic network. In such a system, meeting application requirements and managing resources efficiently is a challenge. In the recent past, researchers have exploited opportunistic links to share content, leading to a wide variety of applications in entertainment, health care, transportation, military etc [73]. However, there has been no prior work in developing a middleware architecture for opportunistic networks. In this chapter, we propose a modular middleware architecture that facilitates application requirements and resource management in opportunistic environments. In particular, we demonstrate the utility of the middleware for fault tolerant service composition in such environments. An in-depth analysis of the service composition process is also presented. A prototype of the service composition process has been implemented. In the prototype, users carry Bluetooth enabled devices in the campus of University of Texas at Arlington. Service execution requests are exchanged when the devices opportunistically come within communication range.

6.1 Introduction

Opportunistic networks are created when pairs of devices communicate through many opportunistic contacts [74]. In an opportunistic environment, applications executing on devices carried by users may need various kinds of resources. Devices such as cell phones, PDAs, laptops host services that are useful to other devices in the network. Examples of such services are: data compression / decompression, data encoding / decoding, audio / image / video processing,

and others. A service might be available on multiple devices. A device can host multiple services. Furthermore, user devices and sensors spread in the environment might carry or supply different kinds of information, that is useful to other users and applications. Such resources may be available within the opportunistic network, but not within direct communication range of the requesting device. An end-to-end path between the requesting device and the required resource may never exist. Hence, it is necessary to make resources accessible and available anywhere in the environment, perhaps with some delay. Enabling collaborative application processing is extremely challenging in opportunistic networks. In this chapter, we propose a generic framework to enhance the availability of all resources in the network to clients and their applications. In effect, the framework exploits opportunistic pair-wise contacts to enhance the availability of the services on one hand and quality of user interaction on the other. The framework essentially creates a platform for collaboration among devices. Pair-wise meeting of devices can be opportunistically exploited to reach services running on other parts of the network, which would not be possible in traditional networking schemes as the requesting application may never get in touch due to its users' mobility pattern. Service results will be provided to the requesting application either directly during contact, or through an opportunistic path found in the network.

When an application needs to perform a service, it generates a service execution request. The device on which this request was generated is henceforth called as a client. The client first checks whether it has hosted the required service. If the client has not hosted the required service, the client checks if the service is hosted by any other device in the network. The process of gathering information about services hosted by other devices in the network is called as service discovery. To facilitate searching, each device maintains an index of services discovered. The amount of information recorded in a device depends on its capabilities. Suppose a requested service is available neither with the client nor with any other device in the network. In such case, the client performs service composition and checks if multiple services can be concatenated together to form the required service. For example: consider that the client is aware of the

locations of two services, conversion of a PDF file to speech and conversion of text file to PDF. An application on the client makes a request to convert text to speech. Since there is no service directly corresponding to the required one, the middleware installed on the client can first execute text to PDF and then send the output to the PDF to speech service. Thus a composite service is created. This process is called as service composition.

Section 6.2 describes the proposed middleware. In Section 6.3, we describe the exchange and updation of device connectivity information. Section 6.4 describes the service composition algorithm and presents analysis for success probability and length of composition. Section 6.5 explains the replicated service execution approach proposed for fault tolerance. Section 6.6 details the simulations. We have implemented a prototype of service composition algorithms. Section 6.7 describes the prototype and compares the results with analysis and simulations.

6.2 Proposed middleware

Figure 6.1 illustrates components of the proposed middleware. The network handler module performs the actual transmission and reception of messages using one or more types of wireless communication capabilities such as Bluetooth, 802.11, etc. The logger module is a utility module that can be used by other modules to record events. The resource monitor module periodically checks and records resource levels of the device. Information such as residual battery, CPU utilization, memory utilization are useful to the modules in the processing and application tiers. The event detector module performs sensing of environment, for example, change in light intensity, motion sensing. The event detector receives context rules from the context manager module in the processing tier. Whenever a specified context rule is satisfied, the event detector module informs the context manager module. The location module incorporates facilities such as location prediction using GPS, received signal strength, etc.

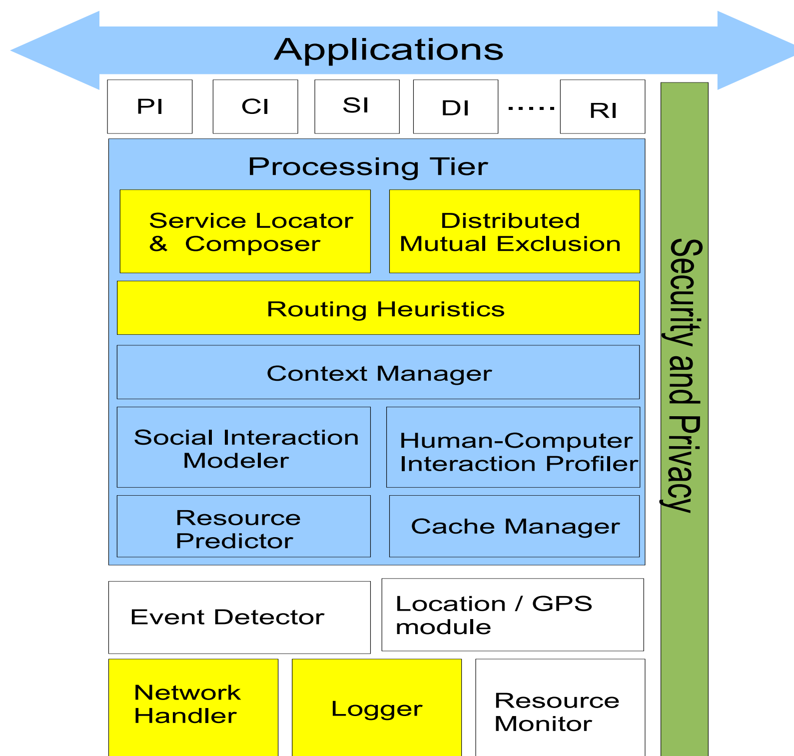


Figure 6.1. Middleware Architecture.

The processing tier is responsible for processing various kinds of information received from the device, the applications running on the device and the devices that come in contact opportunistically. The processing tier is composed of multiple modules as shown in Figure 6.1. It is to be noted that the processing tier is an optional tier. Primitive devices such as a mote sensor might not have the processing tier. Such a device might have only an ID and sensory data (e.g., temperature). The proposed middleware is modular, the processing unit may vary from basic sensing and communication abilities in a sensor to a full set of software suites for carrying out such tasks as aggregation, optimization, resource management and/or other tasks. The cache manager enables information caching and can be used by other modules. The resource predictor module implements algorithms to predict the resource levels at a future time instant. Such predictions are useful for knowing whether a device will successfully complete a given

service. The resource predictor also provides the expected life of the device under device's current usage pattern and resource levels. Examples of such algorithms are CABMAN [75] and history based online prediction [76]. The social interaction modeler defines rules about social standing of the device (or the owner of the device). It defines metrics such as popularity of a device, contributions made by the device to other devices in the network, communities that a device is part of. The social interaction modeler also uses the event detector module to know the ongoing events. BubbleRap [5] uses such social interaction data to perform informed routing in delay tolerant networks. The Human-Computer Interaction (HCI) module is used to record the interaction between a user and his device. Since the opportunistic network might contain personal devices, service execution should not inhibit the user from performing his tasks. For example, on a cellphone, an incoming phone call receives higher priority than a currently executing service. Hence completion of execution of a service may be delayed depending on the amount of interaction between user and her device. Thus, to provide QoS information for service execution, the middleware needs to profile HCI. Eagle et al. [16] have demonstrated the profiling of interaction between users and applications on mobile devices. The context manager module implements rules that define such events as: a person is walking, or running, or device has entered a specified location. It uses information recorded by the event detector module. The routing module implements various algorithms for routing of messages in the opportunistic network. This module utilizes knowledge gained by the context modeler, social interaction modeler and event detector. The routing module consists of algorithms such as Hibop [43]. Users or devices may need exclusive access to shared resources in the opportunistic network. Hence the middleware needs to provide mechanisms for enabling mutual exclusion. The distributed mutual exclusion module implements algorithms such as MEOP [21] which is a token based algorithm for providing mutual exclusion. The security and privacy module provides algorithms for encryption, decryption of messages, and trust profiling of the devices in the network. A user uses information from the social interaction modeler and sets access control to

information, files stored on the device. For example: friends and family might be allowed to see detailed information about the user and his device. Such privacy settings are offered by the security and privacy module.

Above the processing tier, each device has a set of optional index buffers. The personal information index (PI) contains basic user and device information such as the ID of the device, and name, address, work, home information of the user. The content index (CI) indicates the set of information objects the device (and possibly its community) is willing to share with other users/applications. The service index (SI) contains information about services discovered by the service locator and composer module. This module also adds the most frequently used service compositions to the service index. The Device Index (DI) is used to store information about the communication cost and inter contact times between the devices. Creation and updation of the DI is explained in detail in Section 6.3. The reputation index (RI) contains information about reputation and trust values of devices in the network. The security and privacy module updates RI. The Index buffers are optional varying from a simple source of sensory information to complex graph based indices for service composition.

When two devices come within communication range, they can exchange information during the contact. This exchange is divided into two phases. The first phase consists of a handshake, during which the two devices exchange basic information such as the identity of the device/user and device information (e.g., Nokia 3362 cell phone, 2 MB memory, camera equipped, Bluetooth capable) contained in the PI buffer. The information exchanged during handshake is subjected to privacy settings. When a user comes in communication range of friends, family, other trusted people, more information is made visible to the devices. The second phase starts after the end of handshake. During this phase, the processing tier within the user's device makes a decision on whether it needs to request more detailed information such as the content index (CI), service index (SI) from the other device. Due to limited cache and index space, the user's device might remove information that was least recently used or was

received long time back. If the user's device had previously met the other device and has all the previously received information, only updates to the stored information are exchanged.

6.3 Pair-wise contact

This section describes the creation and updation of the device index (DI). Consider a graph representation of the network where each device is denoted by a node and the contact between two devices (or nodes) is represented by a link or edge between the two nodes. The edge weight represents the expected time of contact between the two nodes. The edge weight can also represent other parameters such as the total contact duration, commonality (e.g: belong to same school), privacy level, or a combination of parameters. Without loss of generality, we use the expected intercontact time as edge weight in the rest of this chapter. We represent the pairwise contact between two nodes n_a and n_b by $n_a \Leftrightarrow n_b$, and the expected time of contact by $E_{a,b}(T)$.

Consider node n_a, where $a \in (1, 2, ..., M)$ and M is the total number of nodes in the environment. The terms, device and nodes are used interchangeably. Suppose a device n_a is expected to connect to only a subset of the devices in the network. This set of devices is called as a connectivity set M_a of device n_a. Hence for each device in M_a, the corresponding node has an edge to n_a. The connectivity set M_a indicates the set of nodes that can reach each other by using n_a as the intermediate node. The combination of such simple paths at n_a is represented by the graph G_a^1 as shown in Figure 6.2(a). In Figure 6.2(a), n_a and n_p meet with expected time $E_{a,p}(T)$ and n_a and n_r meet with expected time $E_{a,r}(T)$, therefore the expected time for an application on n_r to access a resource on n_p is given by $[E_{a,p}(T) + E_{a,r}(T)]$. A sample of a similar graph G_b^1 for node n_b is shown in Figure 6.2(b). When two nodes n_a and n_b meet, they can exchange each others' one-hop connectivity sets and the respective graphs stored in the Device Index (DI). In other words, n_a receives (M_b, G_b^1) from n_b, which in turn receives (M_a, G_a^1) from

n_a. As shown in Figure 6.2(c), a combined graph $G^2_{a,b}$ is created, on both the devices, by merging the corresponding edges of G^1_a and G^1_b. The combined graph shows all the paths from n_i to n_j, that use n_a and n_b as intermediate nodes. This combined graph also is stored into the DI of nodes n_a and n_b.

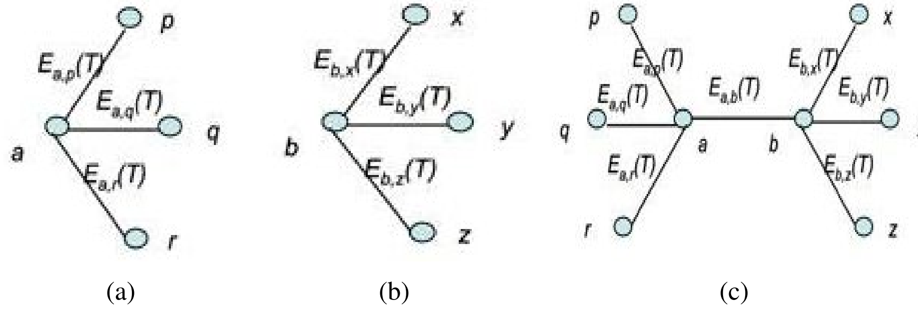


Figure 6.2. Sample graphs at nodes n_a and n_b created during pairwise contacts. (a) G^1_a: paths via n_a (b) G^1_b: paths via n_b (c) $G^2_{a,b}$: paths via n_a and n_b.

When a combined graph is created at a node, the host node can compute the expected time to access the resources on various nodes that can be reached through pair-wise contacts. Once reachability is established, the nodes can request details of a particular type of soft-resource and facilitate the pull operation to get the resource.

6.4 Service composition

This section explains the service composer module. Each device hosts certain services and maintains a service graph of the locally hosted services. This service graph is stored in the service index (SI). When two devices meet opportunistically, they exchange their service graphs and create a composite service graph which has information about locally as well as remotely available services. Let $G_x(s)$ and $T_x(s)$ represent the graph of available services and graph of services requested at node n_x. When n_a and n_b meet, they exchange their respective services and an aggregated graph $G(s)$ is created at each node. The aggregated graph is processed for finding

or composing a service corresponding to required task. While the two nodes are in contact, collaborative services are composed and executed using the basic services represented in $G(s)$.

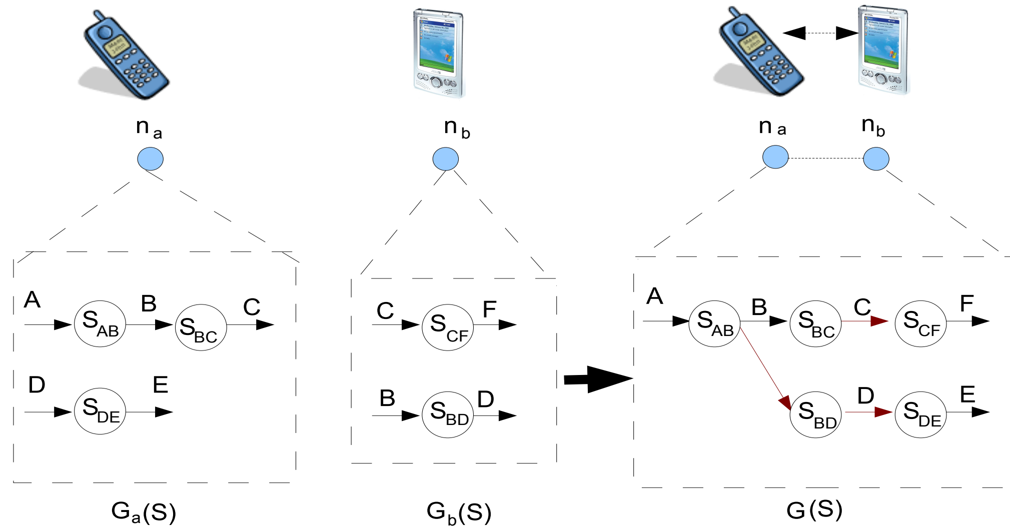


Figure 6.3. Service Composition.

In the example of Fig. 6.3, n_a is required to compose a service to meet the task request $T_a(s) : A \rightarrow F$. Using the aggregated graph $G(s)$, n_a identifies the path from A to F by using services s_{AB}, and s_{BC} on itself and s_{CF} on n_b. In order to execute the composed service, the output parameter C of service s_{BC} is passed on as input parameter to service s_{CF}, on n_b. The output parameter F of s_{CF} is passed from n_b to n_a. Likewise, suppose n_b desires to complete a task $T_b(s) : B \rightarrow E$. The graph $G(s)$ is available at both n_a and n_b. Since $T_b(s)$ is generated on n_b, n_b performs service composition and concatenates S_{BD} with S_{DE}. n_b then locally executes S_{BD}. Since S_{DE} is available on n_a, n_b sends the intermediate result of type D to n_a and invokes S_{DE} on n_a. After completion of S_{DE}, n_a sends the final result of type E to n_b. Thus results are passed on from one service to another across opportunistic connections.

During the pair-wise exchange, nodes also exchange each others' SI table. The SI tables are propagated across the network through a series of contacts. However, it will be necessary

to limit the diffusion of this information both in time and space, based on application, context, resource availability etc. Suppose an application executing on a node n_x wishes to avail of a remote service to execute its task. Now, suppose the remote service (or resource) is available on n_y, n_x determines the opportunistic path from n_x to n_y by using the graphs received from previously contacted nodes. Note that the graph can also be created by using the delay, reputation or another parameter. n_x passes on the parameters for remote task execution through the intermediate nodes. Once the task is executed remotely, n_y returns the outcome of the task execution to the designated node (may be different from n_x).

While performing service composition there might be multiple sequences of services (and the mapping of services to devices). Metrics such as minimum delay or minimum number of services involved or minimum number of devices involved can be used to select the best sequence. The service composer module interacts with the security and privacy module to gain information about trustability of the devices involved in each of the service composition path. Paths that have untrustworthy devices are not considered. The service composition module interacts also with the social interaction module. Services hosted on devices within a social group are considered as preferable. The routing module provides the intercontact time details and the expected delay and route for transmission of data from one device to another. The cache manager provides information such as previously used compositions for a particular service request. If a generated composite service contains services that are available locally on the client, the service composition module interacts with the HCI profiler and the resource predictor modules to know the probability that the specified service will execute successfully.

In the following subsections we present analysis of success probability and expected length of composition. The analysis considers a simplified service composition where factors such as HCI interaction, resource level predictions, social interactions and security concerns are not handled.

Let S_{AB} be a service that accepts input type A and produces output of type B. S_{AB} may be hosted on multiple devices. Let M_{AB} be the number of devices that host S_{AB}. Let N be the number of types of input and outputs and M be the number of devices in the system. Let ρ be the probability that any device hosts a particular service.

6.4.1 Success probability

Probability that a device hosts a service is ρ. The probability of finding a specific service on atleast one device in the network is $1 - (1 - \rho)^M$, where M is the number of devices in the network. The following lemma shows that if service composition is used, the probability of transforming input to output is much higher than performing exact service matching.

Lemma 6.1: If there is a request for service S_{IO}, the probability that there exists a service composition path of length L is given by:

$$P_e(L) = 1 - [1 - \{1 - (1 - \rho)^M\}^L]^{{}^{N-2}P_{L-1}} \tag{6.1}$$

where ${}^{N-2}P_{L-1}$ is the permutation without repetitions and is given by

$${}^{N-2}P_{L-1} = \frac{(N-2)!}{(N-1-L)!}$$

Proof: ρ is the probability that a device hosts a particular service. Hence $(1 - \rho)$ is the probability that a particular service is not available at a particular device. Since there are M devices in the network, $(1 - \rho)^M$ is the probability that a particular service is not available in the entire network. Thus $1 - (1 - \rho)^M$ is the probability that the service is available on at least one device in the network. Given a particular composition of length L, the composition exists,

only if all the services on that path are available on atleast one device in the network. Hence the probability that the service composition path exists is given by $\{1 - (1 - \rho)^M\}^L$.

The probability that the given composition path does not exist, is given by $1 - \{1 - (1 - \rho)^M\}^L$. Since there are N types of data, and the input and output types are already specified by the request, there can be ${}^{N-2}P_{L-1}$ number of paths of length L. Hence the probability that there does not exist any path of length L, for S_{IO}, is given by:

$$[1 - \{1 - (1 - \rho)^M\}^L]^{{}^{N-2}P_{L-1}} \tag{6.2}$$

Thus the probability that there exists atleast one path of length L for the required composition is given by:

$$P_e(L) = 1 - [1 - \{1 - (1 - \rho)^M\}^L]^{{}^{N-2}P_{L-1}}$$

Lemma 6.2: If there is a request for service S_{IO}, the probability that there exists atleast one composition, ie probability of finding a composition, is given by:

$$P_S(S_{IO}) = 1 - \prod_{i=1}^{N-1} \{1 - [1 - (1 - \rho)^M]^i\}^{{}^{N-2}P_{i-1}} \tag{6.3}$$

Proof: The middleware declares that a service composition for S_{IO} is not possible if there does not exist any path between I and O in the service graph. That is, if there is no path of length l for $1 \leq l \leq N - 1$. From (6.2), the probability that a path of length L exists with a probability: $[1-\{1-(1-\rho)^M\}^L]^{{}^{N-2}P_{L-1}}$. Hence the probability of failure, that is the probability that there does not exist any path of any length for S_{IO} is the product of probabilities in (6.2) for all possible values of L. Since there are N data formats, the longest path will have $N - 1$ services chained together. The success probability is ($1-$ probability of failure). Hence the lemma follows.

When a client makes a request for service execution, the middleware on the client's device first checks if such a conversion is possible locally. If not, the service graph is traversed to find a path from specified input to output. If there are multiple possible paths from the input to the required output, the path that has minimum number of services linked together is chosen. This selection of path (path with minimum number of services) is henceforth referred to as Minimum Length Composition (MLC).

6.4.2 Expected length of composition

Lemma 6.3: In MLC, the probability that a requested service S_{IO} gets composed with a composite service of length L is given by:

$$P(L) = (1 - \{1 - [1 - (1-\rho)^M]^L\}^{N-2P_{L-1}}) \prod_{i=1}^{L-1} \{1 - [1 - (1-\rho)^M]^i\}^{N-2P_{i-1}} \qquad (6.4)$$

Proof: In MLC, a path of length L is chosen for service composition only if there does not exist any path of length less than L and there exists at least one path of length L. The probability that no path of length L exists is given by (6.2). The probability that there exists at least one path of length L is given by (6.1). Thus the lemma follows.

The expected length of composite service can now be calculated as:

$$E[L] = P_S(S_{IO}) \sum_{L=1}^{N-1} L.P(L) \qquad (6.5)$$

where $P(L)$ is the probability that a requested service gets composed with a composite service of length L, and is given by (6.4).

6.5 Fault tolerance

Devices in an opportunistic network may be resource constrained, personal and mobile. The message transmission between devices may be prone to packet loss. Consider that a device n_a creates a composite service consisting of services on other devices. Execution of these services on remote devices may suffer from device and communication failures. Hence it is important that the middleware provides fault tolerant service composition. Let $P_s(n_j)$ be the success probability of the jth device in the composite service, that is, the jth device in the composition completes its corresponding service and transfers the result to the next device in the composite service. The entire composition succeeds when all the involved devices succeed in execution and forwarding of the results. Hence success probability of composite service C_i is given by:

$$P_s(C_i) = \prod_{j=1}^{k} P_s(n_j) \tag{6.6}$$

where k is the number of devices in the composition. The service composer sorts all possible compositions in descending order of the preference and selects the best path. Since success probability of the best path might be less than 1, the service composer executes more composition paths in parallel. Let r be the number of independent composition paths executed. The probability $P_s(C)$ that atleast one composition will succeed is given by:

$$P_s(C) = 1 - \prod_{i=1}^{r} 1 - P_s(C_i) \tag{6.7}$$

The middleware selects best r paths such that $P_s(C)$ is sufficiently high and the r paths do not have any device in common. Thus fault tolerance is achieved by replicating service executions.

We now consider the case where the r paths are not distinct, that is, a device is used in more than one composition. Suppose there is a request for composing service $A \rightarrow D$.

Let S^a_{AB} denote that service S_{AB} is hosted on device n_a, S^b_{BC} denote that S_{BC} is hosted on n_b and service S^c_{CD} and S^d_{CD} denote that S_{CD} is hosted on two devices: n_c and n_d. Assume that $S^a_{AB} \rightarrow S^b_{BC} \rightarrow S^c_{CD}$ and $S^a_{AB} \rightarrow S^b_{BC} \rightarrow S^d_{CD}$ are the only two possible compositions for the request $A \rightarrow D$, as shown in Figure 6.4(a).

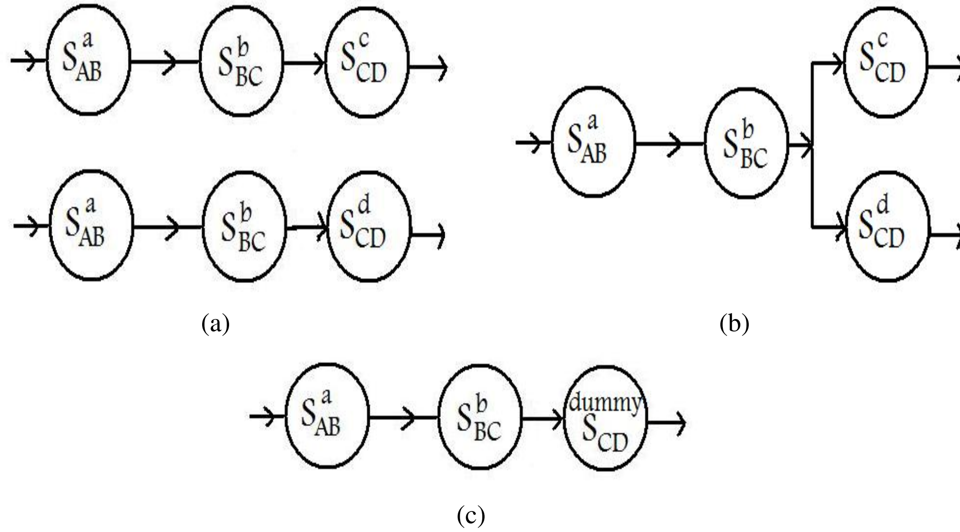


Figure 6.4. Sample combined graph of replicated composite service.
(a) Set of alternatives. (b) Combined replicated service graph. (c) Replacing parallel flows with dummy service.

Figure 6.4(b) shows the combined graph of the two possible execution paths. Since only S_{CD} is replicated on two devices, a failure of n_a or n_b will result in failure of both the paths. The replicated composite service fails if either n_a fails during execution of S_{AB} or n_b fails during execution of S_{BC} (provided n_a has succeeded) or both n_c and n_d fail to complete S_{CD} (provided n_a and n_b have succeeded). The success probability of the replicated composite service is given by:

$$\begin{aligned} P_s(C) \quad = \quad & P_s(S^a_{AB}).P_s(S^b_{BC}).P_s(S^c_{CD}).P_s(S^d_{CD}) + P_s(S^a_{AB}).P_s(S^b_{BC}).P_f(S^c_{CD}).P_s(S^d_{CD}) + \\ & P_s(S^a_{AB}).P_s(S^b_{BC}).P_s(S^c_{CD}).P_f(S^d_{CD}) \end{aligned}$$

where $P_f()$ is the failure probability of the corresponding execution of a service. Since S^c_{CD} and S^c_{CD} are disjoint, we can simplify the above equation as: $P_s(C) = P_s(S^a_{AB}).P_s(S^b_{BC}).\{1 - \left(1 - P_s(S^c_{CD})\right)\left(1 - P_s(S^d_{CD})\right)\}$

Hence to know the probability that atleast one service composition will succeed, first we create a combined graph of the set of compositions. If there are any parallel execution sequences in the graph, the probability of success over those parallel portions is computed using (6.7). As shown in Figure 6.4(c), the parallel portions can then be replaced by a single dummy service whose success probability is equal to that computed for the parallel portion. The graph is thus reduced to a single chain (without any parallelism) of service executions. For this single chain, (6.6) is used.

6.6 Simulation studies and implementation

To simulate mobility of nodes in an opportunistic network, we use Random Waypoint (RWP) mobility model. Modifications proposed in [70] are used to ensure that the simulations are in steady state. The simulation consists of 10 nodes moving in an area of 300m * 300m. The average speed is 10m per minute and average communication range of devices is 10m. Average time required to execute a service on a node is 30 seconds.

We have also implemented the proposed middleware and the service composition algorithms onto Sharp Zaurus PDAs and Toshiba netbooks. The devices communicate using an ad hoc network. We simulate the mobility of these devices as follows. One netbook simulated the mobility of nodes by running the RWP mobility model corresponding to all the network nodes. When two devices in the simulation come within communication range, we establish communication between corresponding two of the actual devices.

We consider a task model similar to that used in [56]. There are 26 types of input/output formats, $A, B, C, ..., Z$. Services convert one format to another. Hence there can be a maximum

of 650 services. Each node hosts a set of services. There might be multiple service providers for a particular service. As explained in Section 5, when two nodes opportunistically come in contact, they exchange information and update their corresponding composite service graphs. When a client node wishes to convert data from one format to another, it first checks if such a conversion is possible locally. If not, the service graph is traversed to find a path from input format to output format. When a node n_a completes execution of a service, n_a transfers the results to the service that is next on the path. Since single hop communication is used, the client might not receive all of the intermediate formats. For simulations in Sections 6.6.1, 6.6.2, 6.6.3 and 6.6.5, we consider that the request consists of only one specified transformation. For simulations in Section 6.6.4, we consider requests with a sequence of specified transformations.

6.6.1 Success probability

We compare MLC approach to a Direct-Match (DM) approach where a single service that provides the required conversion is used. In DM, services are not linked into a composition. Hence if there are few services available in the network, a data format transformation request might not succeed. Since there is always only one service involved in the DM approach, success probability of DM is given by:

$$P_{DM}(S_{IO}) = 1 - (1 - \rho)^M \tag{6.8}$$

Figure 6.5 shows the success of finding a transformation between the requested formats. The analytical trends for MLC and DM are given by (6.3) and (6.8) repectively. The number of devices M in the network is chosen from $\{2, 5, 10, 15\}$. For each subfigure, the number of services provided by each device is varied. Changing the average number of services hosted by each device essentially changes ρ. For analysis curve: value of ρ in (3) = average number of services per device / 650. To find the success probability for, say average 5 services per device

and M = number of devices = 10, the following steps are performed: (1) To ensure that there are average 5 services per device, a *God* program generates 10 numbers such that the average is 5. Each device gets exactly one of these numbers. (2) Each device will host exactly the number, say H_j, of services as provided by the *God* program. (3) There are 26 types of data formats and hence 650 possible services. For each device the following steps are repeated till the device selects H_j service: For each service: a random number between 0 and 1 (inclusive) is generated. If this random value is less than $(H_j/650)$, the corresponding device hosts the service.

As the number of services provided by each node increases, the probability of finding a composite service increases exponentially, whereas that in DM increases linearly. The probability of successfully finding a composite service converges to 1 faster as the number of devices in the system is increased. For DM, the probability of finding the required service also increases with the number of devices in the system. These trends are clearly observed in Figure 6.5.

6.6.2 Length of composition

Figure 6.6 shows the probabilities that the service composition will be of a particular length. The analytical trend is computed using (6.4). Since DM always uses 1 service, the trend for DM is not shown. Each device hosted on an average of 10 services.

In Figure 6.6(a), there were only 2 devices. Hence the probability of success is extremely low. In such an extreme case, the probability of chaining multiple service together is lower than that of finding the exact matching service. Simulations for Figures 6.6(b), 6.6(c) and 6.6(d), there were 5, 10 and 15 devices used respectively. We can see that as the number of devices increases, the probability of finding shorter length composition increases. In all the four subfigures, the summation of probabilities of each length gives the success probability, whose value is same as that in corresponding subfigures of Figure 6.5.

MLC selects compositions that have the minimum number of services and does not consider the intercontact times between nodes. Hence, in an opportunistic network, MLC might

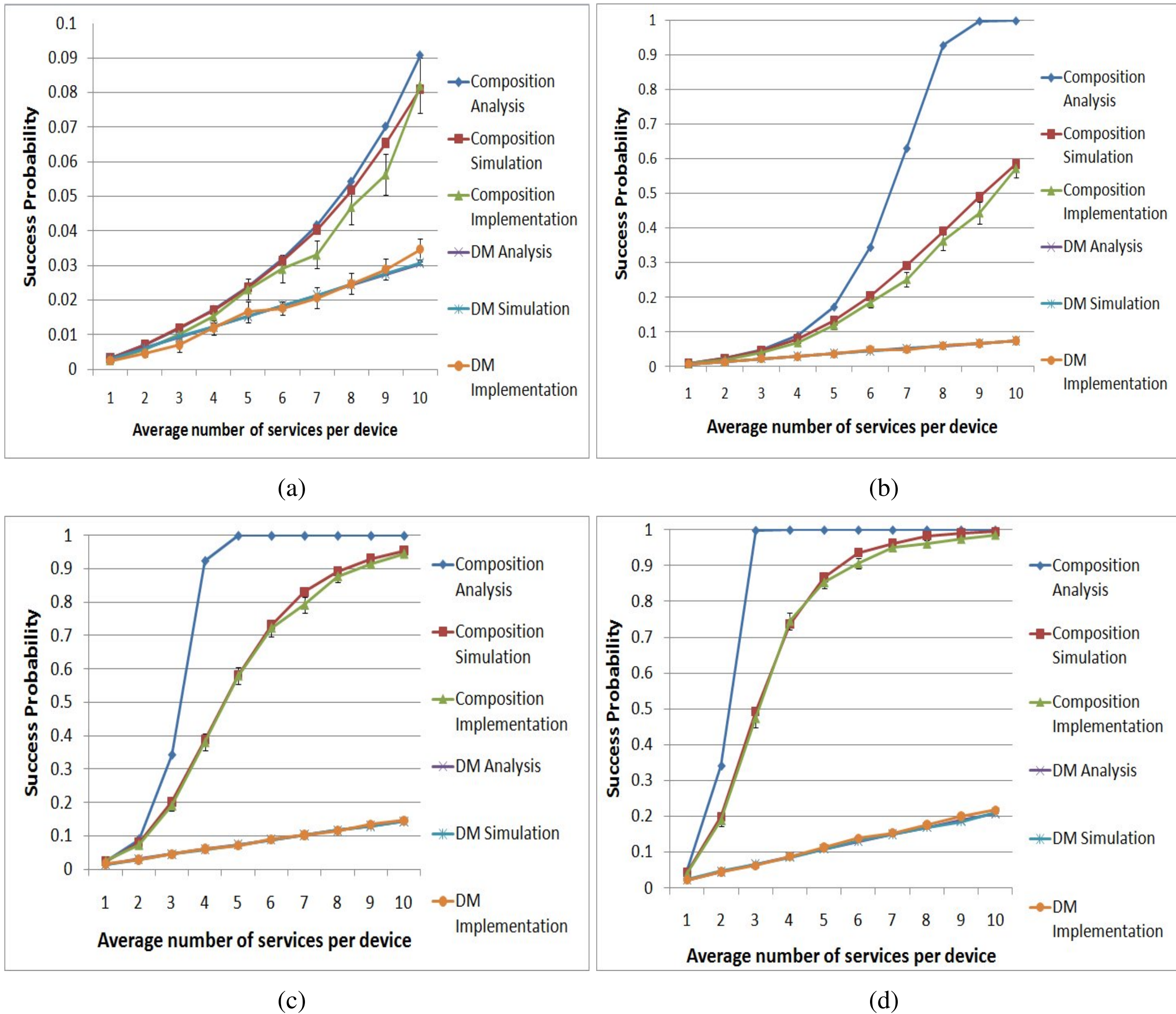


Figure 6.5. Probability of successfully finding a transformation between the requested formats. (a) $M = 2$. (b) $M = 5$. (c) $M = 10$. (d) $M = 15$.

not provide minimum delay in executing the composite service. As the number of services per device increases, initially the success probability is less than 1. During this, the length of composition shows an increasing trend. When the success probability reaches 1, the length of composition starts decreasing with higher number of services per device.

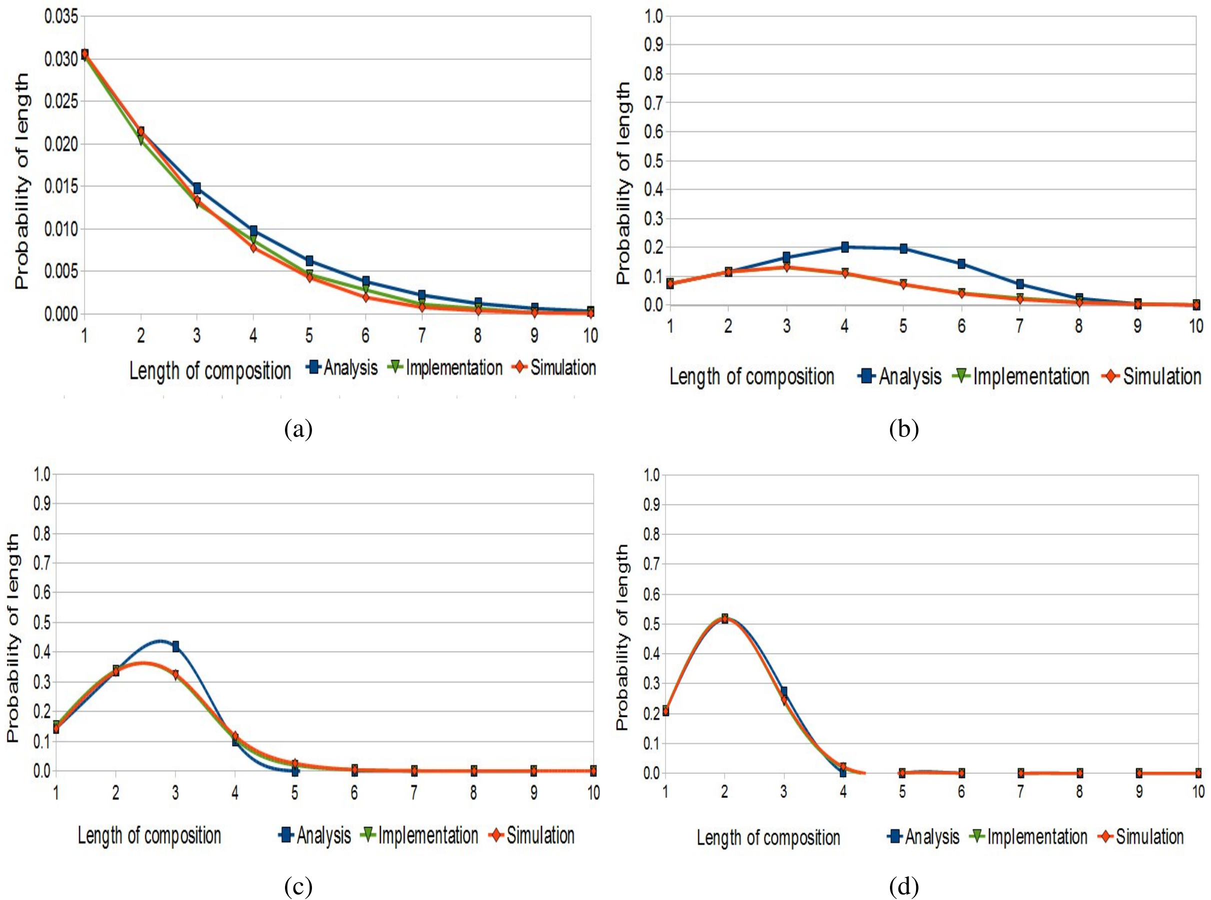


Figure 6.6. Probability of length of composition. (a) $M = 2$. (b) $M = 5$. (c) $M = 10$. (d) $M = 15$.

6.6.3 Time and number of devices required to complete service execution

To make the path selection sensitive to delay, we consider all possible paths for S_{IO}. For each path, we compute the sum of execution times and intercontact times between consecutive services in the compositions. The path with the lowest sum is then chosen. This approach of selecting the service composition that has the lowest sum of expected delays, is henceforth called as Minimized Delay Composition (MDC).

For Figures 6.7 and 6.8, $M = 10$. Each reading is an average of 50 requests, shown with 95% confidence interval. Only those cases where either composite services or direct match

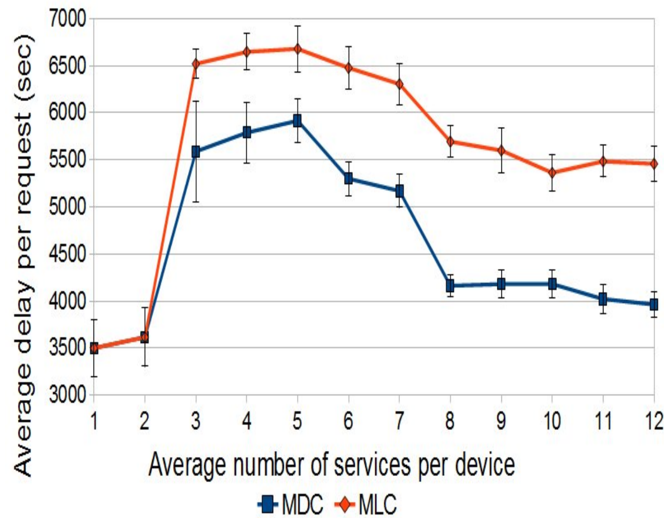


Figure 6.7. Simulation: Average delay in completing the composite service.

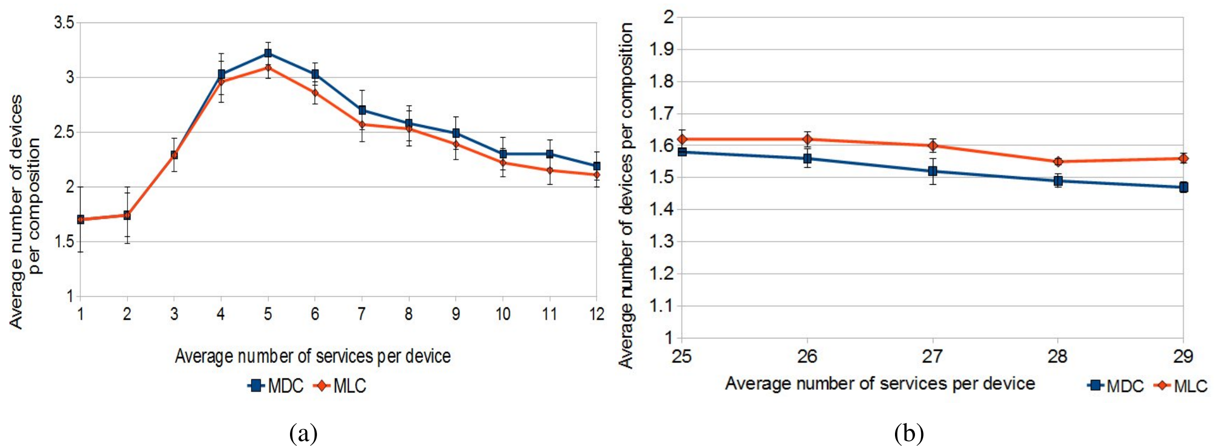


Figure 6.8. Simulation: Average number of devices in composition. (a) 1 to 12 number of services per device. (b) 25 to 29 number of services per device.

were found are considered. Figure 6.7 shows the average delay per request. Since MLC selects paths that have minimum composition length, it has a higher average delay. Figure 6.8 shows the number of devices used in order to satisfy the request. MLC and MDC perform composition whereas DM does not. Thus DM always uses atmost one device to execute the service, whereas MLC and MDC might require multiple devices. Hence the trend for DM is not shown. Figure 6.8 shows a non-intuitive result that even though MDC reduces the average delay per request,

the number of devices used in the composite service is almost same as that in MLC. When the number of services per device increases, MDC tends to reduce the delay due to intercontact time by selecting more services on a device. Though this increases the number of services in the composite service, the number of devices selected by MDC tends to be almost same as than that selected by MLC.

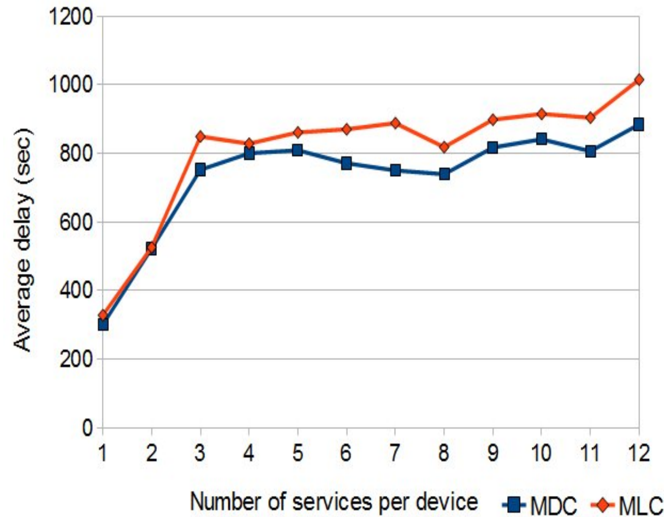


Figure 6.9. Implementation: Average delay while using Haggle dataset.

Figures 6.9 and 6.10 represent data obtained from the implementation on actual devices. Figure 6.9 shows the average delay in executing a composite service. Figure 6.10 shows the average number of devices involved in each composition. The devices use the Haggle [69] real world trace to simulate the mobility. This dataset includes Bluetooth sightings made by 12 students carrying iMotes for six days. Out of the devices of 12 students, we have performed implementation on 10 devices and discarded trace entries of the remaining two students. Since the trace includes timestamps of Bluetooth sightings, establishing connections between devices at the corresponding times ensures that the mobility pattern is followed correctly.

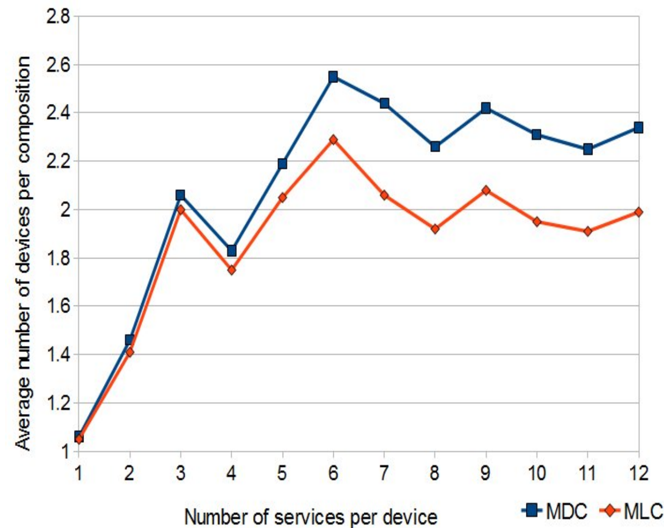


Figure 6.10. Implementation: Average number of devices in composition while using Haggle dataset.

6.6.4 Requests with multiple high level tasks

In the previous simulations we had considered that each request contains only one specified transformation, for example: convert input type I to output type O. In this section, we consider requests that have a sequence of high level tasks. For example: convert input type I to an intermediate results of type (say) X, and convert X to another intermediate result with type (say) Y and finally convert Y to output O. Such requests can be made when a device accepts results in multiple formats but prefers one specific format. For example: request for conversion from HTML to MP3 might also be specified as convert from HTML to TXT to WAV to MP3. Figure 6.11 shows the success probability of composition when the request length is 3. The average number of services per device is varied from 1 to 10. Trends are shown for number of devices varied between 2, 5, 10 and 15. As the number of devices in the network increases, the success probability converges to 1 faster. Figure 6.12 shows the probability of length of composition. There were on average 10 services per device.

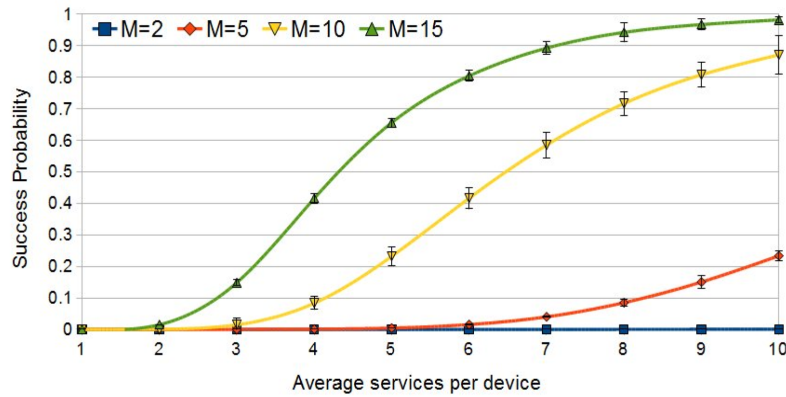


Figure 6.11. Probability of success when request length is 3.

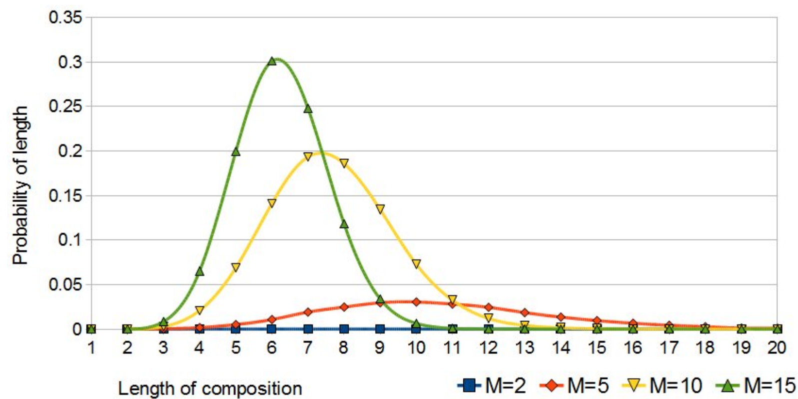


Figure 6.12. Probability of length of composition.

6.6.5 Fault tolerance

Figure 6.13 shows the success probability of service executions with and without using fault tolerance. There were 10 devices in the network and average of 5 services per device. For replication, we need more than one service composition paths between a given input and required output. From Figure 6.5(c), we see that the probability of finding at least one service composition is around 0.6. Hence the probability of finding more than one path will be less than 0.6. Hence in Figure 6.13, when the success probability of individual service is less, the two trends having almost equal values. When the success probability of individual service increases,

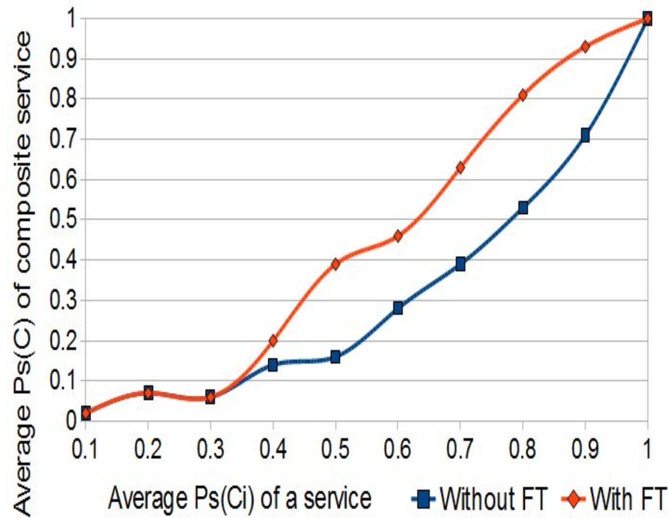


Figure 6.13. Probability of success with and without fault tolerance.

the probability of success with replication is higher. When the success probability of individual services becomes 1, there are no failures, and hence the success probabilities reach 1.

6.7 Prototype using mobile devices in university campus

In addition to emulating mobility as described in previous section, we also have collected Bluetooth sightings between 5 students at the Computer Science and Engineering department of University of Texas at Arlington. The students move around in the campus according to their respective daily schedules. The students move in an area of 1.6 Km x 1.6 Km. The students were chosen randomly. Students and hence their devices came within each other's communication range at such places as labs, classrooms, cafes and dining areas. Each student carried a Bluetooth enabled device (a Toshiba netbook or Fujitsu laptop).

The prototype is implemented using Java and the Bluecove 2.1.0 Bluetooth library [71]. Out of the middleware modules shown in 6.1, the prototype implements the Network Handler, Logger, Service Locator & Composer, Routing Heuristics modules and the PI and SI indices. The task model is same as considered in Section 6.6 with total of 5 distinct data formats. Hence

there were 20 possible distinct services. Each device offers on average 2 services. The prototype software consists of five processes: Request Acceptor, Request Transmitter, Handshake, Service Executor and service requests generator. Each device maintains an *in* queue, which is used to buffer incoming service execution requests and an *out* list, which is used to buffer requests to be transmitted to other devices. The Handshake process periodically performs discovery of other Bluetooth devices. If two devices come within range, the handshake described in Section 6.3 is performed. Each device periodically generates service execution requests. If the service is available locally, the request is added to the local *in* queue. Else the middleware performs service composition. If the device is waiting to initiate service execution on some other device, the request is added to the *out* list. The Request Transmitter process loops over the *out* list and transmits service execution requests to corresponding devices upon opportunistic contacts. The Request Acceptor process receives service execution requests and adds them to the *in* queue. The execution of service is emulated by the Service Executor process. This process reads the earliest received request and sleeps for the time required for service execution. After the sleep, the process adds the remainder of the composite service into the *out* list. Average time required to execute a service on a device is 30 seconds. Current implementation performs Bluetooth device discovery and generation of service execution requests every 100 seconds. These five processes start automatically upon device reboot. The students were allowed to start, stop the processes manually. If a device is low on residual battery, the student may stop the processes. Also, if the student does not expect his device to be within communication range of other participants, the student may stop the prototype to avoid performing unnecessary communication and computations. Allowing students to manually start and stop the processes is realistic due to the fact that while using personal devices, the users might need control over their devices. Hence there were sightings where the processes were not running, but the devices were within communication range.

Table 6.1. Probability of successfully composing a service

Implementation	Analysis	Simulation
0.79	0.81	0.75 ± 0.1

Table 6.1 shows the probability of successfully composing a service using the prototype, analysis and simulation. Figure 6.14 shows the probability of length of composition.

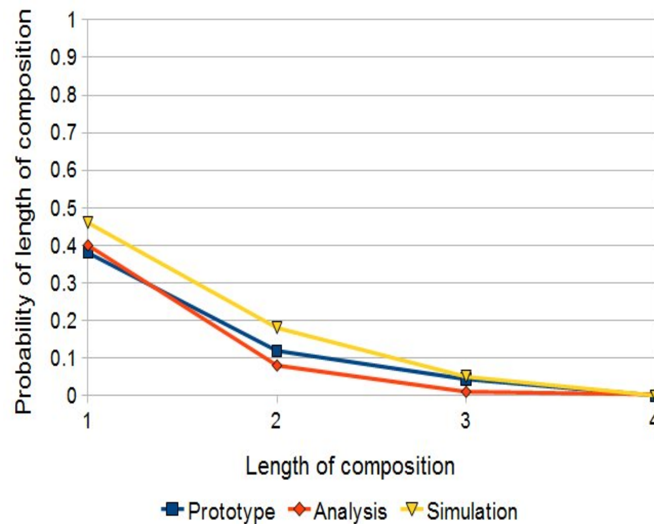


Figure 6.14. Probability of length of composition.

Table 6.2 shows the average intercontact times between each pair of devices. We see that the matrix is not symmetric. This is explained as follows. The Bluetooth hardware on each device is shared amongst the five processes on each corresponding device. It may happen that one device might be in device discovery mode while other device might be transmitting or receiving service execution requests. Hence even though one device n_a, might discover another device n_b, n_b might not discover n_a. Due to mobility, n_b might miss the opportunity to discover n_a. Also the students were allowed to start, stop processes manually. Hence even though n_a discovers n_b, the processes on n_b might not be running.

Table 6.2. Average intercontact time (hours) between devices.

	Device 1	Device 2	Device 3	Device 4	Device 5
Device 1	-	22.21 ± 2	28.96 ± 2.2	17.1 ± 1.2	28.28 ± 1.6
Device 2	55 ± 3.5	-	∞	∞	∞
Device 3	58 ± 4.7	∞	-	∞	∞
Device 4	29 ± 1.6	∞	∞	-	5.91 ± 0.9
Device 5	36 ± 2.8	∞	∞	5.1 ± 1.1	-

Table 6.3 shows the average delay per composition, average number of devices involved per composition and the probability of successfully completing a composite service. If devices n_x and n_a come within communication range, and n_x needs to invoke a service on device n_a, the invocation might fail due to mobility during the communication. Hence n_x attempts that particular invocation during atmost three contacts with n_a. If invocation fails during all the contacts, a failure is declared. Consider that n_x successfully invokes a service on n_a. The student that is carrying the device n_a might turn of the device at any time. Also the device might deplete its battery. We consider these three factors to contribute towards failure.

Though the devices were given to students from same department, the mobility patterns were remarkably different. Student 1 came in contact with all other students. Students 2 and 3 came in contact with only student 1. Students 4 and 5 came in contact with each other and student 1. Service advertisements were transmitted using multi hop communication. Requests for service invocations were limited to single hop. Hence all requests for invoking services on devices that never came within communication range were left unsatisfied.

6.8 Summary and discussion

In this chapter, we have proposed a middleware architecture for opportunistic computing. The middleware is modular and can be deployed onto any device. An extremely resource constrained device might host few of the modules, whereas devices rich in resources can implement

Table 6.3. Average delay and number of devices involved per composition

	Average delay per composition	Average number of devices per composition	Probability of successfully executing a composite service
Device 1	15.03 ± 3.9	1.98 ± 0.045	0.16
Device 2	0.3 ± + 0.08	1.43 ± 0.021	0.05
Device 3	28.91 ± 4.3	1.6 ± 0.015	0.11
Device 4	0.59 ± 0.1	2.1 ± 0.025	0.124
Device 5	8.2 ± 1.7	2.23 ± 0.026	0.06

all of the modules. The chapter also presents service composition in opportunistic networks and provides analysis of success probability and length of composed service. To the best knowledge of the authors this dissertation is the first to perform service composition in opportunistic networks. Since the devices and service executions are prone to failure, the middleware performs fault tolerant service composition by the means of replication. The middleware creates multiple service compositions such that the probability of success of atleast one composite service is high. The service composition algorithm has been implemented in a prototype where students in a university campus carry Bluetooth enabled devices. We have verified the analytical results with the results from simulation and prototype.

The current prototype uses single hop message passing for service execution requests. We are working on implementing existing routing algorithms into the prototype. Traces of 5 Bluetooth enabled devices have been recorded using the prototype. We are enrolling more volunteers to participate in the opportunistic network. The service composition analysis is presented for services that are chained together in a sequential fashion. We envisage to add algorithms and analysis to handle parallelism in service composition where, output of one service might be given to multiple services. In an opportunistic network, information propagation incurs huge time delay. Hence it may happen that the composite service graph created on different devices may be different. For a service request, there may neither be the exact matching service nor

any composite service available in the composite service graph. Instead of declaring that service composition is not possible, the devices may predict a possible path, execute that partial path and then transfer the service request to other devices in the network. We are working on algorithms to predict such paths and perform service composition in presence of limited service graphs.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Summary of Contributions

This dissertation provides algorithms for cooperative service executions in pervasive environments that do not have high end or centralized servers. The CBS algorithm [19] is designed for service execution in smart environments. The devices are assumed to be grouped into logical clusters. Each cluster has a cluster head. A *client* is a device that wishes to schedule a set of tasks. Scheduling is performed in iterations. The client communicates only with the cluster heads. The cluster heads perform scheduling of the DAG onto devices in their respective clusters. The main advantages of CBS as compared to other schemes are: (*i*) Communication overhead is reduced by considering the devices to be grouped into logical clusters. (*ii*) Schedules are generated in a distributed fashion. Hence CBS does not require the task scheduler device to have knowledge of all characteristics of each device in the environment. (*iii*) CBS allows usage of multiple task scheduling algorithms. Simulation results demonstrate time and energy efficient scheduling of tasks in heterogeneous environments.

The Distributed Grading Autonomous Selection (DGAS) [20] assigns independent tasks onto mobile service providers. Each device autonomously decides whether it is resourceful enough to participate in service execution. Factors such as residual energy, mobility, time required for communication are used to decide whether a device is suitable to participate in service execution. DGAS is suitable for task offloading in MANETs and VANETs. DGAS provides fault tolerance by replicating service execution onto multiple devices. The main advantages of DGAS over existing schemes are: (*i*) Communication overhead is reduced since each device maintains its own history and need not share its private information with other devices in the

network. (*ii*) Each device autonomously decides whether to participate in service execution. DGAS allows each device to use a different algorithm for computing its success probability in executing a service. (*iii*) Simulation results show savings in time due to lower communication overhead. From the simulations we can also clearly see that inspite of devices making autonomous decisions, the number of replicas are effectively controlled.

The MEOP algorithm [21] enables exclusive access to shared resources in an opportunistic network. MEOP is independent of the underlying routing protocols and hence does not need continuous monitoring of the entire network. Each device maintains a request pending queue. When a device wishes to access the shared resource, a request is generated and added to the pending queue. When two devices opportunistically come in contact, the pending queues are exchanged and updated. MEOP is a token based algorithm. A device can access the critical section only when the device possesses the token. Token requests are propogated over a logical DAG. MEOP uses a timeout based fault detection algorithm. MEOP is free from starvation, deadlock and allows only one application to access the shared resource at a time. To the best of knowledge, MEOP is the first to enable mutual exclusion in opportunistic networks. Unlike existing mutual exclusion algorithms for MANETs [22], the proposed algorithm does not require continuous monitoring of the network topology. Simulation results show that MEOP is communication efficient as compared to other algorithms proposed for generic MANETs. MEOP uses a novel timeout based fault detection algorithm that exploits the inter contact time distributions.

This dissertation is the first to study service composition in opportunistic networks. Two algorithms: Minimum Length Composition (MLC) and Minimum Delay Composition (MDC) are proposed [25]. Analysis for success probability of service composition and length of compositions is presented and verified. A middleware architecture for opportunistic computing is also proposed. The middleware is modular and can be deployed onto any device. An extremely resource constrained device might host few of the modules, whereas devices rich in resources can implement all of the modules. Service execution over devices in OPNETs might suffer

from device and communication failures. Hence the middleware performs fault tolerant service composition by generating multiple composite services. The service composition algorithms are implemented on devices such as PDAs and netbooks. A prototype of the service composition process has been implemented. In the prototype, users carry Bluetooth enabled devices in campus of University of Texas at Arlington. Service execution requests are exchanged when the devices opportunistically come within communication range.

Algorithms proposed in this dissertation enable distributed, autonomous execution of services in pervasive environments. The algorithms exploit parallelism among pervasive devices. Mutual exclusion is a fundamental problem in distributed systems. MEOP is the first algorithm to enable mutually exclusive access to resources in OPNETs. Service composition algorithms and proposed middleware increase the number of services that can be achieved by devices in OPNETs. The algorithms proposed in this dissertation form a core set that is required to develop user-level applications with increased communication and computational complexity. The proposed algorithms and the simulation results encourage more research in the direction of distributed service deployment, execution and maintenance. Software companies such as [77] perform mobile cloudsourcing and can benefit from the algorithms proposed in this dissertation.

7.2 Future research directions

The research work presented in this dissertation can be extended in many directions. Following is a list of extensions to the current work:

Currently DGAS schedules set of independent tasks onto mobile devices. The algorithm should be extended to schedule dependent tasks given in the form of a Directed Acyclic Graph (DAG). In DGAS and DGCS, if the client does not receive any reply, the client declares the attempt to be a failure and retries by sending the same request again to the nearest agent. There are three cases for failure: (*i*) There were no matching service providers that received the request,

(*ii*) The corresponding service was executed on atleast one device, but the service execution failed due to resource constraints, and (*iii*) Service was executed successfully, but the results were lost during communication failures. If the failure was due to the communication loss and the client retries, the service executions done by surrogates are wasted. Instead, the client should send a *resend_result* message to the surrogates. The failure might have occured due to any of the three above mentioned causes. Hence a history should be maintained to find probabilities of each cause. Depending on these probabilities, the client should decide whether to retry or send the *resend_result* message.

The service composition algorithm described in chapter 6 handles service requests that are in chained together in a sequential fashion. We envisage to add algorithms and analysis to handle parallelism in service composition request, where the output of one service might be given to multiple services.

Self stabilization in opportunistic networks: Consider that there are N types of services in the network. Each service might be hosted by multiple devices. Each device might host a limited number of services and can have (store/have installed) more number of services than it can offer. There is a need to develop novel algorithms to decide which device should host which service, such that: (a) there is at least one service hosted of each type. (b) each service is hosted same number of times. For example, if there are 10 devices, 10 services and each device can host at most 1 service, problem reduces to assignment of ID to devices.

In an opportunistic network, information propagation incurs huge time delay. Hence it may happen that the composite service graph created on different devices may be different. For a service request, there may neither be the exact matching service nor any composite service available in the composite service graph. Instead of declaring that service composition is not possible, the devices may predict a possible path, execute that partial path and then transfer the service request to other devices in the network. Hence algorithms are needed to predict such paths and perform service composition in presence of limited service graphs.

APPENDIX A

PUBLICATIONS FROM THIS DISSERTATION

1. S. A. Tamhane and M. Kumar, Middleware for Decentralised Fault Tolerant Service Execution using Replication in Pervasive Systems, Middleware Support for Pervasive Computing, Workshop at IEEE PerCom, pp: 474-479 Germany, April 2010.
2. S. A. Tamhane and M. Kumar, Token Based Algorithm for Supporting Mutual Exclusion in Opportunistic Networks, International Workshop on Mobile Opportunistic Networking (MobiOpp 2010), Italy, 22-23 Feb., 2010.
3. S. A. Tamhane and M. Kumar, Task scheduling on Heterogeneous Devices in Parallel Pervasive Systems (P^2S), 15th Annual IEEE International Conference on High Performance Computing (HiPC 2008), P. Sadayappan et al. (Eds.), LNCS 5374, pp: 427-438, Bangalore, India, December 17-20, 2008.
4. S. A. Tamhane, M. Kumar, A. Passarella and M. Conti, Service Composition in Opportunistic Networks, in progress.
5. S. A. Tamhane and M. Kumar, Mutual Exclusion in Opportunistic Networks, Journal, in progress.

REFERENCES

[1] A. Bauch, E. Maehle, and F.-J. Markus, “A distributed algorithm for fault-tolerant dynamic task scheduling,” in *Second Euromicro Workshop on Parallel and Distributed Processing*, 1994, pp. 309–316.

[2] M. Weiser, “The computer for 21st century,” *Journal of Scientific American*, vol. 265, no. 3, pp. 94–104, 1991.

[3] W. S. Ark and T. Selker, “A look at human interaction with pervasive computers,” *IBM Systems Journal*, vol. 38, no. 4, pp. 504–508, 1999.

[4] A. Passarella, M. Kumar, M. Conti, and E. Borgia, “Minimum-delay service provisioning in opportunistic networks,” IIT-CNR, Tech. report 11, 2009.

[5] P. Hui, J. Crowcroft, and E. Yoneki, “Bubble rap: social-based forwarding in delay tolerant networks,” in *International Symposium on Mobile Ad Hoc Networking and Computing*, 2008, pp. 241–250.

[6] D. Svensson, G. Hedin, and B. Magnusson, “Pervasive applications through scripted assemblies of services,” in *International Conference on Pervasive Services*, 2007, pp. 301–307.

[7] M. Aiello and S. Dustdar, “Are our homes ready for services? a domotic infrastructure based on the web service stack,” *Pervasive and Mobile Computing*, vol. 4, no. 4, p. 506525, 2008.

[8] A. Chowdhury, B. Falchuk, and A. Misra, “Medially: A provenance-aware remote health monitoring middleware,” in *International conference on Pervasive Computing and Communications (PerCom)*, 2010, pp. 125–134.

[9] J. Liang and K. Nahrstedt, "Service composition for advanced multimedia applications," in *Multimedia computing and networking*, 2005, pp. 228–240.

[10] Y. Yang, F. Mahon, M. Williams, and T. Pfeifer, "Context-aware dynamic personalised service re-composition in a pervasive service environment," in *Ubiquitous Intelligence and Computing, LNCS 4159*, 2006, p. 724735.

[11] J. Sonnek, A. Chandra, and J. B. Weissman, "Adaptive reputation-based scheduling on unreliable distributed infrastructure," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 11, pp. 1551–1564, November 2007.

[12] A. Litke, D. Skoutas, K. Tserpes, and T. Varvarigou, "Efficient task replication and management for adaptive fault tolerance in mobile grid environments," *Future Generation Computer Systems*, vol. 23, no. 2, pp. 163–178, February 2007.

[13] H. Topcuoglu, S. Hariri, and M. Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[14] M. K. Dhodhi, I. Ahmad, A. Y. Muhammad, and I. Ahmad, "An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems," *J. Parallel and Distributed Computing*, vol. 62, no. 9, pp. 1338–1361, 2002.

[15] Y. Yu and V. K. Prasanna, "Energy-balanced task allocation for collaborative processing in wireless sensor networks," *ACM/Kluwer Journal of Mobile Networks and Applications (MONET) Special Issue on Algorithmic Solutions for Wireless, Mobile, Ad Hoc and Sensor Networks*, vol. 10, no. 1, February 2005.

[16] N. Eagle and A. Pentland, "Reality mining: Sensing complex social systems," *Personal and Ubiquitous Computing*, vol. 10, no. 4, 2006.

[17] B. Lagesse, M. Kumar, and M. Wright, "Resco: A middleware component for reliable service composition in pervasive systems," in *Middleware Support for Pervasive Computing Workshop, PerCom*, April 2010.

[18] D. Henrici and P. Muller, "Providing security and privacy in rfid systems using triggered hash chains," in *Pervasive Computing and Communications (PerCom)*, March 2008, pp. 50–59.

[19] S. A. Tamhane and M. Kumar, "Task scheduling on heterogeneous devices in parallel pervasive systems (p^2s)," in *15th Annual IEEE International Conference on High Performance Computing (HiPC 2008)*, P. S. et al., Ed. LNCS 5374, December 2008, pp. 427–438.

[20] ——, "Middleware for decentralised fault tolerant service execution using replication in pervasive systems," in *Middleware Support for Pervasive Computing, Workshop at IEEE PerCom*, April 2010, pp. 474–479.

[21] ——, "Token based algorithm for supporting mutual exclusion in opportunistic networks," in *International Workshop on Mobile Opportunistic Networking (MobiOpp)*, Feb 2010.

[22] A. Derhab and N. Badache, "A distributed mutual exclusion algorithm over multi-routing protocol for mobile ad hoc networks," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 23, no. 3, pp. 197–218, June 2008.

[23] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "An integrated workbench for model-based engineering of service compositions," *IEEE Transactions on Services Computing*, vol. 3, no. 2, pp. 131–144, May 2010.

[24] S. Kalasapur, B. A. Shirazi, and M. Kumar, "Dynamic service composition in pervasive computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 7, pp. 907–918, 2007.

[25] S. A. Tamhane, M. Kumar, A. Passarella, and M. Conti, "Service composition in opportunistic networks," 2010, in progress.

[26] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang, "The case for cyber foraging," in *ACM SIGOPS European Workshop*, 2002, pp. 87–92.

[27] Y.-Y. Su and J. Flinn, “Slingshot: deploying stateful services in wireless hotspots,” in *3rd international conference on Mobile systems, applications, and services*, 2005, pp. 79 – 92.

[28] N. Palmer, R. Kemp, T. Kielmann, and H. Bal, “Ibis for mobility: Solving challenges of mobile computing using grid techniques,” in *Workshop on Mobile Computing Systems and Applications*, 2009, p. 17.

[29] M. Kristensen, “Execution plans for cyber foraging,” in *Workshop on Mobile middleware: embracing the personal communication device*, 2008, p. 2.

[30] O. Gnawali, K.-Y. Jang, J. Paek, M. Vieira, R. Govindan, B. Greenstein, A. Joki, D. Estrin, and E. Kohler, “The tenet architecture for tiered sensor networks,” in *Embedded Networked Sensor Systems*, 2006, pp. 153–166.

[31] M. Ma, C. Wang, and F. Lau, “Jessica: Java-enabled single-system-image computing architecture,” *Journal of Parallel and Distributed Computing*, vol. 60, no. 10, pp. 1194–1222, 2000.

[32] V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan, “Distributed job scheduling on computational grids using multiple simultaneous requests,” in *11th IEEE International Symposium on High Performance Distributed Computing*, 2002, p. 359.

[33] W. Alsalih, S. Akl, and H. Hassanein, “Energy-aware task scheduling: towards enabling mobile computing over manets,” in *International Parallel and Distributed Processing Symposium (IPDPS)*, April 2005, p. 8.

[34] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. G. Sirer, “Design and implementation of a single system image operating system for ad hoc networks,” in *International Conference on Mobile Systems, Applications, and Services*, 2005, pp. 149–162.

[35] K. E. Maghraoui, T. Desell, B. K. Szymanski, J. D. Teresco, and C. Varela, “Towards a middleware framework for dynamically reconfigurable scientific computing,” *Grid Computing and New Frontiers of High Performance Processing*, vol. 14, pp. 275–301, 2005.

[36] R. Shepherd, J. Story, and S. Mansoor, “Parallel computation in mobile systems using bluetooth scatternets and java,” in *International Conference on Parallel and Distributed Computing and Networks*, 2004.

[37] L. McNamara, C. Mascolo, and L. Capra, “Media sharing based on colocation prediction in urban transport,” in *International Conference on Mobile Computing and Networking (MobiCom)*, September 2008, pp. 58–69.

[38] Y. Zhang, J. Zhao, and G. Cao, “Roadcast: a popularity aware content sharing scheme in vanets,” *Mobile Computing and Communications Review*, vol. 13, no. 4, pp. 1–14, 2009.

[39] K. Lee, S. Lee, R. Cheung, U. Lee, and M. Gerla, “First experience with cartorrent in a real vehicular ad hoc network testbed,” in *Mobile Networking for Vehicular Environments*, May 2007, pp. 109 – 114.

[40] F. Delicato, L. Fuentes, N. Gmez, and P. Pires, “A middleware family for vanets,” in *Ad-Hoc, Mobile and Wireless Networks*, August 2009, pp. 379–384.

[41] P. Pantazopoulos, I. Stavrakakis, A. Passarella, and M. Conti, “Efficient social-aware content placement in opportunistic networks,” in *International Conference on Wireless On-demand Network Systems and Services*, February 2010.

[42] J. Ott and D. Kutscher, “Bundling the web: Http over dtn,” in *Workshop on Networking in Public Transport (WNEPT)*, Aug. 2006.

[43] C. Boldrini, M. Conti, I. Iacopini, and A. Passarella, “Hibop: a history based routing protocol for opportunistic networks,” in *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM 2007)*, June 2007.

[44] Y. Wang, S. Jain, M. Martonosi, and F. K., “Erasure-coding based routing for opportunistic networks,” in *ACM SIGCOMM workshop on Delay-tolerant networking*, 2005, pp. 229–236.

[45] P. Costa, C. Mascolo, M. Musolesi, and G. Picco, “Socially-aware routing for publish-subscribe in delaytolerant mobile ad hoc networks,” *Journal on Selected Areas in Communications*, vol. 26, no. 5, pp. 748–760, June 2008.

[46] T. Spyropoulos, K. Psounis, and C. Raghavendra, “Efficient routing in intermittently connected mobile networks: the multiple-copy case,” *Transactions on Networking (TON)*, vol. 16, no. 1, pp. 77–90, February 2008.

[47] M. Abdulla and R. Simon, “A simulation study of common mobility models for opportunistic networks,” in *41st Annual Simulation Symposium (anss-41 2008)*, 2008, pp. 43–50.

[48] S. Nesargi and R. Prakash, “Manetconf: Configuration of hosts in a mobile ad hoc network,” *INFOCOM 2002*, vol. 2, pp. 1059– 1068, 2002.

[49] N. Malpani, Y. Chen, N. H. Vaidya, and J. L. Welch, “Distributed token circulation in mobile ad hoc networks,” *IEEE Transactions on Mobile Computing (TMC)*, vol. 4, no. 2, pp. 154–165, March/April 2005.

[50] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems Concepts and Design*, 2001, ch. 11.

[51] R. Baldoni, A. Virgillito, and R. Petrassi, “A distributed mutual exclusion algorithm for mobile ad-hoc networks,” *International Symposium on Computers and Communications*, pp. 539– 544, 2002.

[52] W. Wu, J. Cao, and M. Raynal, “A dual-token-based fault tolerant mutual exclusion algorithm for manets,” *Mobile Ad-Hoc and Sensor Networks*, vol. 4864/2007, pp. 572–583, 2007.

[53] M. Maekawa, “A $\sqrt{N}$ algorithm for mutual exclusion decentralized systems,” *ACM Transactions on Computer Systems*, vol. 3, no. 2, pp. 145–159, 1985.

[54] G. Ricart and A. Agrawala, “An optimal algorithm for mutual exclusion in computer network,” *Communication of the ACM*, vol. 24, no. 1, pp. 9–17, Jan. 1981.

[55] W. Wu, J. Cao, and J. Yang, "A fault tolerant mutual exclusion algorithm for mobile ad hoc networks," *Pervasive and Mobile Computing (PMC)*, vol. 4, no. 1, pp. 139–160, February 2008.

[56] S. Kalasapur, M. Kumar, and B. A. Shirazi, "Seamless service composition (sesco) in pervasive environments," in *Workshop on Multimedia Service Composition (MSC2005), ACM Multimedia Conference*, Nov 2005, pp. 11–20.

[57] D. Chakraborty, A. Joshi, T. Finin, and Y. Yesha, "Service composition for mobile environments," *Mobile Networks and Applications*, vol. 10, no. 4, pp. 435–451, Aug. 2005.

[58] N. Ibrahim and F. L. Mouel, "A survey on service composition middleware in pervasive environments," *International Journal of Computer Science Issues*, vol. 1, pp. 1–12, Aug. 2009.

[59] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, pp. 406–471, 1999.

[60] T. Phan, L. Huang, and C. Dulan, "Challenge: Integrating mobile wireless devices into the computational grid," in *8th ACM International Conference on Mobile Computing and Networking (Mobicom)*, 2002, pp. 271–278.

[61] T. N'Takpe and F. Suter, "Critical path and area based scheduling of parallel task graphs on heterogeneous platforms," in *12th International Conference on Parallel and Distributed Systems (ICPADS)*, 2006, pp. 3–10.

[62] S. C. Kim, S. Lee, and J. Hahm, "Push-pull: Deterministic search-based dag scheduling for heterogeneous cluster systems," *Transactions on Parallel and Distributed Systems*, vol. 18, no. 11, pp. 1489–1502, November 2007.

[63] M. Chatterjee, S. K. Das, and D. Turgut, "Wca: A weighted clustering algorithm for mobile ad hoc networks," *Journal of Cluster Computing.*, vol. 5, no. 2, pp. 193–204, 2002.

[64] Z.-T. Lin, Y.-G. Qu, X.-F. Zhou, and B. Zhao, "Analysis and design of mobile wireless social model," *Computer Communications*, vol. 32, no. 5, pp. 927–934, March 2009.

[65] L. Lilien, Z. Kamal, V. Bhuse, and A. Gupta, "Opportunistic networks: The concept and research challenges," in *International Workshop on Research Challenges in Security and Privacy for Mobile and Wireless Networks (WSPWN)*, March 2006.

[66] V. Davies, "Evaluating mobility models within an ad hoc network," Ph.D. dissertation, Colorado School of Mines, 2000, master's thesis.

[67] D. B. Johnson and D. A. Maltz, "Dynamic source routing in ad hoc wireless networks," in *Mobile Computing*. Kluwer Academic Publishers, 1996, pp. 153–181.

[68] Q. M. Chaudhari and E. Serpedin, "Clock offset and skew estimation in wireless sensor networks with known deterministic delays and exponential nondeterministic delays," in *International conference on Digital Telecommunications (ICDT '08)*, June 2008, pp. 37–40.

[69] J. Scott, R. Gass, J. Crowcroft, P. Hui, C. Diot, and A. Chaintreau, "CRAWDAD trace cambridge/haggle/imote/cambridge (v. 2006-01-31)," Downloaded from http://crawdad.cs.dartmouth.edu/ cambridge/haggle/imote/cambridge, Jan. 2006.

[70] W. Navidi, T. Camp, and N. Bauer, "Improving the accuracy of random waypoint simulations through steady-state initialization," in *15th International Conference on Modeling and Simulation*, March 2004, pp. 319–326.

[71] "Bluecove: Java library for bluetooth (jsr-82 implementation)," URL: http://bluecove.org/.

[72] A. Lindgren, A. Doria, and O. Scheln, "Probabilistic routing in intermittently connected networks," *MobiHoc 2003 posters*, vol. 7, no. 3, pp. 19–20, July 2003.

[73] M. Conti and M. Kumar, "Opportunities in opportunistic computing," *Computer*, vol. 43, no. 1, pp. 42–50, Jan. 2010.

[74] L. Pelusi, A. Passarella, and M. Conti, "Opportunistic networking: data forwarding in disconnected mobile ad hoc networks," *IEEE Communications Magazine*, vol. 44, no. 11, pp. 134–141, Nov. 2006.

[75] N. Ravi, J. Scott, L. Han, and L. Iftode, "Context-aware battery management for mobile phones," in *Pervasive Computing and Communications (PerCom)*, March 2008, pp. 224–233.

[76] Y. Wen, R. Wolski, and C. Krintz, "History-based, online, battery lifetime prediction for embedded and mobile devices," in *Workshop on Power - Aware Computer Systems*, Dec. 2004, pp. 57–72.

[77] N. Eagle and B. Olding, "txteagle," www.txteagle.com.

BIOGRAPHICAL STATEMENT

Sagar A. Tamhane was born in Thane, Maharashtra, India, in 1979. He received his B.E degree in Computer Engineering fom Mumbai University, India, in 2000. He received his M.S. degree in Computer Science and Engineering from the Indian Institute of Technology, Bombay, India in 2005 and his Ph.D. degree in Computer Science from The University of Texas at Arlington in 2010. From 2003 to 2005, he worked as a software engineer for Sun Microsystems, Bangalore, India. His current research interests are in the areas of Pervasive and Mobile computing, Distributed and Parallel computing. He is a student member of IEEE. In 2009, he was inducted into the Upsilon Pi Epsilon (Texas Gamma Chapter) honor society for the computing and information disciplines.