Report Number: WUCS-2008-16

2008-08-01

Supporting Collaboration in Mobile Environments

Rohan Sen

Continued rapid improvements in the hardware capabilities of mobile computing devices is driving a parallel need for a paradigm shift in software design for such devices with the aim of ushering in new classes of software applications for devices of the future. One such class of software application is collaborative applications that seem to reduce the burden and overhead of collaborations on human users by providing automated computational support for the more mundane and mechanical aspects of a cooperative effort. This dissertation addresses the research and software engineering questions associated with building a workflow-based collaboration system that can operate... **Read complete abstract on page 2.**

Supporting Collaboration in Mobile Environments

Rohan Sen

Complete Abstract:

Continued rapid improvements in the hardware capabilities of mobile computing devices is driving a parallel need for a paradigm shift in software design for such devices with the aim of ushering in new classes of software applications for devices of the future. One such class of software application is collaborative applications that seem to reduce the burden and overhead of collaborations on human users by providing automated computational support for the more mundane and mechanical aspects of a cooperative effort. This dissertation addresses the research and software engineering questions associated with building a workflow-based collaboration system that can operate across mobile ad hoc networks, the most dynamic type of mobile networks that can function without dependence on any fixed external resources. While workflow management systems have been implemented for stable wired networks, the transition to a mobile network required the development of a knowledge management system for improving the predictability of the network topology, a mobility-aware specification language to specify workflows, and its accompanying algorithms that help automate key pieces of the software. In addition to details of the formulation, design, and implementation of the various algorithms and software components. this dissertation also describes the construction of a custom mobile workflow simulator that can be used to conduct simulation experiments that verify the effectiveness of the approaches presented in this document and beyond. Also presented are empirical results obtained using this simulator that show the effectiveness of the described approaches.

2008-16

Supporting Collaboration in Mobile Environments

Authors: Rohan Sen

Corresponding Author: rohan.a.sen@gmail.com

Web Page: http://www.cs.wustl.edu/~ras6

Abstract: Continued rapid improvements in the hardware capabilities of mobile computing devices is driving a parallel need for a paradigm shift in software design for such devices with the aim of ushering in new classes of software applications for devices of the future. One such class of software application is collaborative applications that seek to reduce the burden and overhead of collaborations on human users by providing automated computational support for the more mundane and mechanical aspects of a cooperative eff ort. This dissertation addresses the research and software engineering questions associated with building a workflow-based collaboration system that can operate across mobile ad hoc networks, the most dynamic type of mobile networks that can function without dependence on any fi xed external resources. While workflow management systems have been implemented for stable wired networks, the transition to a mobile network required the development of a knowledge management system for improving the predictability of the network topology, a mobility-aware specifi cation language to specify workflows that execute across mobile networks in the physical world, a mobile workflow management system to execute the workflows, and its accompanying algorithms that help automate key pieces of the software. In addition to details of the formulation, design, and implementation of the various algorithms and software components, this dissertation also describes the

Type of Report: PhD Dissertation

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science

Department of Computer Science and Engineering

Thesis Examination Committee:
Gruia-Catalin Roman, Chair
Christopher Gill
Yixin Chen
Carl Gunter
Peter Hovmand
William D. Smart

SUPPORTING COLLABORATION IN MOBILE ENVIRONMENTS

by

Rohan Sen

A dissertation presented to the School of Engineering
of Washington University in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

August 2008
Saint Louis, Missouri

ABSTRACT OF THE THESIS

Supporting Collaboration in Mobile Environments

by

Rohan Sen

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2008

Research Advisor: Professor Gruia-Catalin Roman

Continued rapid improvements in the hardware capabilities of mobile computing devices is driving a parallel need for a paradigm shift in software design for such devices with the aim of ushering in new classes of software applications for devices of the future. One such class of software application is collaborative applications that seek to reduce the burden and overhead of collaborations on human users by providing automated computational support for the more mundane and mechanical aspects of a cooperative effort.

This dissertation addresses the research and software engineering questions associated with building a *workflow-based* collaboration system that can operate across mobile ad hoc networks, the most dynamic type of mobile networks that can function without dependence on any fixed external resources. While workflow management systems have been implemented for stable wired networks, the transition to a mobile network required the development of a knowledge management system for improving the predictability of the network topology, a mobility-aware specification language to

specify workflows that execute across mobile networks in the physical world, a mobile workflow-management system to execute the workflows, and its accompanying algorithms that help automate key pieces of the software.

In addition to details of the formulation, design, and implementation of the various algorithms and software components, this dissertation also describes the construction of a custom mobile workflow simulator that can be used to conduct simulation experiments that verify the effectiveness of the approaches presented in this document and beyond. Also presented are empirical results obtained using this simulator that show the effectiveness of the described approaches.

Acknowledgments

I would like to thank my advisor, Professor Catalin Roman, who has been a mentor since my undergraduate years at Washington University. His guidance and support was valuable in my research efforts as well as other aspects of academic life. Thanks also go out to Professor Chris Gill who has always been there with advice, help, and support during my graduate career as well as my committee, Professor Yixin Chen, Professor Carl Gunter, Professor Peter Hovmand, and Professor William Smart.

I am grateful to have had the opportunity to work with several talented individuals in the Mobilab. A special thanks to my collaborators - Radu Handorean, Greg Hackmann, Mart Haitjema, and Justin Luner and undergraduate students Andrew Frank, Sean Fellows, and Aayush Munjal for their invaluable help in delivering various projects over the years. Thanks also to the other members of the lab for many lively research discussions and the odd game of "hammer ball."

To the so called "Mobilab adoptees" and members of the lunch club - Sajeeva Pallemulle, Paul Gross, and Haraldur Thorvaldsson - A special thanks for the lively discussions, debates, and conversations which were a much needed offset to the seriousness of pursuing a doctoral degree.

To the staff of the CSE office - Ron Brown, Myrna Harbison, Madeline Hawkins, Andrea Levy, and Sharon Matlock, thank you for the many varied and impromptu conversations in the CSE office, helping with GSA events, monopolizing trivia night prizes, for helping me keep the t's crossed and the i's dotted, and above all for your friendship.

Last and by no means least, I would like to thank my wife, Snehal for her love and for being a co-conspirator in the business of getting doctoral degrees.

Rohan Sen

Washington University in Saint Louis
August 2008

Dedicated to my parents, Snehal, and Vivek.

Contents

List of Figures

Chapter 1

Introduction

Rapid advances in mobile computing technology are redefining the notions of a computing device, the manner in which people interact with such devices, and the places where such interactions take place. Just as the Internet revolution of the 1990's changed computing by linking millions of previously isolated computers to the World Wide Web, so too is mobile computing by taking this computing and communication power from the desk and putting it in users' pockets and purses. The opportunities made possible by a mobile computing device are incredible. For the first time, a computing device is able to perceive different physical and computational environments by virtue of the fact that the device itself is *physically mobile*. Perhaps even more importantly, due to the fact that these devices are typically carried on one's person, the computational and physical environment perceived by the device is the *same* as that perceived by the user. The proximity to the user and the complementary perception of the environment makes it possible for mobile devices to become digital representatives of their users, not only for local computing needs, but also in collaborations with other people *and their devices*.

Mobile devices foster a very different type of usage model. Since the device is easily accessible at all times, the notion of a "working session" where the user's sole focus is on interacting with a computer gives way to a usage model where interactions with the computing device are interspersed and mingled with other activities in the user's life. In addition, instead of focused sessions, usage of the computer occurs sporadically at all times and in many different places, depending on the user's movements over time.

A large fraction of the software built today has been conceptualized with traditional computing environments and usage models in mind which assume powerful desktop

and laptop computers, stable wired network connections, and focused, single user tasks. Initial applications for mobile devices beyond personal information management (PIM) functions sought to take these so-called "desktop" applications, miniaturize them and shoehorn them onto mobile devices. However, as mobile device usage continues to grow and people mature as users, there will be a demand for a new generation of applications that are designed for and specifically tailored to mobile computing platforms. The grand challenge will be to design a new class of software that can exploit the novel environment, usage model, and hardware capabilities of mobile devices to provide value-added services that users cannot expect from traditional computing platforms.

There are, however, several technical challenges that must be addressed in order to achieve the aforementioned goals. The hardware capabilities of mobile devices continue to lag significantly beyond those of traditional computing platforms, imposing constraints on software footprints. Limited screen real estate and restricted (and sometimes counter-intuitive) user interfaces can be off-putting to users and must be managed carefully to present all the necessary information. Finally, the motion of users (and therefore the devices) creates a dynamic network topology which is not a conducive substrate for running complex, distributed applications.

This dissertation describes the design principles, algorithms, software architecture and implementation of one such new class of software applications for mobile devices - *workflow-based* collaboration management systems that allow several users equipped with mobile devices to work collaboratively towards a common goal and have the nuances of their collaboration handled automatically by the software that runs on their respective mobile devices. Users can input a specification describing their collaborative activity, the data and notifications to be exchanged, as well as the overall structure of the collaboration. The system takes this specification and *executes* it, sending notifications and collecting data from the various participants in a structured manner until the entire collaborative activity is completed.

The subsequent sections in this chapter describe mobile devices, applications, and collaboration technology in more detail to set the context for the work in this dissertation. The core contributions of the dissertation are presented in subsequent sections.

1.1 Mobile Devices and Applications - A Brief Historical Perspective

1.1.1 Mobile Hardware

The notion of a mobile device has evolved significantly over the years. In the early 90's, a laptop was considered a mobile device because it was portable and provided computing resources away from the work desk. However, with wireless networking in its infancy, a laptop was tied to a fixed location for its communication needs. The late 90's saw the introduction of the personal digital assistant (PDA) with Palm Inc. offering several models, which redefined the notion of a mobile device. PDA's brought several innovations such as a smaller form factor, limited touch screen user interface, and synchronization with a desktop or laptop computer using a USB dock. Most importantly, it was the first pseudo general purpose computer that could fit in one's pocket.

Early PDAs still suffered from several restrictions however. Colored displays did not appear until later (Handspring being one of the earliest to offer this feature) and high resolution UXGA displays did not become commonplace until the next decade. Wireless network connections (Wi-Fi and Bluetooth) too were not common features in the initial models and the processor architectures were very different from those used in traditional computers. PDAs also suffered from limited system memory (on the order of 10's of megabytes) and lack of storage space for applications and data.

Around the same time that PDAs were becoming popular and adoption rates were increasing, cellular phones too started offering significantly more capabilities. Colored screens, more sophisticated applications, and connectivity over cellular networks resulted in the cellular phone becoming a viable computing platform. In addition, the immense number of cellular phones (approximately 2.5 billion worldwide in 2006 [33] with a further 1.1 billion sold in 2007 [8]) made developing the platform an attractive economic venture.

Today, PDAs and cellular phones have converged to an extent as the Smartphone, a combination of PDA and cellular phone capabilities, though there exist strong markets

for each individual technology. Smartphones run mobile OSs like Symbian, the mobile version of Apple Inc.'s OS X, Google's Android, and offer several applications. In fact, Apple's iTunes App Store and Nokia's N-Series applications represent commercial efforts to have a large library of applications for mobile devices. The mobile device of today is a small form factor device with a high resolution display, approximately 50 to 100 MB of RAM, persistent storage on the order of a few GB via expandable flash-based media, powerful mobility optimized processors operating at a few hundred megahertz, multi-mode communication capabilities via Bluetooth, Wi-Fi, and cellular networks, and a varied and customizable application suite.

1.1.2 Mobile Applications

Applications on mobile devices have been restricted primarily by the processing capabilities of the devices. As such, initial PDAs and cellular phones offered only basic personal information management tools, calculators, converters, etc. With advances in hardware, PDAs in particular were able to offer miniaturized versions of desktop applications such as word processors, spreadsheets, presentation viewers (Documents To Go, Microsoft MobileOffice), to name a few. Over time, these miniaturized applications grew more and more sophisticated, implementing many additional features.

As communication hardware became a standard feature of mobile devices, applications evolved to exploit this feature. Applications now offered synchronization of data over the air, web browsers (MobileIE, OperaMini), email clients (OutlookMobile, Apple Mail, Mobile Safari), instant messaging (mobile versions of AOL IM, MSN Messenger, GoogleTalk, and Yahoo Messenger), and RSS feed readers. The advent of the so-called Web 2.0 resulted in mobile devices offering various social networking (Facebook, MySpace), and user-created content functions (live mobile blogging). Today, the boundaries are being pushed further with location-based services (geo-tagging of photos in Google's Picasa and Apple's Mobile iPhoto) and applications which seek to customize the user experience according to one's instantaneous location (Zagat restaurant guide etc.)

It is worth noting however that in today's applications, the communication capabilities of a phone or PDA are being used solely to reach into the network core and

interact in a manner that is very similar to a user browsing the web on a desktop PC, albeit mobile device users can move around while accessing this content. There is very little attention being paid to applications that can utilize the communication capabilities of the devices to interact with others in the immediate vicinity *directly* in an ad hoc manner. Such peer-to-peer interactions are the ideal platform for building collaborative applications, which is the topic of this dissertation.

1.2 Introducing Collaborative Technology

A human being typically interacts with several other human beings during the course of the day and a fair fraction of these interactions are in person, or to be more precise, with another human being in the immediate physical and temporal vicinity. Mobile devices connected directly to other devices in proximity in a peer-to-peer manner are digital mirrors of the human interaction and can help support and manage such interactions. This observation is the basis for engineering collaborative applications for mobile devices.

Collaborative work is not necessarily restricted to people working with others in their proximity. Rather, collaborations can be defined more expansively to be an activity which requires the skill and input of multiple people in order to be completed successfully. As such, collaborative applications have a very different structure and user interaction pattern compared to single-user applications. The following example contrasts a reference single-user application with its collaborative version.

In a modern word processor, only one person may work on a document at any given time. There is no scope for two people to work on the document simultaneously or any facility by which notifications are traded when one user wants the other user to take over or offer feedback. It may be argued that this paints a rather bleak picture. There are, after all, versioning systems that support multiple users editing separate copies of a document simultaneously and then merging the changes. There are also ways to comment on documents, highlight changes, and share them using a workspace such as in Groove [61]. However, it should be observed that in each of these cases, the burden of taking the steps to ensure a smooth collaboration falls on the user. Also,

most of the functionality described above are external, i.e., they are not built into the word processor itself.

Consider a similar example where a group of people are working on writing an academic paper. Each person is responsible for certain sections and the people writing the later sections must have the text of the earlier sections as a reference. One way this can be done today is by exchanging several emails to determine who is responsible for which section. Once this is done, the person writing the first section generates the text using a word processor, saves the document, and emails it to the next person in line. Alternately, he saves the document to a common space and then emails the next person in line that he/she can now access the document.

Consider the alternative using more advanced collaborative technologies. Initially, a shared space is created with the pieces of work being represented graphically. All participating authors connect to the shared space and claim their "piece" simply by clicking on the graphical representations of each work task. A chat window is available to discuss any conflicts or details. Once the work is divided, the appropriate author begins writing the first section. When the section is finished, the author simply selects a "save and notify" option rather than simply "save". This action saves the document and notifies the next person in line that the document is ready for his/her attention.

It should be apparent from reading the description above that the structured nature of the collaborative activity helps define the actions that the underlying software technology must take in order to facilitate the smooth progress of the activity involving multiple people. One example of such an underlying software technology that is well suited to supporting structured collaborations is workflow technology. This dissertation uses workflow technology to build collaborative systems for mobile devices.

1.3 State of the Art in Workflow Management

The basic workflow model has been in use for several decades and some forms of it predate even the earliest computers. Non-computerized workflows have been used extensively in a wide variety of settings such as film production, industrial manufacturing, and inventory management, to name a few. As computers began to play

an increasingly larger role in business enterprises, and tasks previously performed by people became the domain of machines, a need was perceived for software systems that could execute electronic versions of workflows. This gave rise to the notion of business process management, where business processes such as a loan approvals, insurance claims processing, etc. were encoded as a workflow, and a *workflow management system* (WfMS) invoked several software components in a structured manner to *execute* such business processes.

Currently, there are several commercially available workflow management systems (WfMSs) that are designed for the World Wide Web or enterprise LANs such as Oracle 9i Workflow [74], ActiveVOS [3], JBoss [46], BizTalk [62], and i-Flow [26]. While there are some differences in features (a summary of which appears in [106]), they are all designed as centralized systems with the WfMS resident on a central server, invoking software *services* synchronously across the LAN or the Internet as required to advance the state of the business process.

While these systems represent mature and proven technology in wired settings, there are significant challenges in getting such systems to work across mobile networks. As with all software designed with stable settings in mind, WfMSs of today suffer from the fact that they are centralized and monolithic pieces of software that cannot easily adapt to a more fluid mobile setting. However, building a mobile WfMS is an important endeavor since it is the lack of such a WfMS that prevents the use of workflows in more dynamic and mobile settings. As indicated earlier in this section, workflows have historically been used in a wide variety of contexts and there is no factor (other than the lack of a suitable WfMS) that prevents their use for supporting collaborations involving humans and software services across mobile networks in the physical world.

1.4 Problems with Bringing Workflows to Mobile Environments

The work presented in this dissertation tackles the problems associated with bringing WfMSs to mobile settings and specifically to MANETs. The work is motivated by the

fact that mobile devices represent the next big paradigm shift in computing technology and enjoy unsurpassed ubiquity across the world. However, successfully transitioning a WfMS to a mobile setting will require solving several research, intellectual, and engineering challenges, which can be summarized into three broad categories:

- **Context Knowledge.** In a dynamic network that extends over the physical world, knowledge about both the physical and computational environment is key to shaping the execution of the workflow and making decisions that result in the fewest number of errors. Gathering, storing, trading, and using this knowledge requires a sophisticated software infrastructure that can deliver the required information reliably.

- **Decentralization.** In mobile networks, where the participants exhibit physical motion, the dependence on any centralized resource is risky since that resource could move away and become unavailable. Any WfMS that must survive in such an environment must be built in a decentralized and distributed manner without compromising its consistency and integrity.

- **Uncertainty.** Human behavior can change in unpredictable ways and unanticipated phenomena in the physical environment may affect the system. The system must be able to deal with unpredictable changes and advance the workflow execution in exceptional circumstances.

In addition to these problems, a viable approach must contend with the fact that both hardware and software for mobile devices are excessively fragmented with there being numerous incompatible architectures, languages, programming tools, and execution environments. While the artificial barriers to compatibility exist, the true potential of mobile computing can never be realized. Initial steps are being taken to rectify this problem [13, 65, 90] including an effort to standardize the computational environment of mobile devices (Google's Android project [72]). The solutions described in subsequent chapters address these problems through their design and architecture.

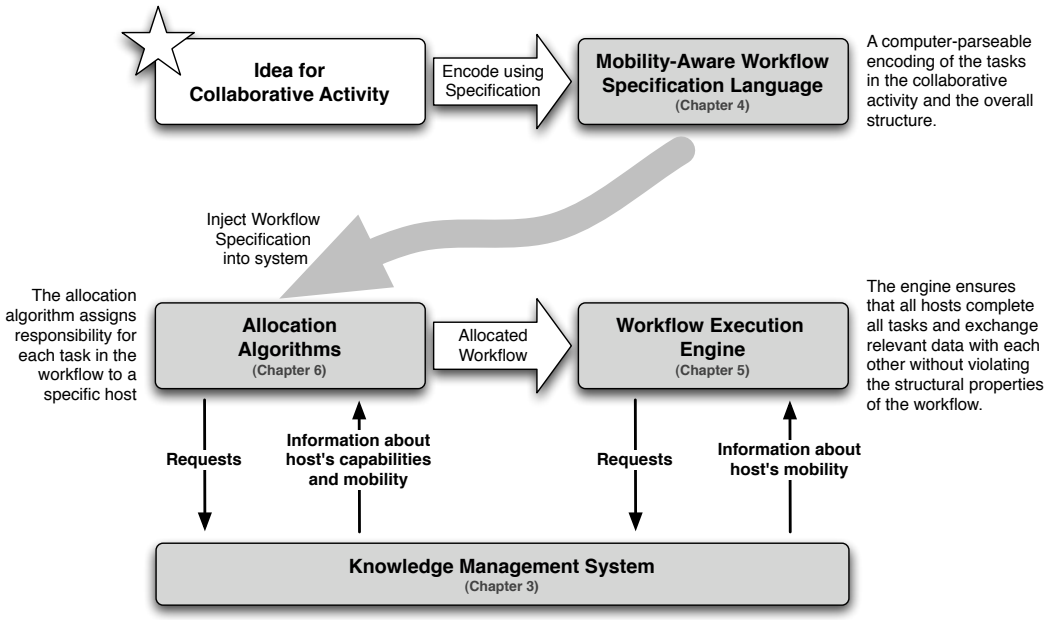

Figure 1.1: Relation between key contributions of this thesis

1.5 Contributions and Dissertation Overview

This dissertation is concerned with designing and building a workflow management system for mobile devices and MANETs. The aim of such an effort is to enable the application of workflow technology to model computer assisted collaborations that go beyond business processes executing on a server machine and encompass person to person collaborations taking place in the physical world. The contributions of this dissertation can be divided into four key parts, the relationships between which are shown in Figure 1.1. Each of these four contributions is described in detail below.

Mobility-Aware Workflow Specification Language. Any WfMS takes as input a workflow that is encoded using a workflow specification language. For workflows in mobile settings, it is important to augment a "traditional" workflow specification language with constructs that allow the description of spatiotemporal constraints and workflow behavior in response to contextual changes, as well as constructs that allow simple decomposition and reconstruction of the specification in response to the demands imposed by network conditions. The CiAN specification language is a workflow specification language that has been designed with mobility in mind and is

based on the most fundamental representation of a workflow - the directed graph. The CiAN specification language provides constructs for describing fragmentable, flexible, and context-aware workflows for mobile environments and is presented in Chapter 4.

Knowledge Management System. A knowledge management system makes it easy for applications and middleware platforms to exchange *relevant* information across the MANET with the idea that the possession of such information by other participants in the network will improve decision making and performance of the system as a whole. The knowledge management system trades information about participants in the network using a gossiping protocol. This information is then stored in a knowledge base organized by host and then by parameter, where a parameter is a property (functional or non-functional) of a participant. The contents of the knowledge base are made available to locally executing programs which may then use the information to their benefit. The knowledge management system is described in Chapter 3.

Allocation Algorithms. A workflow by definition consists of several smaller tasks which must be executed by multiple participants, each possibly having a different set of skills. The process by which the tasks in the workflow are assigned to *suitable* participants is called *allocation*. The allocation process is especially crucial in a system designed to run across a MANET because an incorrect decision can seldom be reversed due to the relevant hosts being unreachable for significant periods of time. Chapter 6 describes two allocation algorithms. The first is a baseline *a priori* centralized approach that is mobility-aware and uses heuristics to limit backtracking. The second is a just-in-time distributed approach that uses a system of bids and measures of fitness to determine task allocations.

Workflow Execution Engine. Once a workflow has been allocated, it must be executed and managed by a runtime system. The CiAN workflow management system is a WfMS designed for MANETs. It advances the state of the art in WfMSs by providing what is possibly the first WfMS that is designed to run in a completely distributed and disconnected manner across MANETs with no dependence on any external resources. This is made possible by CiAN's *filter-based* architecture which supports distributed workflow management and a hybrid host-agnostic communication protocol that uses a combination of store-and-forward and publish-subscribe

approaches to enable disconnected communication across the MANET. The CiAN workflow management system is described in Chapter 5.

The significance of bringing workflows to mobile environments is that it opens the door for applying workflow technology to model arbitrary structured collaborations anytime and anywhere. By untethering workflows from the wired network environment, this dissertation makes it possible for workflows to play a significant role in computer assisted collaboration software. Such software could be used in novel applications such as construction management, geological surveys, management of outdoor hospitality events, coordinating the renovation of a home, and much more.

The next chapter provides background information on the work in this dissertation, followed by chapters that present the core technical contributions described above. In addition to those contributions, this thesis also describes the design and implementation of a workflow simulator for MANETs in Chapter 7. This simulator has been designed in a generic manner to empirically evaluate algorithms and protocols relating to workflow management in MANETs. The simulator was used to generate the results presented in certain chapters in this document. Chapter 8 describes future extensions to the work described here before concluding remarks are given in Chapter 9.

Chapter 2

Background & State of the Art

The work presented in this dissertation, the context for which was set in Chapter 1, is concerned with building a workflow-based collaboration platform that can function across mobile ad hoc networks (MANETs). The resultant workflow management system (WfMS) relies on several existing technologies including coordination models, service-oriented computing, and collaborative computing. This chapter expands upon the presentation of Chapter 1, adding the necessary technical background information relevant to the remainder of the document.

Since the work presented in this dissertation is designed for operation across MANETs, the first section of this chapter describes the unique characteristics, constraints, and challenges of the MANET environment. This is followed by a presentation of how coordination technology can be used to design basic communication infrastructures for such networks. Next is a description of service-oriented computing and the manner in which mobile versions of this paradigm have been used to promote sharing of software resources among capability-constrained mobile devices. The presentation is concluded with a discussion of collaborative systems that compose the capabilities of individual service-oriented computing systems and their human users to provide collaborative computer supported problem solving capabilities.

2.1 Mobile Ad Hoc Networks

Ad hoc networks are a special type of network that are formed opportunistically among groups of hosts. Typically, ad hoc networks do not depend on any fixed

external resources–the infrastructure of the ad hoc network is borne completely by the hosts that comprise it. Due to the opportunistic manner in which the network is formed, the composition of the network may evolve rapidly over time, with hosts entering and leaving the network at will, which in turn results in a very dynamic network topology. Wireless sensor networks [20] are one type of ad hoc network consisting of several sensor nodes. Each sensor node has a duty cycle and the network at any time covers only those sensor nodes that are not in sleep mode.

Mobile ad hoc networks (MANETs) are ad hoc networks that are formed between physically mobile hosts that are within communication range of each other. This physical mobility adds yet another degree of dynamism to the ad hoc network, and in combination with the relatively restricted communication range of mobile devices creates an environment where the network topology is highly unsettled, making it almost impossible to offer guarantees for being able to communicate with another host in the network. In this type of network, wireless connections and communication windows between hosts are transient which leads to a highly decoupled style of computing. This lack of ability to communicate with other hosts as needed also forces the use of decentralized and redundant software architectures as opposed to centralized architectures with a single point of failure.

MANETs have received a fair degree of attention from the research community. The Internet Engineering Task Force (IETF) established the MANET working group to *standardize IP routing protocol functionality suitable for wireless routing application within both static and dynamic topologies with increased dynamics due to node motion or other factors* [44]. This group has published several drafts and RFC's [45] organized as shown in [15]. MANETs have also featured prominently in conferences such as Mobihoc and COORDINATION. While the role of MANETs has diminished somewhat due to widely available cellular connectivity, there are still many application areas in which MANETs are a necessity, especially in very remote areas, developing nations, military operations, or in circumstances where communication is very localized and a cellular uplink is not required, e.g., vehicular ad hoc networks, etc. As such, MANETs represent a challenging and relevant area of research even in the context of alternate, more pervasive communication media.

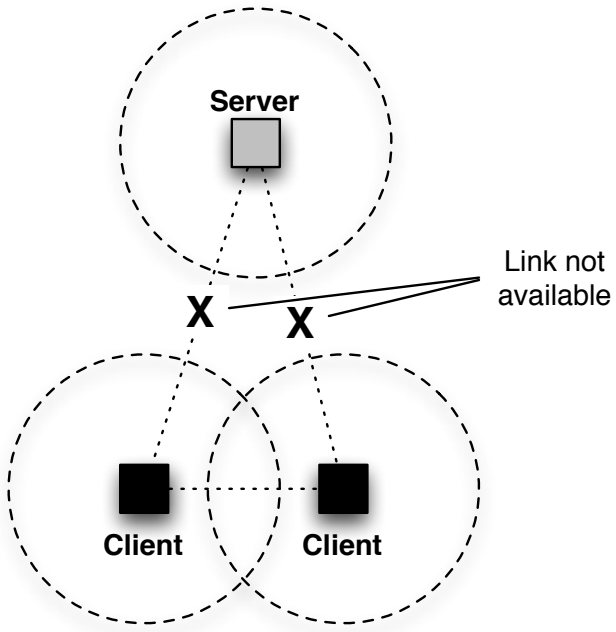

Figure 2.1: Shortcomings of a Centralized Architecture in MANETs

The unique environment of MANETs forces software designers to think very differently about modeling, designing, and implementing software that can survive in such situations. The following is a listing of the core design philosophies that must be taken into account when programming for a MANET environment:

- **Decentralization.** Any distributed application running across a MANET must be designed with a decentralized architecture and engineered to operate in a peer-to-peer manner. In traditional client-server architectures, clients coordinate with each other through a central server. In MANETs, the central server may become unavailable for periods of time because it is not within communication range of some or all clients (see Figure 2.1). This creates a situation where clients may be within communication range of each other but cannot interact due to the lack of access to a coordinating server. Centralized architectures in effect create a single point of failure which is highly undesirable in MANETs. This problem can be eliminated to an extent by having redundant servers but this too is a partial solution since hosts would need to be able to reach at least

one of the servers (by no means guaranteed in a dynamic network) and in addition, the system has the burden of keeping the various redundant replicas synchronized over the unstable network. In the work presented in this dissertation, the principle of decentralization has been applied both to the design of the knowledge management system in Chapter 3 and the workflow management system in Chapter 5.

- **Redundancy.** Any action taken or message sent across a MANET may get dropped due to the communication link between host pairs breaking at an inopportune moment. In wired networks, where the likelihood of communication links going down is relatively very small, the existence of a reliable communication protocol like TCP/IP suffices to ensure that data can be delivered from one host to another with minimal risk of loss. In a MANET where the hosts' mobility results in routes being available for very short periods of time, the likelihood of a message getting lost or dropped is very high. Hence, it is always advisable to build additional redundant mechanisms for coordinating among hosts so that the likelihood of system failure due to the failed delivery of a message or other communication is minimized. Such redundancy principles are used in the publish-subscribe-based communication protocols used by the workflow management system described in Chapter 5.

- **Mobility Awareness.** Any application that is aware of the mobility patterns of the hosts in the MANET and exploits this information is likely to have a more successful execution compared to ones that do not. By definition, communication windows in a MANET are opportunistic which means that disconnections from other hosts can occur unexpectedly at any time. The effect of this on applications is that they cannot offer any guarantees as progress is predicated on disconnections not occurring at inopportune times. Applications that are aware of the mobility pattern of the host on which they are resident and other hosts in the network can use this information to compute when communication windows are likely to be available and when disconnections are likely to occur. This in turn can be used to either schedule communication within available communication windows (and therefore offer some basic guarantees based on available windows) or notify hosts that certain messages cannot be delivered due to the

lack of a suitable window (rather than attempting the transmission and waiting for a timeout). Mobility awareness is present in each and every aspect of the work presented in this dissertation ranging from the software support for awareness presented in Chapter 3 to specifications and algorithms designed with mobility awareness in mind presented in Chapter 4 and 6 respectively.

- **Flexible Communication.** Communication middleware built for MANETs must be flexible and robust against disconnections. In socket-based communication, if either the sender or the receiver disconnects, an exception occurs and the socket must be recreated. In MANETs, where disconnections are commonplace, such an approach can result in the system spending a large amount of time recovering from errors. Communication middleware designed for MANETs must catch these errors and recover gracefully from them, all the time shielding the application from such low level communication issues. This principle is applied in the design of the software system and communication protocols described in Chapter 5.

2.2 Coordination Technology

The volatility of a MANET can result in frequent breakdowns in communication links which in turn affect applications. Coordination technology seeks to separate the behavior of a program from its interactions to improve modularity. In other words, coordination is concerned with separating computing from communication. Coordination technology has several applications in a wide range of environments beyond MANETs. However, coordination concepts are extremely relevant in MANETs as one of the aims in MANET programming is to shield the applications from disruptions in communication and similar low level details. A brief survey of coordination technology is included here for completeness as the knowledge management system presented in Chapter 3 as well as the algorithms presented in Chapter 6 of this dissertation use coordination concepts in their design and their software architecture.

Typically, a coordination model is characterized by its use of a shared dataspace that offers the following operations: (1) `out`, which places data in the shared space, (2) `in` which removes data from the shared space, and (3) `rd` which creates a local copy

of some data in the shared space. An agent that wishes to interact with another agent places data in the shared space which is subsequently retrieved by the target agent, thereby completing the interaction. The first example of a coordination model was Linda [28]. In Linda, coordination is characterized by a centralized coordination mechanism while the application that uses it may be distributed. In modern implementations of the coordination concept, such as JavaSpaces [101] and TSpaces [114], various parts of the application coordinate with each other by means of a tuple space maintained at a central location.

Coordination models have also found favor in agent-based systems. TuCSoN [71] introduced multiple tuple spaces, called tuple centers while in MARS [12], mobile agents are provided upon arrival on a particular host with a handle to the local tuple space, which is shared among all agents present on the same host. Ara [82] introduced a constrained rendezvous type of coordination: some agents assume the role of coordination servers and represent meeting points where agents can ask for services.

More recently, coordination models have been adapted to novel computational environments [79], [78], [11], and [25], which highlights their versatility. LIME [66] was the first piece of work that brought coordination technology to MANETs. LIME proposed the idea of multiple (local) tuple spaces that were transiently shared to form a federated shared dataspace when hosts are in communication range. Limone [24] is a lightweight alternative to LIME implemented to offer fewer guarantees. TOTA [56] uses spatially distributed tuples, injected in the networks and propagated according to application specific patterns.

Coordination models adapted for MANETs often support only peer-to-peer connections. To support multi-hop connections, they need to be combined with MANET routing protocols which fall into four broad categories: (1) proactive protocols such as DSDV[83], WRP[67], CSGR[16], which constantly maintain and update routes using routing tables at the cost of high bandwidth usage; (2) reactive protocols such as AODV[84], TORA[80], ABR[104], DSR[47], which only search for routes when they are required at the cost of low responsiveness; (3) hybrid protocols such as ZRP [34]

that try to balance the trade offs between the first two approaches, and (4) disconnected routing such as Epidemic[105] and Message Relay[52], which allow messages to be sent via a gossiping protocol.

2.3 Service Oriented Computing

While coordination technology can be used to communicate across a MANET, service-oriented computing (SOC) can be used as a means to share *functionality* and *capabilities* of hosts across the network. Service-oriented computing is an evolution of object-oriented and component-oriented computing that seeks to encapsulate software functionality using standardized wrappers. These wrapped software components are referred to as services and clients may exploit these services using flexible interfaces that are specified at a high level. A service-oriented computing framework is a conglomerate of elements, each element fulfilling a very specific role in the overall framework. The salient elements required for a viable service-oriented computing framework are as follows:

- The *service description* element is responsible for describing a service in a comprehensive, unambiguous manner that is machine interpretable to facilitate automation, and human-readable to facilitate rapid formulation by users.
- The *service advertisement* element is responsible for advertising a given service description on a directory service or directly to other hosts in the network. The effectiveness of an advertisement is measured as a combination of the extent of its outreach and the specificity of information it provides up front about a service, which can help determine whether a user would like to exploit that service.
- The *service discovery* element is the keystone of any service-oriented computing framework and carries out three main functions. It *formulates* a request, which is a description of the needs of a user. This request is formatted in a similar manner to the service description. This element also provides a *matching function* that pairs requests with similar service descriptions. Finally, it provides a mechanism for the user to *communicate* with the service provider.

- The *service invocation* element is responsible for facilitating the use of a service. Its functions include transmitting commands from the user to the service provider and receiving results. It is also responsible for maintaining the connection between the user and the provider for the duration of their interaction.
- The *service composition* element provides mechanisms to merge two or more services into a single composite service, which combines and enhances the functions of the services being composed.

In a SOC framework, a *service provider* initially advertises some functionality (*service*) it wishes to share by placing a *service advertisement* in a publicly available *service directory*. *Clients* discover and use these functionalities at run time. In proxy-based SOC, the service advertisement is in the form of a proxy object which can be retrieved by clients and used as a local handle to the service process on a remote server. The client interacts with the proxy as it would with any other local resource. The proxy tunnels requests to the provider's server in most cases. However, in some cases, the proxy itself can deliver the entire functionality of the requested service and does not need to tunnel client calls to its parent server. From the application programmer's perspective, the proxy-server communication is abstracted from the client but is relevant to the middleware programmer. Jini [110] is an implementation of the proxy model, targeted towards wired and fairly reliable networks. This is evidenced by its use of a centralized service directory, which is a single point of failure for the system.

In Web services (WS), which is the dominant service-oriented architecture (SOA) for wired network environments, service advertisements in the form of Uniform Resource Identifiers (URI) are placed in a well known centralized Universal Description Discovery & Integration (UDDI) [69] directory. Clients can search the directory for a service using associated UDDI protocols, retrieve the URI for the service, and then connect directly to the service at its specified URI using the Simple Object Access Protocol (SOAP) [116]. Since the client interacts directly with the service, it requires the client to be aware of the interaction protocol and the location of the service. Strict enforcement of standards means that clients have to specify their requests in a standardized format and syntax, and also follow standardized protocols for interaction with the service. A collection of languages, all of which use XML [115] syntax,

are used for client-service interactions. The lowest level language is the Resource Description Framework (RDF) [87], which provides a basic set of constructs to describe entities and relations. The Web services Description Language (WSDL) [113], describes the actual functionality of the service. SOAP describes what is being sent across the wires and encapsulates all application layer protocol issues. Higher level languages describe why a piece of data is being sent between two hosts. High level description languages are required to be clear and concise, and to support easy matching between two entities. Ontologies are high level languages that capture the semantics of an entity and its relation to other entities. An ontology is often itself structured into layers. DAML+OIL [43] is a combination of a markup language to construct ontologies (DAML) and an ontology inference layer (OIL) to interpret the semantics of the description. A popular ontology for Web services is OWL [111, 57].

The major difference between Web services and proxy-based models is that in Web services, a service is essentially a process running on a well known remote host and it is up to the client to know the correct protocol to contact and use the server. However, in proxy-based systems, a service is a combination of a process running on a remote server and the proxy that it ships to the client. The client therefore needs to only make local method calls to interact with the remote service. Despite the differences, both approaches are targeted to wired infrastructures, evidenced by centralized architectures that are not viable in more dynamic networks. A separate research effort that adapted these approaches to MANETs is described in [39].

While SOC is not the focus of this dissertation, it does play an integral role in the work presented here. The knowledge management system described in Chapter 3 was originally designed for a SOC framework for MANETs [39] and was subsequently generalized for use in the WfMS presented in Chapter 5. Services also play a key role in the workflow management system as the tasks in a workflow are ultimately satisfied by a software service *or* a human user. The principles of service advertisement, discovery, and invocation are all used within the context of the WfMS. Indeed, the WfMS is one type of mechanism supporting service composition.

2.4 Collaborative Computing

Computer Supported Collaborative Work (CSCW) seeks to solve problems associated with building systems in which multiple users collaboratively solve a single problem. The aim of such collaborative systems is to leverage the power of computational tools to simplify person-to-person collaboration, automate certain mundane tasks, and reduce the overhead associated with collaboration. The question as to the exact definition of a collaborative system is debateable. According to some definitions, devices like telephones, software like email clients, file sharing clients, etc. are collaborative systems since they allow sharing of information. In modern computing systems however, the definition is being narrowed to cover only sophisticated systems that provide end-to-end collaboration facilities [60]. The focus of this section will be solely on such systems, which can be further broken down into three categories:

- **Workspace Based Systems.** Workspace-based systems such as [61, 75, 73, 4] are based around the notion of a shared structure (the workspace) which contains all the necessary resources to perform a collaborative activity. Users are given controlled access to the resources within the shared structure and systems are in place to control versioning of documents and other such necessities. Workspace-based systems typically split their functionality into modules such as messaging, e-mail, document management, video conferencing, etc. The onus is on the user to use these features in the manner that best supports the collaboration. Typically, the workspace is located on a central server with users all seeing a consistent view. Some systems allow for temporary disconnection from the central server (nomadic mobility) during which the user has a local copy of the workspace available. When a user reconnects, the views of the workspaces are synchronized.

- **Ad hoc Processes.** In certain collaborative activities, the goal of the activity is well known but the exact manner in which the goal will be realized is not clear. For example, in the collaborative design of a software system, the end goal is a design document which captures the discussion. However, it is not clear in which direction the discussion might go and how the discussion might evolve. Ad hoc processes [19, 10] capture precisely such collaborations, allowing the specification of the activity in terms of high level goals and checkpoints

and leaving the choice of the actual strategy for realizing those goals until runtime. To date, there have not been many systems supporting ad hoc processes primarily due to the fact that ad hoc processes rely heavily on semantics to translate between high level goals and actual strategies, which has not yet been developed to an extent to be feasible.

- **Synchronous and Structured Processes.** For collaborations where the goal as well as the path to the goal is well-defined, the use of synchronous and structured processes is preferred. In these cases, the activity is typically decomposed into smaller tasks that must be completed in a certain order to achieve the requisite goals. Structured processes are the least complex type of system to implement but offer the least flexibility. Introducing an additional degree of flexibility and migrating such processes to mobile networks is the focus of this dissertation.

The work presented in this dissertation seeks to build a system that executes structured collaborative process across a MANET. More specifically, this dissertation describes (in Chapter 5) systems that execute *workflows*. Workflows can be conceptualized as a graph where the vertices represent tasks that must be completed and the edges provide ordering among tasks and overall structures. The workflow as a whole represents the collaborative activity and typically has a lattice structure. The collaborative aspect of the workflow comes from the fact that different hosts can be assigned (or take on) responsibility for different tasks in the workflow (described in detail in Chapter 6).

For the work presented in this dissertation, it is assumed that there exists a group of human users, each of whom is equipped with a mobile computing device such as an ultramobile PC, PDA, or smartphone. All users are assumed to co-located initially. Since the devices are carried on the person of the users, it is assumed that they are physically mobile and that their motion pattern is the same as their associated user. The devices are capable of communicating with each other using 802.11b/g/n radios when they are within communication range of each other. However, such *windows of communication* [37] (the intervals of time during which a pair of hosts are within range) may be transient due to mobility.

Each host that participates in the execution of a workflow provides the following information about itself: 1) The name of the host, assumed to be unique in the network, 2) a schedule, which indicates time intervals during which the host is not available and cannot participate in the workflow execution, 3) the location of the host before and after any time interval during which it is not available, and 4) the list of services available on the host. Note that the list of services includes both software services as well as services that represent the skills of the associated user that can be exploited without the use of a computer (e.g., a metal welding skill). These pieces of information are timestamped when they are created and traded freely in the network using a gossipping protocol. Hosts store this information about other hosts in a local knowledge base along with any additional non-functional information that a host may choose to share (such as its current location, velocity, battery levels, etc.). The contents of the knowledge base can be queried by other middleware components. Finally, hosts may have sensors attached to them, e.g., a GPS receiver, which can also be queried by the middleware.

Recall that a workflow is used to model a structured activity which can be divided into smaller tasks. A workflow language on the other hand, provides a textual representation of the workflow specification. The challenges in developing an appropriate workflow language lie in determining the constructs that are provided, the degree of flexibility allowed in the specification, the ability to fragment the specification for easy distribution, and adopting a format that is both human readable and machine parse-able.

After a workflow is specified, the next step is to determine *how* and *to whom* to distribute each task based on individual properties of hosts and their motion pattern. This process is called *allocation* and it is necessary to do this before execution begins because at later stages, hosts may not be in communication range and hence may not receive requests to perform tasks in the workflow and end up stalling the workflow execution as a whole. The allocation process requires the development of complex scheduling algorithms.

Once the workflow is allocated, it must be executed by WfMS. A task in the workflow is considered to be ready for execution when a valid set of inputs are available. The first task in the workflow has no inputs and is considered to be always ready. If

the task involves execution of a software service, it is handled automatically by the middleware. If not, the middleware prompts the user to take actions that fulfil the requirements of the task. When the task is completed, the notification of completion and any relevant data are sent to succeeding tasks as dictated by the structure of the workflow. The challenge at this stage is to provide an execution engine that can 1) manage the execution of the workflow in a distributed fashion and 2) communicate notifications and data to succeeding tasks over the dynamic MANET.

The next chapter describes the knowledge base, followed by chapters which describe the specification language, execution engine, and allocation algorithms, respectively.

Chapter 3

The Use of Knowledge in Mobile Environments

By definition, a mobile ad hoc network (MANET) has a dynamic network topology which gives rise to frequent and unpredictable disconnections between pairs of hosts. These unanticipated disconnections can cripple the execution of distributed applications across hosts in the network by occurring at the most inopportune moments. It is therefore critical that mechanisms are put in place to prevent or mitigate the effects of these disconnections.

3.1 Motivation

MANETs represent an exciting new frontier for mobile computing which present unique engineering challenges. In MANETs, the typical host is a portable mobile device that is constrained in terms of processing power, memory, and battery life. The network itself is volatile and its topology evolves rapidly, making disconnections a fact of life. Two factors affect disconnections in a MANET. The first of these is the limited communication range of current generation radios. A typical current generation wireless radio has a range between 25 to 200 meters (depending on the environment and obstructions in the area). It may be argued that advances in radio technology that improve ranges would reduce the likelihood of disconnection. While this observation is correct, it still stands that if the radio range is significantly lesser than the area spanned by the MANET, the problems of unanticipated disconnections will still persist. The second cause of the frequent disconnections is the physical

mobility of hosts. The physical movement of hosts results in them constantly coming into and going out of communication range of each other. Since mobility patterns are neither periodic nor known to the applications operating across the network, these disconnections are almost always unanticipated.

A distributed application executing across a MANET is at the mercy of these unanticipated disconnections. If the disconnections occur at an inopportune moment, the application may, in the best case scenario, lose a significant amount of partially completed work or at worst, crash due to errors caused by the disconnection. It is therefore imperative that mechanisms be put in place to mitigate the effects of such disconnections.

Current solutions to this problem involve attempting to eliminate the possibility of disconnection. For example, in a *hoarding* strategy, the code for an entire service implementation is copied to the client machine and is used locally so that disconnections cease to be a factor. However this strategy cannot be employed in cases when the software footprint is very large or when the code is proprietary. *Nomadic* strategies rely on the fact that mobile hosts for the most part will stay within communication range of a set of access points attached to a wired infrastructure and only move out of range for short periods of time, which results in longer periods of connectivity and to some extent, offers a guarantee of eventual reconnection. Such strategies are limiting, as they impose bounds on the physical mobility of hosts (hosts cannot move too far from the access point for extended periods of time) which is often neither practical nor desirable.

The approach presented in this chapter does not seek to eliminate the possibility of disconnection. Instead, it focuses on choosing partner hosts carefully so that a disconnection does not occur at a critical juncture. This chapter introduces knowledge-driven interactions between hosts in MANETs as a strategy that exploits the spatiotemporal aspects of a distributed application's requirements to carefully select partner hosts that are likely to remain in communication range for the necessary duration. This strategy thus reduces the likelihood of a disconnection at a crucial juncture, without compromising other aspects of program execution or host operation. The essential idea is to use knowledge about hosts' physical motions to compute the time at which two hosts are likely to be within communication range for a reasonable interval of

time. This information is then matched with the application's *requirement profile*, which is specified *a priori* by the application programmer and is a list of applications to be provided by remote hosts and the intervals of time during which those applications are required. The result is a proactively planned *satisfying set*, a list of specific peer applications that are resident on hosts that are most likely to be within communication range of the client at the times that they are required, and which will remain within communication range for the projected duration of that need.

The theoretical solutions presented in the initial portion of this chapter have been implemented in the context of a service-oriented system for MANETs called SPAWN [39] to show proof of concept. The same approach can be used in a wide variety of contexts and forms an integral part of the workflow engine described in Chapter 5 and the allocation algorithms described in Chapter 6.

3.2 Background & Related Work

The work presented in subsequent sections is implemented in the context of a service-oriented computing (SOC) middleware for MANETs. In the interest of a complete and well-rounded presentation, a brief overview of SOC is provided here along with an explanation of why a proxy-based SOC architecture was chosen for the SPAWN system.

In the SOC paradigm, a *service provider* advertises services, which represent some capability that it is willing to share. These *service advertisements* are placed in a publicly accessible *service directory*. Interested clients browse this service directory for suitable services. When an appropriate service is found, the client obtains an address (potentially a local handle) for the service. The client can then use the service directly.

3.2.1 The Case for Proxy-based SOC for MANETs

Most SOC architectures, such as the Service Location Protocol (SLP) [50], Salutation [91], and those employed for Web services (WS) [5, 77] are designed for infrastructure-rich centralized wired networks and, as such, are unsuitable for MANETs. For example, in WS, the service directory is maintained at a central well-known location. This is acceptable in the environment of the World Wide Web but unacceptable in a MANET where a particular host may not be accessible to all other hosts as it can move out of communication range or shut down. In addition, it is not reasonable to impose on a single, resource poor host, the responsibility for handling all directory functions.

The addressing scheme used in WS is also geared toward a reliable network. WS uses a uniform resource identifier (URI) to indicate the logical location at which a service may be accessed, and relies on the DNS infrastructure of the Internet to map the address to a physical machine. In MANETs, having centralized DNS servers is impractical and a logical address may not necessarily map to a device at a particular physical location due to the movement of hosts, thus rendering this approach ineffective. SLP takes some steps towards accommodating mobility through a special mode of operation that does not require a functioning directory agent, which is analogous to a distributed service directory. In this mode, services and clients directly seek each other out using a peer-to-peer model. However, this approach still does not solve problems of addressing and service access.

Previous work in this area [95] has shown that the most effective kind of SOC model for MANETs is a proxy-based model. In a proxy-based model, a proxy object is included in the service advertisement. When a client retrieves the advertisement, it obtains the proxy which it installs locally. The proxy then becomes a local handle to the service on a remote machine. The client can then interact with the service by making local method calls on the proxy, which then delegates the requests to its parent service. The idea of proxy-based service oriented architectures was first proposed in Jini [110] which was designed for wired networks. It has since been adapted for use in MANETs as was shown in [38].

Proxy-based architectures are especially effective in MANETs for two reasons: (a) They help reduce the amount of software required on the client side, thereby making thin clients possible; and (b) they abstract details of the protocol to be used between the client and the service provider. Since the proxy is a self-contained piece of code that can communicate with the provider host, the client is not required to be aware of the communication protocol or to possess any other code to communicate with the provider. The client is only required to carry the code that allows it to browse for services and discover proxies. This results in a small footprint for the client software, which is useful when running such software on mobile devices. The fact that proxies abstract the communication protocol is especially useful in MANETs, where standardized application level protocols are not prevalent. Hence, the abstraction of a heterogeneous set of protocols by proxies allows a large set of hosts to communicate with multiple service providers without the overhead of needing to know the specific protocol for each provider.

Though proxies are a solution to certain problems associated with SOC in MANETs, their usage does raise certain issues, some of which have been addressed in previous work. In [40], the authors proposed an automatic code management system which transparently ships and installs proxy code on the client host (once the client has declared an interest in the service) as a solution to the problem of distributing the binary code required by clients to execute proxies. A proxy upgrade system [94] ensures that these proxies are upgraded transparently at run-time with very little impact on the client application, so that the proxy software is kept consistent with upgrades to the software on the provider host without the client application having to handle this procedure explicitly. These mechanisms, which are portions of a larger system supporting SOC in ad hoc networks solve some of the issues associated with proxy-based SOC architectures. The important remaining problem of ensuring that interactions between the proxy and the service are interrupted to the least extent possible is the subject of this chapter.

3.2.2 Related Work

Ensuring that interactions between the proxy and its associated service are interrupted as few times as possible is the responsibility of a knowledge management

system. Knowledge management encompasses various topics such as meta information management, information gathering and dissemination, information semantics, and planned behavior. As such, a selection of related work from each of these topic areas is presented in this section.

The overarching goal for introducing knowledge driven interactions to SOC in MANETs is to establish a sense of predictability in an otherwise chaotic environment. One method of achieving this is to use the notion of perception of the environment to make decisions, as in the Task Control Architecture (TCA) [98] designed for autonomous agents that control robots. Among other things, the TCA uses information about the environment to ensure that the robot is not in any physical danger. The approach presented in this chapter performs a similar kind of knowledge aggregation, though it is used to protect clients of a service from unexpected disconnection.

All knowledge aggregation and dissemination that must be done to support knowledge-driven architectures occurs at a meta-level, in the sense that the knowledge that is traded is independent of the particular applications that are running on the individual hosts. Thus, meta-information management and the structure of meta-information structures become key facets of the system. One system proposed by Costa et al [17] uses a centralized type repository to maintain meta data. However, such a centralized solution creates a potential single point of failure. The knowledge base described in this chapter uses multiple decentralized repositories called *knowledge bases* on each host with the aim of decreasing the probability of failure.

Another issue that relates to knowledge management is information dissemination. Essentially, to support knowledge driven interactions, each host must disseminate knowledge about itself so that others may react to it. In [49], the authors propose three schemes for information dissemination that are especially tailored to MANETs which focus on rapid dissemination of information without duplication. Three strategies - *select then eliminate* (STE), *eliminate then select* (ETS), and a hybrid of the two are described as ways to ensure that only the relevant hosts get the information. The approach presented in this chapter is able to use a less complex system due to its use of the tuple space paradigm (described in detail in Section 3.4) which ensures that information is distributed only to all connected hosts, which is exactly the set of

hosts to which the knowledge should be disseminated. The use of the more lightweight system also saves scarce computational resources on mobile devices.

The final issue relates to scheduling, with the added restriction that any such "scheduler" must make its decisions in the presence of information about the environment, i.e., the scheduler should support the parameterization of its behavior. An example of such a scheduler is [41], implemented for the DECAF architecture. Essentially, the DECAF scheduler is able to take in functions that enable it to do planning. Two strategies are suggested. The first is contingency planning, where every possibility is evaluated and the putatively best one is chosen, and if that fails for some reason the next best option is then chosen. The second strategy is to give the scheduler a utility function, which it can use as a metric to choose among options.

3.3 Formal Model

This section outlines the underlying formal model for the knowledge management system described in Section 3.4. The discussion of the formalization is preceded by an example that motivates the scenarios where the work is applicable. From this point on, the terminology used is consistent with the SOC paradigm. The reader is reminded however, that the knowledge management model presented here applies more generally beyond SOC–services in the model could be easily replaced by components in a component-oriented middleware context [100], application fragments in a path-oriented context [63], or workflows in a workflow management system.

3.3.1 Motivating Example

The knowledge management software is targeted towards hosts that primarily function in a MANET environment and must exploit some functionality on one of many possible remote hosts in order to complete a task. The choice of the partner host is crucial since a wrong choice can result in the hosts moving out of range before the interaction is completed. The work presented in this chapter facilitates individual applications in making better choices.

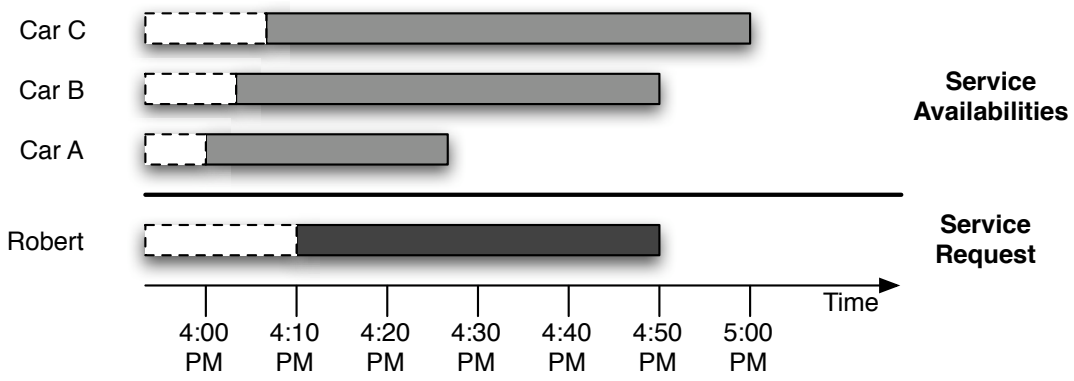

Figure 3.1: Service request and service availability information for the client and three service providers

Consider a scenario involving cars travelling on a highway. If the speed limit on the highway and the destination of the cars is known, the location of cars on the highway can be approximated to a reasonable degree as a function of time. This can be used to build a motion profile for each car and therefore the motion profile of devices that are carried by the occupants of the cars which offer various kinds of services. In one of the cars, Robert is currently reading news items that were downloaded to his PDA, but would like to listen to some music in 10 minutes time, when he will have finished reading the news. Robert already has a pre-compiled playlist of his favorite songs stored on his PDA. However, his PDA must contact a streaming music service to obtain the songs. Robert instructs his PDA to have a music service discovered and ready to use at 4:10 PM, which is 10 minutes from now. The PDA queries PDA's in other cars that are near the car that Robert is in for a streaming music service. When the replies come back, it turns out that there are three cars that can offer the service, as is shown pictorially in Figure 3.1.

Robert has also indicated that he would like to listen to the music for an hour and a half, so the service must be available for that duration. From the figure it is clear that the service offered by Car A will not remain connected for the required duration. Car B will remain connected for the duration, but the knowledge Robert's PDA possesses about the device in Car B indicates that it is farther away from Robert's car than Car C, which could result in signal degradation resulting in a poor quality music stream. Additionally, since the service on Car B becomes unavailable at the exact moment that Robert's requirement ends, there is no room for accommodating errors (perhaps

due to Car B traveling slightly faster than it advertised). Hence, Robert's PDA chooses the service on Car C as the service that it will use. It stores this information until it is time to invoke the service.

The determination of the correct candidate services is done by exploiting knowledge about the other hosts in the network, i.e., their intended *motion profile*, which gives the host's projected location as a function of time. The client collects hosts' motion profiles via a gossipping protocol. It feeds the profiles it collects to an algorithm that helps it choose the *satisfying set* of services. Once the algorithm returns the satisfying set, the services in the set are ranked according to some criteria (currently time to disconnection from the client is the criteria that a host uses to rank services). The client connects to the service that is ranked highest from among those in the satisfying set. It should be noted that in this example, the client had only one service requirement. It is expected that a client application will have more than one requirement during normal operation, in which case there would be multiple satisfying sets, one per requirement. One service would be chosen from each set to satisfy the corresponding requirement.

3.3.2 Knowledge Management Model

In the formal model for the knowledge management system, there are a finite set of hosts (mobile devices), each distinguished by a unique identifier, that are capable of physical motion in a predetermined area. Points outside this area are considered undefined. A host may have one or more agents executing on it. Each agent represents a service, i.e., an instance of a process that offers some functionality, and is capable of logical mobility. Note that agents are used simply as units of modularity. It is also assumed that the clocks on all participating hosts are synchronized using an external solution such as [31]. The following definitions apply across the model:

H - the set of identifiers for hosts in the MANET
S - the set of identifiers for services available in the MANET
L - the set of valid locations which hosts may occupy
T - the time domain
K - the global knowledge in the MANET

While K represents the global knowledge, $K(h)$ is used to denote the knowledge available on host h. More formally, it can be said that:

$$K = \prod_h K(h)$$

where the global knowledge K is defined as the logical union of the various local knowledge bases on hosts in the MANET (the knowledge base on each host contains non-functional information about the local host as well as other hosts in the MANET). Examples of non-functional information are velocity of hosts, remaining battery power, etc. The knowledge composition process $\prod$ is made possible by the assumption that the knowledge held by any one host is always accurate albeit possibly incomplete. For instance, a host h cannot have knowledge about another host i in its local knowledge base K(h) that the host i does not have about itself in its own knowledge base K(i), i.e., a host cannot know more about others than others know about themselves. These and other similar restrictions are not within the scope of this discussion. A more extensive discussion of issues may be found in [88].

In addition to the key elements of the model, individual hosts, services, service requests and the knowledge associated with them are characterized in more detail.

Hosts. Hosts are devices capable of moving in physical space. In this model, a host possessing a unique identifier h is characterized by its motion profile $\mu(h)$ and knowledge base $K(h)$. The motion profile of a host describes that host's mobility pattern, essentially a function that takes in a time value and returns the location of the host at that time. For this work, It is assumed that the motion profile is provided to the system, though such profiles can be easily computed by analyzing a personal schedule, work schedules, etc. Formally:

μ: $H \rightarrow (T \rightarrow L)$
Note that for convenience, $\mu(h)(t)$ is written simply as $\mu(h,t)$

The motion profile function can be used to express various types of mobility. For example:

$\langle \forall t : \mu(h,t) = l \rangle$ where $(l \in L)$ indicates a stationary host

$\mu(h,t) = \mu(h,t_0) + (t - t_0)v_h$ captures motion in a straight line at velocity v_h.

Similarly, more complex motions can be captured with more complex specifications for $\mu(h)$. The motion profiles of hosts are examples of knowledge since they help in understanding how various para-functional (i.e., beyond the primary computation) components of the system are behaving. Motion profiles are exchanged freely among hosts using a gossiping protocol. Whenever two hosts meet, they exchange their own motion profiles as well as the motion profiles they have collected from hosts during previous encounters. This results in an epidemic flooding of motion profile information in the network. A discussion of how these motion profiles are used appears in a later subsection.

Services. Services are software processes that provide functionality that can be used by interested clients. Each service s is characterized by its unique identifier ($s \in S$) and the following attributes:

$\chi(s)$ - the capabilities of the service
$\pi(s)$ - the performance attributes of the service
$\delta(s)$ - the external dependencies of the service
$\alpha(s)$ - the allocation profile of the service

The capabilities $\chi(s)$ describe the functionality of the service, e.g., a "printer" service, while the performance attributes $\pi(s)$ describe how well the functionality can be discharged, e.g. resolution, pages per minute, etc.

The external dependencies of a service, i.e., other services that are required to discharge the advertised functionality, are captured by δ, the set of dependencies. This set of dependencies is computed by analyzing the code of the service:

$\delta : S \rightarrow \wp(S)$ which induces the binary relation
$x \ \overline{\delta} \ y \equiv (y \in \delta(x)) \vee \langle \exists z :: x \overline{\delta} z \wedge z \overline{\delta} y \rangle$

The first formula indicates that δ is a function from the set of service identifiers S to the power set $\wp$ of service identifiers, i.e., the dependency of any service is on

zero, one, or more other services. The second formula captures the closure of the dependency function. It is assumed that any dependency needs to be migrated to the host on which the primary service is executing and be available for the entire duration of the service requirement. Admittedly, such a strategy may be considered wasteful–dependencies could be available only when they are required, i.e., during a small fraction of the service requirement interval overall, and could be used remotely. However, this complicates the presentation, and a formal treatment of this issue is deferred to future work.

Since services can be logically mobile, they can be resident on various hosts over time. This is captured by the service's allocation function $\alpha(s)$, which is specified as follows:

$\alpha : S \rightarrow (T \rightarrow H)$
Again, for convenience, write $\alpha(s)(t)$ as $\alpha(s,t)$

The function $\alpha(s,t)$ returns the host on which the service is resident at a given point in time. If a service is not logically mobile, its allocation function always maps to the host that created the service. In other words, $\alpha(s,t)$ gives the logical location of the service at any given point in time.

Services also possess a motion profile μ, which like the motion profiles of hosts gives the physical location of the service at a given point in time. However, unlike hosts, the motion profile of a service cannot be defined in an arbitrary manner. Rather it must be derived from the allocation profile. The motion profile of the service at a given instant is the same as the motion profile of the host on which it resides at that instant (which is indicated by its allocation profile). If a service moves to another host, it "inherits" the motion profile for that host. I overload the definition of μ to include mobility of services. Formally:

$\mu(s,t) = \mu(\alpha(s,t),t)$

Like hosts, the motion profile of a service along with its allocation profile is considered knowledge that is associated with that particular service and is freely traded in the epidemic manner described earlier.

Service Requests. Service requests are used by clients to specify the kind of services in which they are interested. Traditionally, services were requested according to their capabilities and performance attributes. In the model presented here, due to the exploitation of knowledge, *pre-planning* can be supported, i.e., the client can decide *a priori* the types of services it requires and the *time intervals* during which it requires them. The system then uses all available knowledge to choose the services that are best suited to the client's needs and are actually within communication range of the client during the interval of the requirement. Each request is assumed to have a unique identifier. Thus a service request r is characterized using the attributes defined as follows:

$\beta(r)$ - the host that made the request
$\chi(r)$ - the capabilities desired
$\pi(r)$ - the performance level desired
$st(r)$ - the starting time for the service requirement
$et(r)$ - the ending time for the service requirement

Observe that while the time interval for which the service is required is specified explicitly, the location at which the service is required is not specified. This is because the location value is implicitly specified. For a service request $\langle r, h, c, p, t_1, t_2 \rangle$, the locations at which the service is desired are given by the motion profile of the client host h over the time interval $[t_1, t_2]$. In other words, the locations can be obtained by evaluating the motion profile of the requesting host for all points in the time interval specified explicitly in the request.

Having defined hosts, services and service requests, the next step is to define how services are selected.

3.3.3 Defining the Satisfying Set

Recall that in traditional SOC systems, service selection was done according to capabilities and performance attributes. In addition to matching capabilities and performance attributes, the system must also take into consideration the spatiotemporal

characteristics of the service. This subsection formally describes how services are chosen.

Basic Satisfiability. When looking for a service s which can *satisfy* a service request r, the first thing that needs to be considered is whether that service is actually what the client is looking for. This is called the *basic satisfiability requirement*. For a service requirement r and a service s, a basic satisfiability relation σ is formally defined as:

$$\sigma(r, s) \equiv \chi(r) \leq \chi(s) \wedge \pi(r) \leq \pi(s)$$

The " $\leq''$ operator is overloaded in the formula. In the case of capabilities, the " $\leq''$ operator indicates that the public interface of service s completely covers the set of operations specified in the service requirement r. In the case of the attributes, " $\leq''$ indicates that the service s has all the attributes specified in the service requirement r and for every such attribute, the value of the attribute in the service subsumes that in the requirement. The determination of whether one value is subsumed by another is determined by the attribute type itself which is assumed to have a built-in comparator. Observe that the notion of basic satisfiability is identical to the satisfiability requirements in traditional SOC systems. However, this is where limitations of the current state of the art becomes apparent. A service that has all the capabilities and performance is useless if it is not within communication range of the client at the time of the requirement. Hence, the spatiotemporal characteristics of the service must be taken into consideration, in addition to its capabilities.

Reachability. In addition to meeting the basic satisfiablity requirements, a service must be *reachable* for the entire duration of the service requirement. A service is considered reachable if its allocation profile $\alpha(s)$ evaluates to some host h that is within communication range of the client host h_c for the time interval of the service requirement $[t_1, t_2]$. Formally, reachability ρ is defined as

$$\rho(h_c, s, t_1, t_2) \equiv \langle \forall t : t_1 \leq t \leq t_2 :: |\mu(h_c, t) - \mu(\alpha(s, t), t)| \leq \Delta \rangle$$
where Δ is the communication range.

For convenience, the definition of ρ is overloaded to encompass the reachability between two hosts. It is simply defined as:

$$\rho(h_c, h, t_1, t_2) \equiv \langle \forall t : t_1 \leq t \leq t_2 :: |\mu(h_c, t) - \mu(h, t)| \leq \Delta \rangle$$

Observe that the definition of reachability ρ depends on having access to the motion profiles of hosts and the allocation profiles of services in question. The reader is reminded that these profiles constitute knowledge and are disseminated freely among hosts in the MANET via a gossipping protocol. This knowledge is stored in a local knowledge base on each host. The motion profiles stored in the knowledge base are used to determine whether a service will be allocated to a host within range at the time of the request, thereby meeting the reachability requirement.

Under the assumption that services do not exhibit logical mobility, a service s is said to satisfy a service requirement r made by a client on host h if the following conditions are met:

1) $\sigma(r, s)$ - basic satisfiability
2) $\rho(h, s, t_1, t_2)$ - reachability
3) $\langle \forall d : d \in \delta(s) :: \rho(\alpha(s, t), d, t_1, t_2) \rangle$ - dependencies
   all w.r.t. $K(h)$

Condition 1 above states that the service should meet the basic satisfiability requirements, i.e., capabilities and attributes, and that the service should be reachable for the entire duration of the requirement. Further, all dependencies of the service should also be reachable from the host on which the primary service is executing. Note that $K(h)$ is used to denote the knowledge base on the host identified by h. Since the formulas depend on motion and allocation profiles, they have to be evaluated with respect to some knowledge base. Since planning normally takes place at the point of origin of the request, i.e., host h, the satisfiability is relative to its knowledge base, i.e., $K(h)$.

Logical Mobility. Thus far in this presentation, an assumption was made that services were not logically mobile. Here, this assumption is removed and services are allowed to migrate from host to host. The logical mobility of services adds a degree of freedom. Services can now be migrated from a host that is not in range of the client to one that is in range. Consider a scenario where an idle service can be moved from the host on which it is currently resident (which ostensibly would

not be in communication range at the time of the requirement) to a host that will be in communication range at the appropriate time. For this, a function called Γ is introduced and defined as follows:

$$\Gamma(h_1, h_2, t) = \langle \exists t_s, t_e : t_s < t_e \leq t :: \rho(h_1, h_2, t_s, t_e) \rangle$$

The predicate Γ indicates whether there is a time interval before a deadline time t, during which hosts h_1 and h_2 are in communication range (this interval can then be used to logically move the service between the two hosts). Note that for this case, once the service has moved to this host, the service is resident on the host at least until the end time of the client's request. Thus, under these assumptions, a service s satisfies a requirement r made by host h by relying on host h_t if the following conditions are satisfied:

1) $\sigma(r, s)$ - basic satisfiability
2) $\rho(h, h_t, st(r), et(r))$ - reachability of some host h_t
3) $\texttt{migrate}(\alpha(s, t), h_t, et(r))$ - migrate service to host h_t
4) $\langle \forall d \in \delta(s) \exists h_d :: \rho(h_d, h_t, st(r), et(r)) \wedge \texttt{migrate}(\alpha(d, t), h_t, st(r)) \rangle$ -
   reachability to some host h_d and migrate dependencies to host h_d
   w.r.t. $K(h)$

where

$$\texttt{migrate}(h_1, h_2, t) = \Gamma(h_1, h_2, t) \vee \langle \exists h, t' : h \in H \wedge t' < t ::$$
$$\texttt{migrate}(h_1, h, t') \wedge \texttt{migrate}(h, h_2, t) \rangle$$

Even in the presence of logical mobility, the service must still meet the basic satisfiablity requirement. However, the reachability requirement is not a strict one. In combination with the `migrate` operation, the reachability requirement can be stated as: there should be a suitable host h_t that is reachable, and the service should be able to migrate to this host before the start time $st(r)$ of the request and remain there for the duration of the request.

To summarize, a service that is otherwise suitable in terms of capabilities and performance but is not resident on a reachable host is being moved to a host that is

reachable by the client before the client actually needs the service (proactive logical movement). Naturally, any dependencies the service requires must also be moved in a manner similar to the service itself.

Thus far, the conditions that result in a service satisfying a request have been formally defined. It should be noted that the consideration of the spatiotemporal aspects of a service and the proactive selection of services is only possible due to the knowledge that is gathered by each host.

3.4 Software Architecture & Implementation

This section covers the software architecture and implementation details of the knowledge management system that was formalized in Section 3.3. First, a brief overview of SPAWN is provided. SPAWN is the SOC middleware for MANETs within which the knowledge management system has been implemented. This is followed by a presentation of the basics of the knowledge management system and its role within the context of the SPAWN system. A more detailed discussion of each of the components of the knowledge management system appears next before the section is concluded with code examples that show how an end user might access the system to request services in a knowledge managed SPAWN architecture.

3.4.1 SPAWN Overview

SPAWN is a proxy-based SOC middleware for MANETs. It is based on the Jini [110] model and is written entirely in Java. To adapt the Jini model to MANETs, several changes were required. The centralized service directory of Jini was replaced with local service directories on each host. When a host needs to advertise a service, it places an advertisement (and the appropriate proxy) in its local service directory. Groups of hosts that form a clique (i.e., all hosts in the group are within communication range of each other) logically merge their local service directories to form a transiently shared service directory that is federated across all members of the clique.

In this way, the advertisements in the local service directories become accessible to other hosts in the MANET.

When a host moves away, it un-shares its local service directory. Hence, the service advertisements belonging to that host are no longer available in the federated directory, which is consistent with the fact that the services offered by that host are also not available (due to the host not being in communication range anymore). This eliminates the need for the Jini leasing mechanism, which was not used in SPAWN. The Java RMI mechanism for invoking services was replaced with a tuple-space-based communication mechanism (described later) due to it being more resilient against disruptions. Finally, the centralized code repository for proxy code was replaced with a transiently shared, federated repository, much like the service directory itself.

In addition to the modifications made to the Jini approach, two new features were added. The first is an automated upgrade system where the services and their proxies can be upgraded while they are running with very little interruption in communication. The second is a mobile thread system for Java where a service can be migrated from host to host to stay in range of the client. More details on these enhancements can be found in [39].

The dynamism of MANETs also precluded the direct use of traditional socket streams as communication channels between service providers. Socket streams are associated with two stable endpoints. When the connection between the endpoints breaks down, the socket closes throwing an exception. To circumvent this, SPAWN uses a tuple space abstraction which catches such exceptions and automatically tries to reconnect with other co-located neighbors, thereby abstracting frequent disconnections from the programmer. The tuple space also allows a generative style of content-based communication as described in [28]. The service directory and code repository are implemented in terms of tuple spaces which are automatically federated among hosts in proximity. Tuple spaces are containers for tuples, which are ordered sequences of Java objects that have a type and a value. An agent places a tuple in the tuple space using the `out(tuple)` operation, making it available to all other agents that are sharing the same tuple space. To read a tuple from the tuple space using the `in(template)` operation, an agent needs to provide a template, which is a pattern describing the tuple that the agent is interested in. A template is a sequence of fields,

each of which can contain a formal (wildcard) representing the required type for that field or an actual value that identifies the type and value of the corresponding field. A template is said to match a tuple if all the corresponding fields match pairwise. Service advertisements are implemented as tuples that contain a description of the service's capabilities while service requests are implemented as templates.

The `ServiceDirectory` class is a wrapper class that owns a generic tuple space. Together, this class and tuple space comprise a service directory. The `ServiceDirectory` class provides standard SOC operations such as advertise, request, and invoke. A service is advertised by placing a tuple in the tuple space owned by the `ServiceDirectory` class using the `out` operation. A request is implemented as a `rd` operation, with the interface of the desired service being passed as the template. Invocation is done through a targeted remote `out` operation where the tuple is stamped to indicate its destination host.

To provide asynchronous interactions, SPAWN offers a reaction mechanism. An agent can declare interest in a tuple by registering a reaction on a tuple space using a remote operation parametrized by an appropriate template and by providing a callback function to be called when a matching tuple becomes available. All reactions in SPAWN are *weak* reactions, meaning that once the condition for the reaction becomes true, the callback function is guaranteed to be called eventually, but not necessarily within a single atomic step.

3.4.2 The Knowledge Management System

The Knowledge Management System described in this chapter has been implemented as an extension to the SPAWN system. The knowledge management system fits between the API layer and the communication layer of SPAWN. This required the extensions to SPAWN shown in Figure 3.2.

The Knowledge Management System is responsible for handling the exchange of knowledge among hosts in the MANET and managing the knowledge base on each host so that the information it contains may be obtained easily by interested applications. More precisely, the Knowledge Management System performs the following

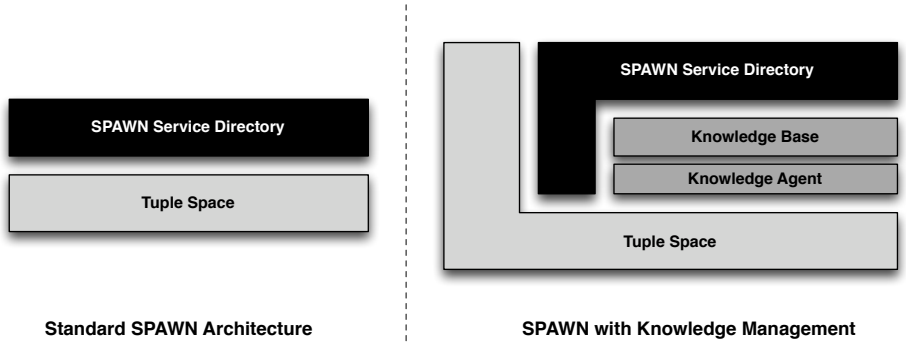

Figure 3.2: SPAWN architecture with and without knowledge management

functions: (1) aggregation of knowledge about other hosts in the MANET, (2) dissemination of knowledge about the local host to other hosts, and (3) management of the local knowledge base.

The knowledge management system is represented by a singleton `KnowledgeManager` that runs on each host in the MANET. Like the rest of SPAWN, the `KnowledgeManager` has been implemented in Java. On startup, the knowledge manager starts a SPAWN agent which is called the *knowledge agent*. This agent is the interface between the knowledge manager and the tuple space that is owned by the `ServiceDirectory` class (the agent is necessary due to a design feature of SPAWN which allows only SPAWN agents to access tuple spaces).

The SPAWN `ServiceDirectory` class now owns a `KnowledgeManager`, which encapsulates all knowledge management functions, in addition to a tuple space. All service advertisements and requests are now directed to the `KnowledgeManager` instead of being placed directly in the tuple space (service invocations and other communication are still placed directly in the tuple space). The `KnowledgeManager` has access to the tuple space owned by the `ServiceDirectory` (through the `KnowledgeAgent`) for communication related to the exchange of knowledge about services. To discover services, the `ServiceDirectory` now calls the `findService(...)` method on the `KnowledgeManager` which returns an appropriate service if one is available. Note that the `KnowledgeManager` has access to the tuple space from which it retrieves

all service advertisements, but it only makes services that meet spatiotemporal requirements available to the `ServiceDirectory` class. Thus, the `ServiceDirectory` contains only those services with an acceptable level of connectivity.

The `KnowledgeManager` has three sub-components: (1) a `KnowledgeDisseminator` that distributes knowledge about its parent host and services running on it to other hosts in the MANET, (2) a `KnowledgeAggregator` that gathers knowledge about other hosts and services in the MANET, and (3) a `KnowledgeBase` that stores this gathered knowledge. The complete structure of the knowledge management system is shown in Figure 3.3. Before the details of these three components are presented, the representation of the knowledge that is used for these components is described in detail.

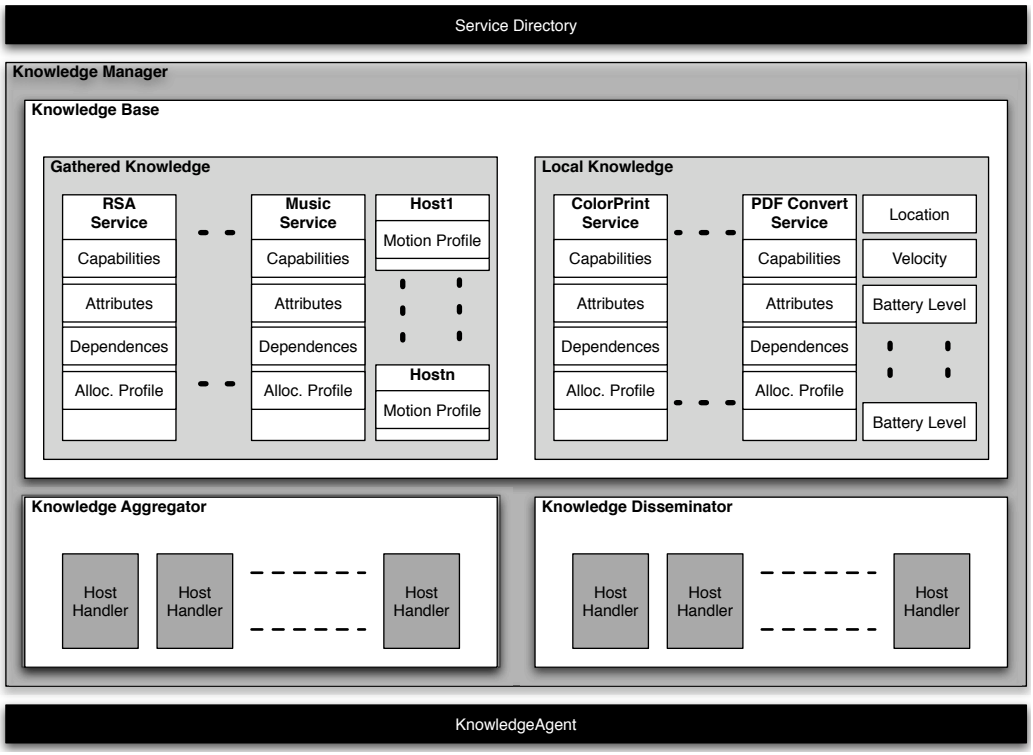

Figure 3.3: The structure of the Knowledge Management System

3.4.3 Knowledge Representation

The question of how knowledge is represented and codified is crucial as it determines how knowledge can be organized and the flexibility with which it can be exchanged. In the system, each piece of knowledge (excluding identifiers) is referred to as a *parameter*. For example, a host h by this definition has one parameter, its motion profile μ. A service s has multiple parameters, such as its capabilities χ, performance attributes π, and so on. Every parameter is associated with an identifier which identifies the host or service with which the parameter is associated. Parameters are also stamped with the time of their last update. While the system can support an arbitrary number of parameters, the following discussion restricts itself to those described in Section 3.3.

Since the parameters represent knowledge and are intended to be communicated from host to host, these parameters are encapsulated within tuples, just like all other communications in the system. A special class of tuples is defined called "knowledge tuples" that carry only knowledge. Knowledge tuples are distinguished from other tuples by virtue of the fact that its first field contains the reserved string "Knowledge". The general form of the knowledge tuple is:

<"Knowledge", String:owner , ParamType:type, Value:value>

Thus, for example, a knowledge tuple containing the allocation profile of some service "printService" would look like:

<"Knowledge", "printService" , AllocationProfile.class, profileValue>
where profileValue is an object of type AllocationProfile

The Knowledge Base

The Knowledge Base on each host is divided into two sections: gathered knowledge and local knowledge. Gathered knowledge is knowledge associated with other hosts and services in the MANET. Local knowledge is knowledge associated with the local host. The local knowledge section is organized as a `Vector` of *entries*. Entries can

be only of type *service entry* since the only host being tracked in this section of the knowledge base is the local host whose parameters are stored separately. Each entry has within it a vector of parameters and their values, e.g., the host entry will have a parameter called "motion profile" with a value that is a function definition. The local knowledge portion offers the following basic API for access and updates:

```
Object:value getLocalParamValue(Entry:entry, String:paramName)
setLocalParamValue(Entry:entry, String:paramName, Object:newValue)
```

The gathered knowledge portion of the knowledge base is different from the local knowledge portion in that more than one host entry may be present in the gathered knowledge section, each host entry representing a host in the MANET of which the local host is aware. The API of the gathered knowledge portion is:

```
Object:value getGatheredParamValue(Entry:entry, String:paramName)
setGatheredParamValue(Entry:entry, String:paramName, Object:newValue)
```

The second method is only accessible to the Knowledge Aggregator (described later) to ensure that local applications do not tamper with the gathered knowledge. Both parts of the knowledge base share a common API to create new entries and parameters:

```
createHostEntry(String:hostName)
createServiceEntry(String:serviceName)
createHostParam(Entry:hostEntry, String:paramName, Object:paramValue)
createServiceParam(Entry:serviceEntry, String:paramName, Object:value)
```

Entries and parameters in the knowledge base are created on an on-demand basis. A host entry for some host h is created only when some knowledge pertaining to that host is received for the first time. Similarly, parameters are created as they are needed. This allows for hosts to register interest in only some parameters of a host or service, which is not essential in the limited scope of the presentation in this section, but can be useful if a large amount of knowledge is traded and hence is an accommodation for extensibility.

Finally, the knowledge base supports several convenience methods:

```
Service:service findService(ServiceRequirement:requirement)
Vector:services getAllKnownServices()
Vector:motionProfiles getMPOfServicesWith(Capabilities:cap,
    Attributes:attrib)
MotionProfile:mProfile getLocalMP()
```

The Knowledge Aggregator

The knowledge aggregator is responsible for aggregating knowledge from other hosts in the MANET. Initially, the knowledge base on a host is empty save for the knowledge about the local host. However, each host has a knowledge disseminator (described next) that is responsible for distributing the contents of the knowledge base to other hosts. Thus when a host encounters another host, the knowledge disseminators on each host pairs up with the knowledge aggregator on the other host and exchange the information in their knowledge base. Simply put, as hosts encounter more and more hosts, their knowledge base continues to grow. Given the knowledge manager design, it is also possible for hosts to exchange more restricted sets of parameters about certain hosts rather than the complete contents of their knowledge bases. However, this is outside the scope of this dissertation. It is assumed here that all knowledge is freely exchanged.

The complete process from the creation of the knowledge manager to the aggregation of knowledge is shown in Figure 3.4. Numbers in parentheses in the following text refer to steps shown in the figure. To aggregate knowledge, the knowledge manager registers reactions on the tuple space (7) (via the knowledge agent (6) since no other class can perform tuple space operations). The reactions can be registered for each type of knowledge that the knowledge manager wants to aggregate or for all types of knowledge. For example, to aggregate any type of knowledge, the knowledge manager registers a reaction with a template of the form <"Knowledge", String.class, String.class, Object.class,>. The first field indicates interest in knowledge tuples while the second field is a wildcard indicating interest in any host that provides knowledge. The third field is a wildcard indicating that all types of knowledge are of interest,

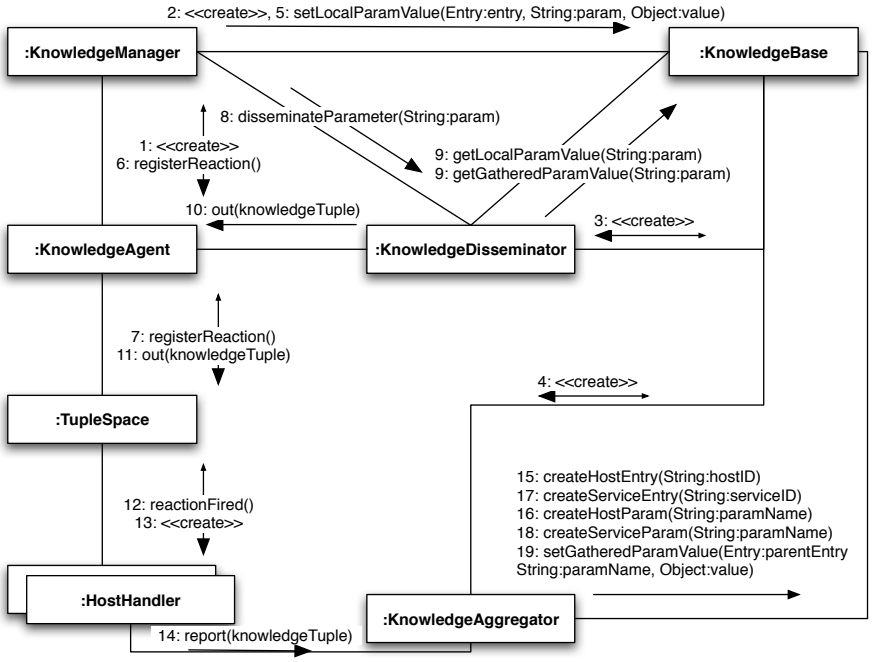

Figure 3.4: Interaction of the various components of the knowledge management system

while the fourth is a class that is a wildcard for the actual value of the knowledge parameter. Wildcards are used since any knowledge associated with any host in the MANET is desirable.

When a host comes within communication range of the reference host, SPAWN raises an event indicating that a new host has been detected (12). This event causes the reaction registered (for knowledge) to fire. When the reaction fires, indicating that new knowledge of the type for which interest was previously registered has been placed in the tuple space, a `HostHandler` is created to retrieve a copy of the knowledge tuple from the tuple space (13). Note that a *copy* of the knowledge tuple is retrieved so that other hosts in the network may also read the same knowledge. It is the responsibility of the host that disseminates the knowledge to remove any old tuples using the `in` operation before outing a tuple with updated knowledge. Once the tuple has been retrieved (14), the second field is examined. If the identifier in that field is not present in the knowledge base, then a new entry is created in the knowledge base using `createHostEntry(...)` (15) or `createServiceEntry(...)` (17) as appropriate. If the entry exists, the next thing that is checked is whether the parameter type

(indicated by the third field) is present under the host or service entry. If not, the parameter type is created using `createHostParam(...)` (16) or `createServiceParam` (18) as appropriate. If the parameter type already exists, the timestamp of the value in the fourth field of the tuple is checked against the value in the knowledge base. If the newly retrieved value is more recent, the knowledge base is updated using the `setGatheredParamValue(...)` method (19).

One concern is that the knowledge base may grow forever. This concern is addressed by examining the motion profiles of the hosts and services and maintaining the entry up to the latest point in time for which the motion profile has information. After this time, the motion profile is deleted, as it is no longer of any use.

The Knowledge Disseminator

The knowledge disseminator is responsible for disseminating knowledge that is in a host's local knowledge base. This knowledge can be knowledge about the local host or knowledge about other hosts that has been aggregated by the local host in the past. The `RequestHandler` within the disseminator listens for requests from aggregators on other hosts. When such a request is received (8), it obtains the appropriate contents from the knowledge base and packages it. Each parameter in the knowledge base is packaged into a knowledge tuple of the form

<"Knowledge", String:owner , ParamType:type, Value:value>

Access to the knowledge is possible via the `getLocalParamValue` method (9) for local knowledge and `getGatheredParamValue` method (9) for aggregated knowledge as described earlier. Like the aggregator, the disseminator supports the dissemination of a subset of all the information it has, if so desired. This is useful when two hosts have a similar amount of knowledge in their knowledge bases and need send only small updates rather than the complete contents of their knowledge base. Once the knowledge tuples are ready, they are passed to the `Transmitter` which instructs the knowledge agent to place the knowledge tuples into the federated tuple space (10).

3.4.4 Anatomy of a Service Request

Having described the structure and implementation of the Knowledge Manager, this section is concluded with a description of the process that occures when a client wants to discover a service. The client makes a request to the `ServiceDirectory` class, which has been updated to take parameters for the time duration for which the service is required. Thus, for a client to request a service, it needs only three lines of code:

```
//Get handle to local directory singleton
ServiceDirectory dir = ServiceDirectory.getLocalDirectory();
RequestID rid = dir.requestService(new ServiceRequest(
    capabilities, attrib, startTime, endTime));
//At time startTime:
ServiceProxy proxy = dir.getProxy(rid);
```

The client can make the request at any time and gets a request ID which identifies that particular request. A qualifying service is found, and its proxy stored locally. At the time the client actually needs the service, it queries the `ServiceDirectory` with the request ID and obtains the proxy which it can then use as needed.

While the client API is simple, complex operations are going on under the API layer. Since this chapter addresses concerns at the middleware level, the code that executes within the `ServiceDirectory` when a service request is received at the API layer is now presented. The `ServiceDirectory` calls `findService(req)` on the `KnowledgeBase` class. That method executes the following code.

```
ServiceProxy proxy = null;
boolean[] qualifies = true;
Pair[] mProfiles = KnowledgeBase.getMPOfServicesWith(
    req.getCapabilities(), req.getAttributes());
for(int i = 0, i < mProfiles.length), i++){
    for(int j = req.getStartTime(), j < req.getEndTime(), j++){
```

```
            if(!(inRange(mProfiles[i].getMP().evaluate(j),
                KnowledgeBase.getLocalMP().evaluate(j))) {
                qualifies[i] = false;
            }
        }
    }
    for(int i = 0; i < qualifies.length; i++){
        if(qualifies[i]){
            proxy = KnowledgeBase.getGatheredParamValue(
                mProfiles[i].getServiceEntry(), "Proxy")
            return proxy;
        }
    }
    return null;
```

The second line of code gets the motion profiles of all services that meet the capabilities and attributes requirements (but which may not meet the spatiotemporal requirements). Details of how this method works is shown below.

```
Vector qualifyingServices = new Vector();
ServiceEntry[] services = KnowledgeBase.getAllKnownServices();
for(int i = 0; i < services.size(); i++){
    if(KnowledgeBase.getGatheredParameter(services[i],
        ‘‘Capabilities’’).matches(req.getCapabilities()) &&
         KnowledgeBase.getGatheredParameter(services[i],
        ‘‘Attributes’’).matches(req.getAttributes())) {
        qualifyingServices.add(KnowledgeBase.getGatheredParameter(
            services[i], ‘‘MotionProfile’’));
    }
}
```

The matches method for capabilities returns true if the interface offered by the service proxy is a subclass of the interface specified in the service requirement. For

attributes, it checks to see that the service proxy has all the attributes specified in the requirements. If this condition is satisfied, it checks each attribute individually. Every attribute in the implementation is actually a three-tuple [attribute, value, comparator]. The comparator indicates whether a greater value or a lesser value of the attribute is desired. If the attributes offered by the service compared using the comparator are greater than those in the requirement, the `matches(...)` method returns `true`.

Once the motion profiles are obtained, the motion profiles of the local host and the service are checked to see if they evaluate to locations that are in communication range of each other. This procedure is repeated for the duration of the service request. The first qualifying service that meets the spatiotemporal test is returned to the `ServiceDirectory`.

On the back end, the knowledge manager watches the SPAWN tuple space for knowledge tuples using the reaction method described previously. When such tuples are located, the information in them is added to the `KnowledgeBase` using the `createHostEntry(...)`, `createServiceEntry(...)`, `createHostParam(...)`, `createServiceParam(...)`, and `setGatheredParamValue(...)` methods.

3.5 Results

Having described the formal model, architecture, and implementation details of the knowledge management system in previous sections, this section presents results of simulation experiments conducted to evaluate the approach. The goal of these experiments is to show that the use of knowledge resulted in a better choice of service being made, without which a task would have failed to complete. In the simulations, the focus is on cases where hosts are mobile but services do not move between hosts, as this case is perhaps the most practical given considerations of security, copyright, and licensing. First, the experimental setup is described followed by results of several experiments, which demonstrate the validity of the approach.

3.5.1 Experimental Setup

To simulate the environment of a knowledge-driven MANET, a custom simulator was developed in which hosts move within a well defined space according to predetermined motion profiles and can offer or request services. The details of the experimental setup are now presented:

Simulation Setup. The simulation space is a grid of 250 x 250 squares. Each square is of size $5m^2$, resulting in an area of 1.25km x 1.25 km, an appropriately large area in which hosts can move freely while having communication occur only occasionally. A host occupies a single square, though a single square may accommodate more than one host. The number of hosts used in the experiments ranges from 25 to 250. Each host possesses a knowledge base, motion profile, service requirements profile and a service offerings profile. Details on how these were generated are given below. All simulations were run for 500 time points. At each time point, all hosts moved to the next location as given by their motion profiles. They then exchanged knowledge with neighbors within their communication radius (which ranged between 2 and 20 grid squares). Once the knowledge exchange phase was completed, the hosts tried to satisfy their requirements. Finally, it should be noted that each data point shown in the experiments represents an average value over 20 different data sets collected in repeated runs.

Mobility Model. Experiments were run using two mobility models: (1) Random Walk and (2) Random Waypoint. In random walk, a starting point was generated at random. For each subsequent entry in the motion profile, one of four directions (UP, DOWN, LEFT, RIGHT) was randomly selected with equal probability and the host moved to that grid square. In random waypoint, a random start point and a random waypoint was selected. The host was then moved with constant velocity from the start point to the first waypoint. Once the waypoint was reached, another waypoint was randomly generated and the process was repeated. In both models, hosts could move only one grid square in a single iteration. All motion profiles were generated prior to the actual experiment as hosts were required to know their motion plan *a priori* (to be able to give it out as knowledge). As may be expected, the random walk profiles tended to keep the host within a smaller region of the simulation space while random waypoint tended to provide better coverage. A variable called `MAX_FUT` was

also defined, which restricted the amount of time into the future for which a motion profile was valid.

Service Model. Each host in the simulation had a service offerings profile and a service requirements profile. The service offerings profile for a host was generated by selecting a random number of services from the master set of six services (`printer, pdfconvert, information, mp3convert, rsaencrypt, imagecrop`). The service requirements profile was generated by selecting a random number of requirements for services that were not in the service offerings profile of the host. This resulted in (1) the host not being able to satisfy requests locally, thereby biasing the statistics and (2) the number of requirements being inversely proportional to a hosts capabilities. All requirements in the system had a fixed length as defined by the global variable `REQ_LEN`. A host was considered *satisfied* if all its requirements were met.

Some Comments. The experimental parameters chosen represent all hosts within a area of reasonable size. If the MANET is formed among a group of people working in a remote area, 1.25km x 1.25km represents a fairly large area. If the MANET is formed between cars on a highway, the area represents a sufficient distance ahead as well as behind the car as well as to the side. The choice of an area of a different size does not affect the results as long as the density of hosts is maintained. Similarly, if the density of hosts is decreased for the area chosen, the performance is likewise affected. Finally, the mobility of the hosts affects the outcomes, which is why the experiments use two very different mobility models - the more random walk, and the slightly more predictable waypoint model.

3.5.2 Experiment I: Varying the Communication Radius

The communication radius of a mobile device determines the extent of its reach and affects the number of hosts with which it can exchange knowledge. This experiment shows how the communication radius of hosts affects (1) the percentage of hosts that have their requirements satisfied and (2) the average size of the knowledge base on each host as a percentage of the size of the global knowledge base. These results are shown in Figure 3.5.

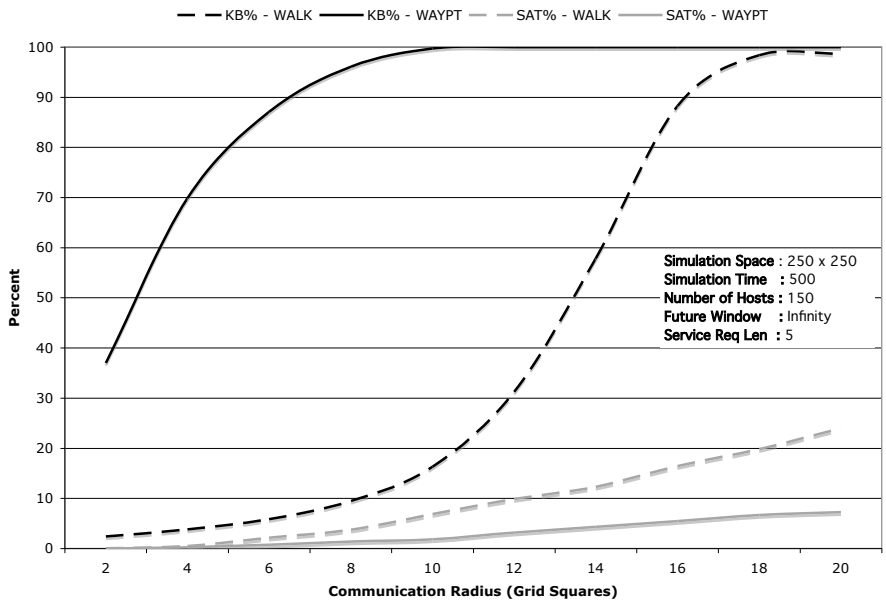

Figure 3.5: Size of knowledge base and percentage of hosts satisfied as a function of communication range

As may be expected, there is an increase in both the percentage of hosts that have their service requirements satisfied, as well as the average size of the knowledge base, with an increase in communication range. The expanded communication range fosters more interactions thereby expanding the knowledge base which in turn increases the chances of finding suitable services. One may question the fact that even with a high level of knowledge, a relatively low percentage of hosts requirements were satisfied. There are several reasons for this low number. (1) Only the hosts that have *all* their service requirements fulfilled are counted as having been satisfied. Thus, there were a number of hosts that had most of their requirements satisfied but were not included in the count because they were not completely satisfied. (2) In some cases, the host acquired knowledge of a required service after the time of requirement had passed. Thus, even though the knowledge was acquired, it did not help identify a service. (3) In many cases, even though the knowledge about a host was acquired, it was not suitably located to meet the timing constraints of the service requirements, i.e., the fraction of *exploitable* knowledge was low. It is exploitable knowledge that explains the difference in numbers between the random walk (WALK) and random waypoint (WAYPT) models. Since WAYPT has better coverage of the entire space, a host meets a lot of other hosts and gathers a lot of knowledge but those hosts are seldom

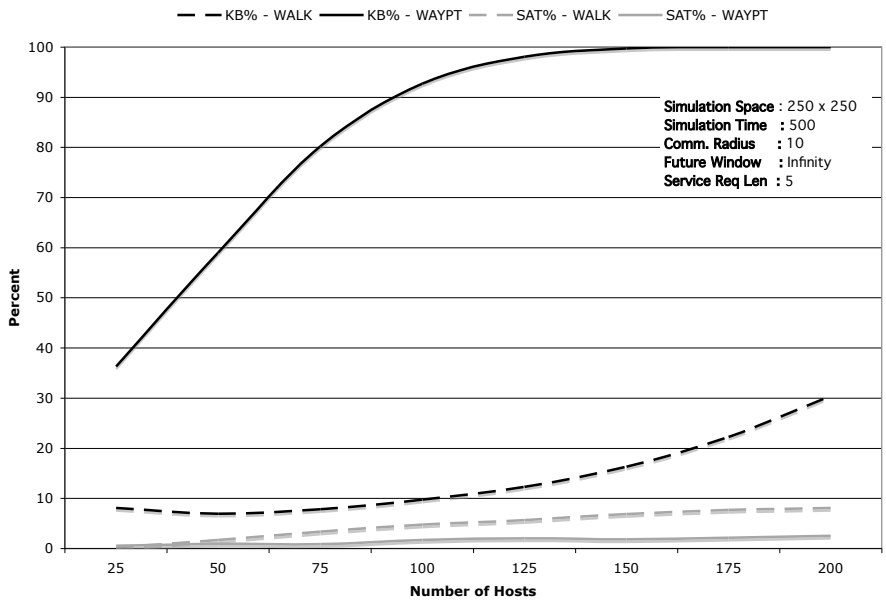

Figure 3.6: Size of knowledge base and percentage of hosts satisfied as a function of host density

on hand when a service is required. In WALK, the host stays within a small region and meets fewer hosts and thereby has a smaller knowledge base, albeit one containing a higher fraction of exploitable knowledge, i.e., the hosts in the knowledge base are those nearby and which remain in close proximity due to their limited mobility. Thus, in WALK, there is a greater correlation between knowledge base size and exploitable knowledge, i.e., knowledge about hosts that can satisfy requirements.

3.5.3 Experiment II: Varying the Number of Hosts.

As mentioned for the previous experiment, the number of interactions a host has with other hosts affects the percentage of hosts that are satisfied and the average size of the knowledge base on each host. This experiment examines how host density affects these parameters, the rationale being that a greater host density will, for a fixed communication range, increase the potential number of interactions. The results are shown in Figure 3.6.

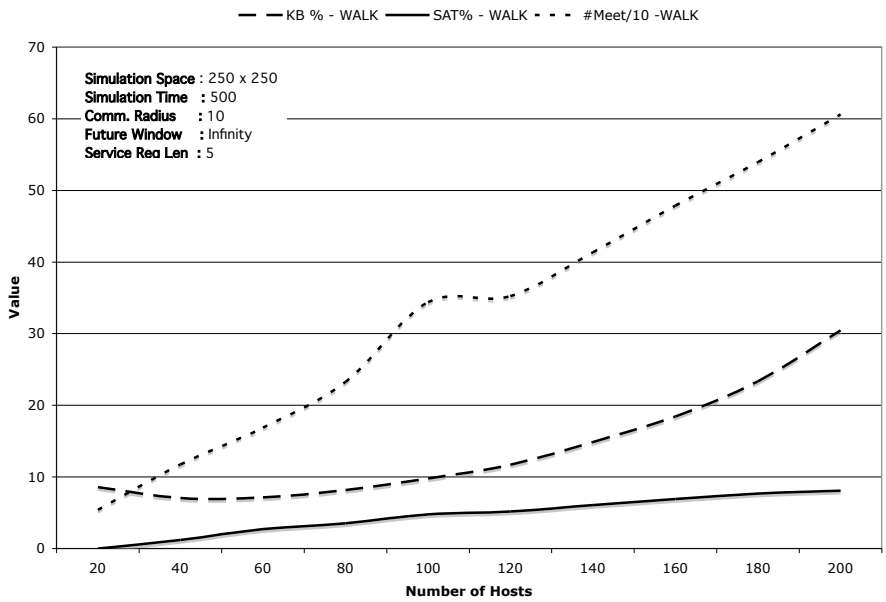

Figure 3.7: Number of meetings, knowledge base size, and satisfied hosts (random walk model)

Increasing the host density increases the percentage of hosts satisfied, though the rate of increase is not as marked as when the communication range is increased. In fact, for WALK, the average knowledge base size actually decreased initially. The explanation for this is simple. Adding hosts is not effective unless these additional hosts interact with other hosts frequently. In the case where the average knowledge base size decreased, the additional hosts were isolated from the rest, and hence their knowledge bases were empty resulting in the average being brought down. Note that this problem was not seen with the WAYPT model which tends to foster more interactions and seldom isolates hosts in a specific region.

Manipulating the communication range and the number of hosts serves to increase or decrease the number of meetings between hosts. The number of meetings is crucial since a higher number of meetings theoretically translates to more opportunities for knowledge exchange and service usage. In the experiments, it was found that an increase in the number of meetings improves the percentage of hosts satisfied for both mobility models. However, the size of the knowledge bases were much smaller in WALK for a similar number of meetings. This was due to the localized nature of

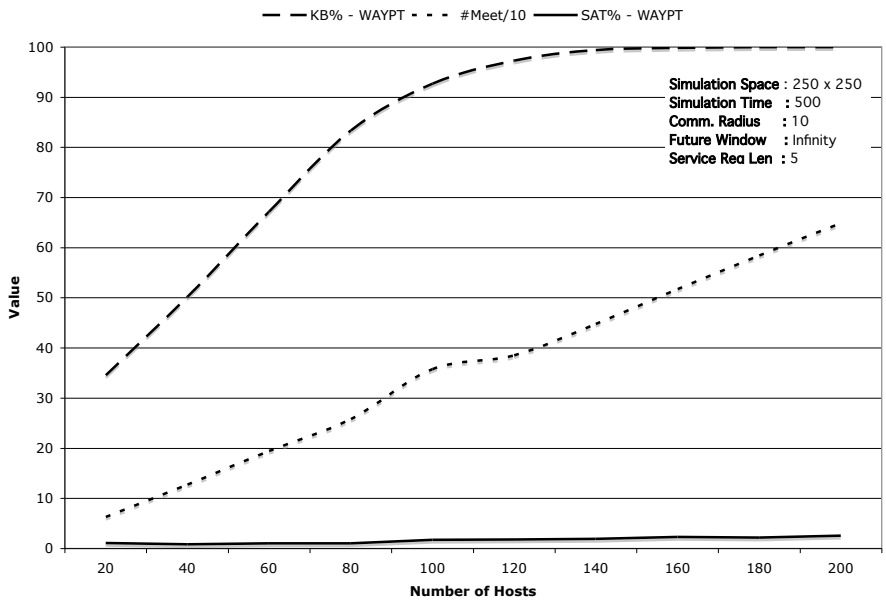

Figure 3.8: Number of meetings, knowledge base size, and satisfied hosts (random waypoint model)

WALK which caused multiple meetings but seldom with unique hosts. The results are shown in Figures 3.7 and 3.8.

3.5.4 Experiment III: Varying the Length of Requirements

Another factor that affects the percentage of hosts that are satisfied is the length of the service requirement interval, with longer duration requirements having a lower chance of being satisfied because they require the client and the service provider to be within range and to have very similar mobility patterns over the requirement interval. The experiments (Figure 3.9) confirmed this expectation. However, it is interesting to note that even when the request length was longer than the maximum time a host could be in communication range, a small percentage of requirements were still being satisfied, which would be impossible without forward looking knowledge.

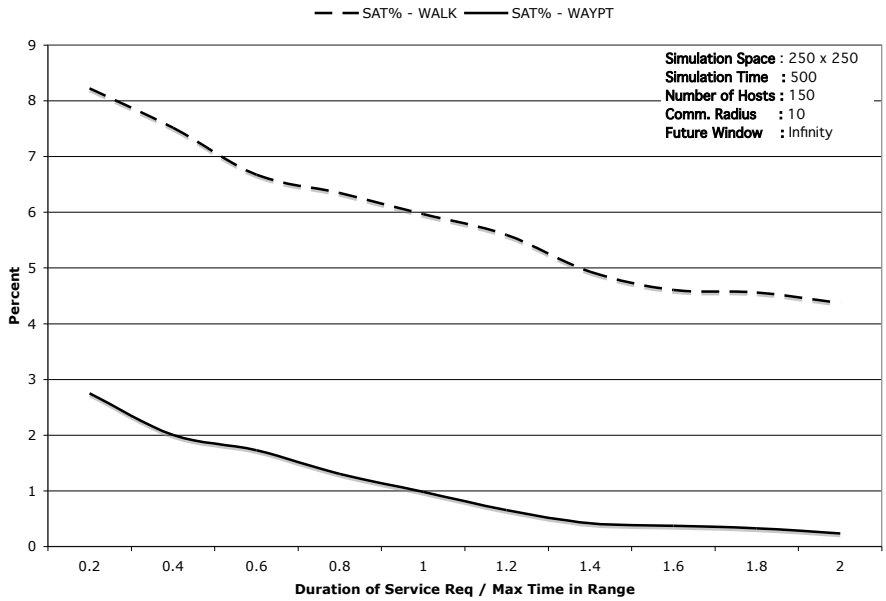

Figure 3.9: Percentage of hosts being satisfied as a function of requirement length

3.5.5 Experiment IV: Varying the Forward Looking Window Size

The final experiment examined the relationship between the amount of time into the future for which a valid motion profile is available and the percentage of hosts that are satisfied. If a larger forward looking window is available, then the chances of finding a suitable service are greater than if the window is smaller. However, in the experiments (see Figure 3.10) it was found that even with a window size that is 10% of the optimum window size, approximately 50% of the hosts that would be satisfied with the unlimited window still end up being satisfied. This is an important result since it means that hosts need to predict their motion for only a short time into the future, which is useful if the host "changes its mind" frequently or does not know its long term mobility pattern.

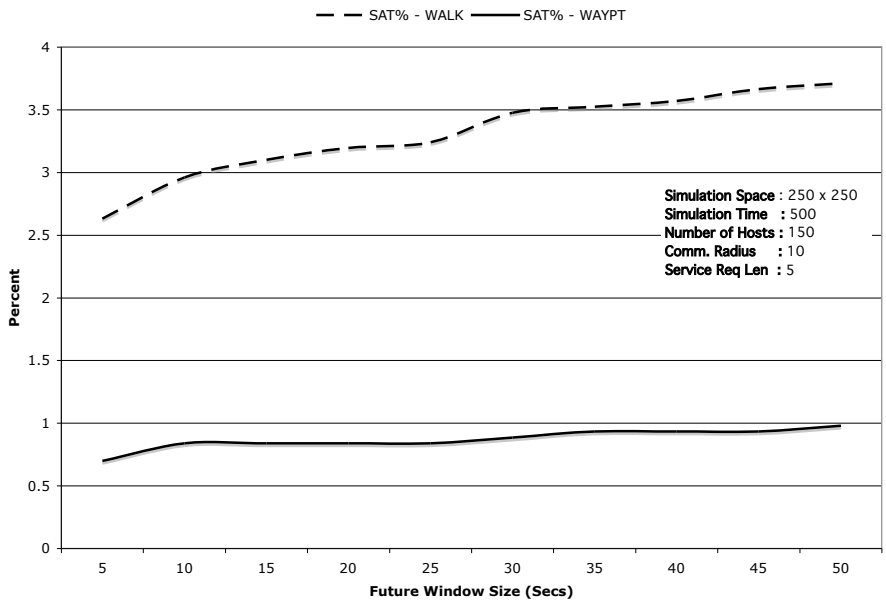

Figure 3.10: Percentage of hosts being satisfied as a function of future window size

3.5.6 Some Remarks

It should be noted that all the experiments shown above were conducted in a knowledge managed environment. When the knowledge management feature was turned off, the percentage of satisfied hosts fell to zero. Admittedly, it will not always be the case that the satisfied hosts percentage will be zero as a non-knowledge-managed system depends purely on chance to satisfy requests. The take home point is that with knowledge management, the performance is better.

The results presented were obtained using randomly generated data sets. While the experiments show general trends, it is acknowledged that the system will not be effective in particular scenarios where the requirements are in conflict with mobility patterns, e.g., an isolated host can never hope to satisfy its requirement profile, nor can a host that requests services before it has met another host. The aim of this work is to show that for reasonable patterns of mobility and requests, the knowledge management approach can bring a greater degree of predictability where a reasonable mobility pattern is defined as a mobility pattern in which the host periodically encounters other hosts and does not remain completely isolated for extended periods of time.

Given the results, one might question whether the overhead of knowledge management is worth the relatively small gains in satisfaction percentage. Given that the target devices are laptops and PDAs, there are relatively more resources to play with over devices such as cellphones. These devices can also be considered to be longer-lived since they can be recharged when their batteries run out. The SPAWN middleware was tested both with and without the knowledge management. There was no significant drop in performance when the knowledge management was enabled. However, knowledge management does result in the number of messages exchanged in the network to increase significantly due to the gossiping protocol used to exchange knowledge. If the network is formed of devices using the 802.11b standard, then the additional messages do not overload the network since the bandwidth is high enough to support the knowledge data, which is not very large (on the order of a few KB at most). However, where the impact of knowledge management is felt most is in the battery life of the devices as more network traffic translates into the radio being on for longer periods which is power intensive. Ultimately, if the fallout from an application failure due to lost connectivity is high enough, the cost of the knowledge management can be justified.

The final remark relates to those hosts that had part of their requirements satisfied. There is a possibility for hosts that have most of their requirements satisfied to be fully satisfied with the help of logically mobile services which can relocate themselves with the aim of satisfying hosts that are close to being fully satisfied, which is part of future investigations.

3.6 Chapter Summary

Using knowledge about other hosts in a MANET to plan interactions between clients and service instances can yield benefits in terms of predictability and stability of applications that expand their capability via the opportunistic use of external services. This chapter has described a formal model for proactive service selection in a knowledge managed MANET. It has been shown that such an architecture is desirable in the dynamic environment of a MANET. Details of a proof-of-concept implementation

and evaluation results obtained through simulation are also presented. The knowledge management system described in this chapter has been used in the context of the CiAN workflow management system (described in subsequent chapters) where it has been used to exchange capabilities of hosts, their motion pattern, availability to perform tasks, and other such information critical to the functioning of the overall system.

Chapter 4

A Mobility-Aware Specification for Workflows

The knowledge base described in the previous chapter is an integral part of the CiAN *workflow management system*. As described in Chapter 2, a workflow management system executes workflows that represent structured activities that can be executed collaboratively. The CiAN research can be subdivided into three key projects: (1) the design of a specification language to describe workflows with flexible structures and mobility awareness, (2) the formulation and implementation of a set of algorithms that allocate tasks in a workflow to suitable hosts taking into account their capabilities and mobility pattern, and (3) the design and implementation of the runtime engine that manages the collaborative but distributed execution of the tasks in the workflow by a group of hosts. The design of the specification language is described in this chapter while the runtime system and allocation algorithm projects are described in Chapters 5 and 6 respectively.

There is a close relationship between workflow management systems (WfMSs) and the specification languages they support, with the design and the architecture of the former usually heavily influenced by the latter. As described in Section 2.4, a workflow can be conceptualized as a directed acyclic graph. The workflow specification language is an encoding of the nodes and structure of such a graph in a format that is easily interpretable by a machine. Workflows encoded in the CiAN specification form inputs to the CiAN workflow engine described in Chapter 5, which executes the workflow specification and produces the results of the collaboratively executed workflow.

4.1 Motivation

The most fundamental representation of a workflow is a directed graph where the vertices represent tasks and the edges impose the ordering constraints and structure among the tasks. By annotating the vertices and edges with additional information, a graph can capture a complete workflow specification. However, a graph-based specification cannot show the execution state of the workflow. This has prompted the use of Petri-nets [85] where places and transitions are used to show the structural specification and the tokens used to show the execution state.

While workflow specifications exist today that are based on Petri-net models (e.g. PNML [22, 107]), the most popular workflow specification, BPEL [70] is based upon the hierarchical composition of pre-defined structural components. Specification languages today also mirror the fact that almost all deployed WfMSs are designed to run across corporate LANs or the Internet, i.e., across relatively stable network environments. The designs of these specification languages are geared toward a fully connected distributed setting and do not include the kinds of provisions for flexibility and adaptability that are needed in mobile environments.

When moving to a mobile environment, the rigidity of current generation specification languages become a liability. Changes in a dynamic network environment may result in the execution of the workflow becoming unviable, and the rigidity of the workflow structure then induces a brittleness which can cause errors to occur frequently. In addition, current specification languages do not offer spatiotemporal constructs that allow the author of a workflow to place tasks at a physical location at a particular time.

The work presented in this chapter is motivated by a desire to have a specification for *flexible* and *mobility-aware* workflows that can not only allow the user to specify environmental parameters to tasks but also allows the user to specify a structure that is flexible and can adapt to changes in the execution environment. The CiAN specification is a clean sheet approach to tackling this problem based on the most fundamental representation of a workflow - the directed graph. Subsequent sections of this chapter discuss the design and rationale of the CiAN specification for flexible and mobility-aware workflows.

4.2 Related Work

As indicated in Section 4.1, a sizeable selection of existing languages can be used to specify workflows. Most of these languages use XML syntax such as BPEL [70], WfXML [102], and XLANG [103]. Each of these languages provides a unique set of constructs. What is common across these languages though, is that the constructs are designed with wired networks in mind. For example, in BPEL, the workflow management is centralized, the workflow itself is a monolithic entity, and tasks interact with each other via centralized shared variables, which is appropriate for a wired LAN but is not suited to the dynamic nature of MANETs. Attempts have been made to partition the BPEL specification and facilitate coordination among the pieces by adding special constructs like DoStart, ReceiveStart, DoEnd, and ReceiveEnd in [59]. While that work describes a method for partitioning the specification, it does not support fully decentralized execution, and as such falls short of the needs the CiAN specification addresses. Another approach is described in [14] where the authors parse a BPEL specification, discard all the structural constructs and use the *link* construct to build a more graph-like specification. This approach in effect eliminates the need for all but one of BPEL constructs and is still fairly rigid e.g., it does not allow optional redundant edges.

A Petri net based language such as PNML [22] is more flexible as it adopts the graph structure of Petri nets, but it is designed to be a general purpose language and hence does not possess the specific constructs required to specify a workflow. However, these constructs can be created, as shown by Alt et al. in [7], but as the authors indicate, their language is targeted to grid computing applications exclusively. Perhaps the most versatile Petri net based workflow language is YAWL [107], which uses XML under the hood but can be written using a graphical tool where workflow patterns are represented by a combination of annotated places, transitions, and arcs. The versatility of YAWL is highlighted by the study in [106], which compares the features of YAWL to other systems. In developing the CiAN specification language, there was a choice between developing a language from scratch or extending a language such as YAWL. The former option was chosen for the following reasons: (1) augmenting an existing language with constructs for mobility and context-awareness would constrain the CiAN specification to the semantics of that language; (2) in most cases, the syntax

and the form of the original specification would have needed to be heavily modified which would have required a re-engineering of the runtime environment; and (3) a tight integration between the language and the runtime system was desirable so that the runtime system supported exactly the features of the language and no more, thereby being more forgiving in terms of resource usage.

The next section describes the key features of the CiAN specification, the filter model, which is the key concept around which the CiAN specification is based, and some example CiAN specification code. This is followed by a qualitative comparison of CiAN to existing approaches.

4.3 The CiAN Specification

The goal of the research presented in this dissertation is to build a WfMS for executing collaborative tasks involving people and mobile devices in the physical world. When workflows and WfMSs are migrated from wired network settings to the physical world, it becomes necessary for the specification language to support certain features. The notion of *where* and *when* a task is executed assumes a much larger significance in the physical world than in a purely computational environment, as does the responsiveness of the system to factors in the physical environment. In mobile environments, executions of monolithic specifications can result in vulnerabilities in the system due to a single point of failure, which points to a need for distributeable specifications that are flexible and can be adjusted in response to instantaneous network conditions. Finally, it is crucial to include constructs that maintain the structural integrity and semantics of the workflow, even if the specification is fragmented and distributed. From these observations, three high level requirements can be distilled: (1) the workflow must be completely fragmentable, i.e., it should be possible to divide a workflow into as many pieces as there are tasks in that workflow, (2) there must be support for the basic control flow patterns as suggested by van der Aalst in [106], and (3) the structural semantics of the workflow must be context-dependent, i.e., the semantics of the workflow structure could change depending on the context in which it is executed. Given these requirements and the observations in the previous section, the choice was made to model the CiAN specification using a directed acyclic graph

model with *filters*. This section presents the salient features of the CiAN specification, and the manner in which they are realized within the filters, as well as a description of the syntax.

4.3.1 Specification Features

Fragmenting the Workflow. The adoption of the graph model allows easy fragmentation of a workflow into its constituent tasks (the first requirement) and assignment of individual tasks for execution on available hosts since each task appears as an individual unit connected by edges rather than as part of a composite structure that may not be as easy to partition. In CiAN, the tasks in the workflow are listed in no particular order. Then, treating these tasks as graph nodes, two adjacency lists are built for each task, the input adjacency list and the output adjacency list. If there is an outgoing edge from Task A to Task B, Task B is added to Task A's output adjacency list (Task A is automatically added to Task B's input adjacency list since an outgoing edge from A to B implies an incoming edge to B from A). Since any single task in the workflow is only interested in the tasks from which it receives inputs and those to which it must deliver outputs and is not affected by the remainder of the workflow, the adjacency lists capture exactly the information needed by each single task. Thus, the tasks and their associated adjacency lists can be easily distributed across multiple hosts with the overall structure of the workflow being preserved, albeit in a distributed fashion, which is desirable for MANETs. To ensure structural integrity of the workflow (e.g., checking for dangling edges, lack of lattice structure, etc.), the specification is passed through a checker which lists errors in the workflow in a similar manner as a compiler does for a program.

Context-aware Execution. When executing a workflow in an environment that is as dynamic as a MANET, the emphasis is on designing a system that is *adaptive*, rather than rigid. In traditional workflow specifications, the semantics of the edges and the overall structure of the workflow cannot change once the workflow begins execution which makes for very rigid constraints at runtime. To remedy this, features were added to the CiAN specification that make the workflow context-aware, so that the workflow can adapt to changes in the context in which it is executed. This context-awareness is achieved by the use of *selection conditions*. As shown in Figure

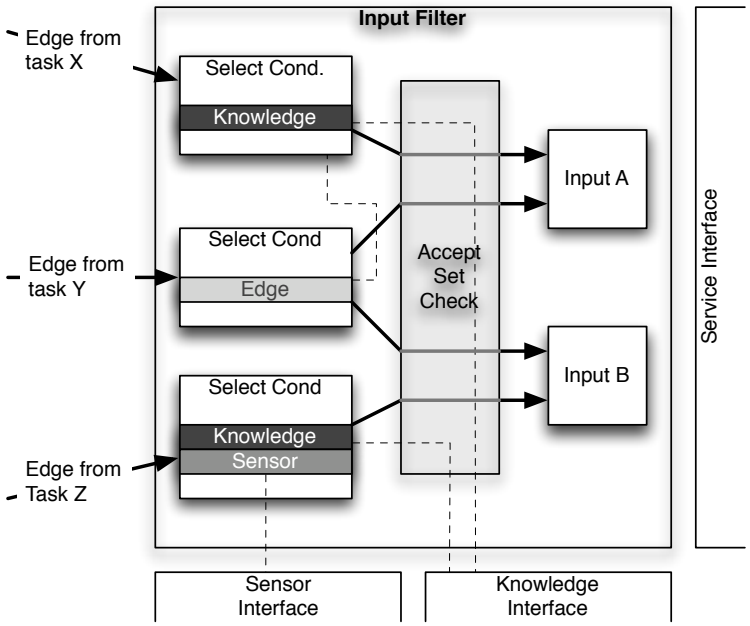

Figure 4.1: Detail of an input filter

4.1, each incoming edge to a task may have one or more selection conditions. Each selection condition has one or more associated sub-conditions. If an edge has at least one selection condition which has all its sub-conditions evaluate as true, then the edge is marked *active*, otherwise the edge is marked as *inactive*. The sub-conditions that make up the selection condition are of the form `parameter name, comparator, value` where parameter name can be the name of another edge, a parameter in the local knowledge base, or the name of a sensor. The comparator is any logical comparison operator, and the value is the value that is being tested against. For example `sensor:velocity, >, 10m/s` is a condition that tests whether the velocity of the host is greater than 10m/s. The use of selection conditions also allows redundancy to be built in. For example, if `edge1` and `edge2` are two redundant edges, it can be ensured that only one is selected by imposing the following conditions on `edge1` and `edge2` respectively: `(edge1, !=, null) AND (edge2, ==, null)` and `(edge1, ==, null) AND (edge2, !=, null)`.

Support for Workflow Patterns. A graph-based workflow can be decomposed into a set of canonical patterns, as suggested by van der Aalst in [106, 108]. These patterns

dictate edge selection (e.g., all edges, one edge, etc.) and specify synchronization semantics. CiAN supports this via *accept sets*. Accept sets are subsets of incoming edges that form acceptable input to the task. If all the edges in an accept set are active, as determined by evaluating their individual selection conditions, then the task is ready for execution. Accept sets provide support for the basic workflow patterns. Support for some advanced patterns require the use of appropriate selection conditions in combination with accept sets. As an example, if `edge1` and `edge2` are two incoming edges, then an accept set containing both edges would result in AND semantics (all inputs required). Two accept sets with one edge in each would result in OR semantics (any one input is acceptable). When an acceptable combination of input edges is available, the data transmitted along each edge is copied to a corresponding input variable (specified as part of the workflow) from where it is read by the service that performs the activity associated with the task. Detailed code examples showing the translation of the workflow patterns to CiAN code is shown in Appendix D.

These features have been presented in the context of input edges. The same features are available on output edges with small differences. The service performing the activity associated with the task writes its results to output variables, which map to output edges. Accept sets for output edges check if there is an acceptable subset of edges that have results written to their corresponding output variables. If so, each of those edges have their selection conditions tested and if any one of them is true, they transmit the data in their output variables to the tasks at the sinks of those edges.

4.3.2 The Filter Model

The filter model has been designed under the assumption that the primary source of flexibility in a workflow will be due to the re-arrangement or elimination of edges, keeping the set of tasks in the workflow intact. Thus the tasks in the workflow are treated as anchors and are represented just as they would be for a workflow that is designed to execute across a stable network. For flexibility, the tasks are preceded and succeeded by the input and output filters respectively (shown in Figure 4.2). These filters encapsulate and manage the dynamics of the workflow structure.

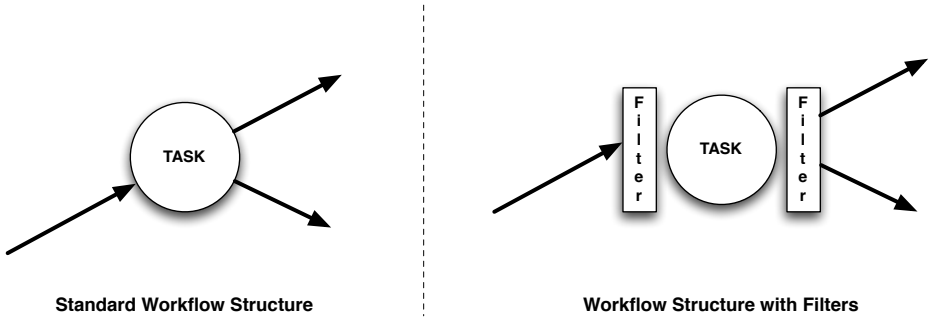

Figure 4.2: Workflow structure with and without filters.

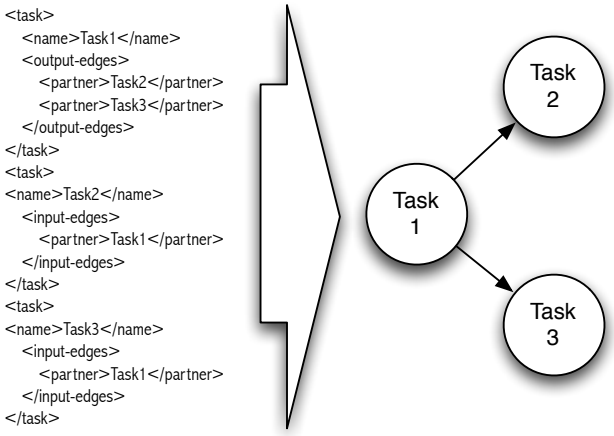

Figure 4.3: Translation of CiAN code to a graph structure

As shown in Figure 4.1, the various conditions, selection conditions, and accept sets associated with a task specification are contained in the filter. As such, bespoke input and output filters are created for each task. The information in the `inputs` section of a task is used to customize a generic input filter for a specific task. The `edge` tags are used to determine how many incoming edges need to be handled, and the `selection-condition` tags to determine the conditions that the filter checks before allowing the data associated with an edge to pass through it. Finally, the filter uses the information in the `accept-set` to determine when the associated task should be begun. Correspondingly, the `outputs` section is used to customize a generic output filter for a specific task in the same manner. Figure 4.3 is a partial view of how a snippet of CiAN specification code corresponds to and is translated to a graph structure.

The input filter handles all the processing to determine if an acceptable set of inputs is available for the task. Once such a set is available, it invokes the task. The output filter waits for the task to complete and then accepts the outputs of that task. As a result of this setup, the task is agnostic to the manner in which its inputs are made available or the destinations to which its output is delivered. This decoupling allows the design of the task execution mechanism to be done assuming a notion of stability in the workflow structure.

The filter also facilitates in providing the task with a notion of network stability. Since the filter structures handle the receiving of inputs and disbursement of outputs to hosts that are responsible for preceding and succeeding tasks in the workflow, they can handle all the dynamics associated with the network, once again abstracting it from the task. A detailed description of the runtime infrastructure to achieve this is given in Chapter 5.

4.3.3 Syntax and Tags

The actual CiAN specification is an XML-based specification where each CiAN workflow must appear in its own file. A sample of the code is shown in Figure 4.4 and detailed explanations of each of the tags appears in Appendix B. A CiAN workflow is delimited by the `<collaboration>` tags. Within these tags, the specification is split into two elements – the header and the body. The header declares the non-functional information about the hosts and the network that this workflow relies on while the body contains the actual task definitions. The header itself is split into two sections delimited by the `<knowledge-base>` and `<sensor>` tags. The knowledge base section specifies the names of the parameters found in the knowledge base of hosts that this workflow relies on. This knowledge base is the same as the one presented in Chapter 3. Each of these parameter names are delimited by `<knowledge-var>` tags. The sensor section specifies the names of the various sensor parameters that the workflow relies upon and these are delimited by `<sensor-var>` tags (the sensors are assumed to be attached to the local device). The absence of these parameter values at runtime does not halt the execution of the workflow but could compromise its flexibility.

```
<collaboration>
        <knowledge-base>
                <knowledge-var>host-capabilities</knowledge-var>
                <knowledge-var>motion-profile</knowledge-var>
        </knowledge-base>

        <sensors>
                <sensor-var>location</sensor-var>
                <sensor-var>time</sensor-var>
        </sensors>

                <task>
                    <task-name>ExampleTask</task-name>
                    <earliest-start><date>09/21/2007</date><time>09:05:00</time></earliest-start>
                    <duration>180</duration>
                    <deadline><date>09/21/2007</date><time>09:09:00</time></deadline>
                    <loc-x>0</loc-x>
                    <loc-y>0</loc-y>

                    <inputs>
                            <edge>
                                    <name>InEdge1</name>
                                    <var>InputValue</var>
                                    <partner>ExamplePredecessor</partner>
                            </edge>

                            <accept-set>
                                    <set>
                                            <name>InEdge1</name>
                                    </set>
                            </accept-set>
                    </inputs>

                    <activity>
                            <input-vars>
                                    <var>InputValue</var>
                            </input-vars>

                            <service>http://cian/Example</service>
                            <method>doExampleTask</method>

                            <output-vars>
                                    <var>OutputValue</var>
                            </output-vars>
                    </activity>

                    <outputs>
                            <edge>
                                    <name>OutEdge1</name>
                                    <var>OutputValue</var>
                                    <partner>ExampleSuccessor</partner>
                            </edge>

                            <accept-set>
                                    <set>
                                            <name>OutEdge1</name>
                                    </set>
                            </accept-set>
                    </outputs>
          </task>
</collaboration>
```

Figure 4.4: CiAN code example showing a single task within a workflow

The body of the workflow comprises one or more task definitions, delimited by the <task> tag. The tasks can be specified in any order regardless of their position in the workflow. A task definition consists of (1) a task name, delimited by the <task-name> tag and unique in the scope of a workflow, (2) the earliest start time of the task delimited by the <earliest-start> tag, (3) the duration of the task delimited by the <duration>tag, (4) the deadline for completing the task delimited by <deadline> tags, (5) the location of the task delimited by <loc-x> and <loc-y> tags and three additional sections: input, activity, and output, delimited by the <inputs>, <activity>, and <outputs> tags respectively. The input section defines one or more edges, each delimited by the <edge> tag. Each edge represents a connection to another task in the workflow and is analogous to the edge in the graph representation of the workflow. The input section also defines one or more accept sets, delimited by the <accept-set> tag. Inside each <set> element is a list of names of edges that represents subsets of the input edges that are considered valid input. For example, if a task had 6 incoming edges labelled 1 to 6, but those 6 edges were really a pair of redundant inputs consisting of 3 edges each, then values from the first of the redundant pairs (edges 1 to 3) are equally valid as the values from the second of the redundant pairs (edges 4 to 6) and it is not necessary to wait for all the inputs. The accept set captures this information. The outputs section is similar to the inputs section in structure except that the edges are outgoing rather than incoming.

The activity section specifies information about the actual service that needs to execute to complete a particular task. This section specifies the names of the input variables to the service, delimited by the <input-vars> tag and the names of the variables to which the service writes its output, delimited by the <output-vars> tag. It should be noted here that since the input filter, service, and output filter for a given task reside on the same host there is no issue of consistency or overhead in using these variables. Finally, the actual location or URI of the service is delimited by the <service> tags and the method to be invoked on that service by the <method> tags.

Workflows specified in CiAN are flexible and context-sensitive due to the way an edge is structured. Each edge has a name, delimited by the <edge-name> tag, which is unique in the scope of a task. The <partner> tag is used to define the name of the task at the other end of the edge. For incoming edges this is the source task, while

for outgoing edges it is the sink task. The `<var>` tag specifies the name of the local variable to which the value transmitted along the edge is written (in the case of input edges) or from which it is read (in the case of output edges). These variables are the same as those that appear as input variables and output variables in the activity section. Values transmitted along an incoming edge are written to an input variable from where the service reads it. The service's output is written to an output variable from where it is read and transmitted over an outgoing edge.

In addition to this basic information, each edge can specify zero or more selection conditions (denoted by the `<select-cond>` tag) which may consist of multiple sub-conditions. If any one of the selection condition blocks have all their sub-conditions (denoted by `<cond>`) evaluate to true, then the value of the edge is considered acceptable input to the task. Each condition is a three-tuple consisting of a parameter name denoted by `<param>`, a comparator, and a value. The parameter can be of four different types: 1) `knowledge:hostname:paramname`, which refers to the non-functional parameter called `paramname` of a host called `hostname` which may be found in the local knowledge base, 2) `sensor:sensorname`, which refers to the value of a sensor called `sensorname`, 3) `edge:edgename`, which refers to a value transmitted along another edge whose name is `edgename`, and 4) `var:varname` which refers to the value of a local variable called `varname`. The comparator may be the operators {`<`, `>`, `<=`, `>=`, `==`, `!=`}.

The graph like structure of the CiAN specification language helps support all the basic control flow patterns. If parallelism is required, multiple output edges can fan out from a single task. If sequential processing is desired, then only one output and input may be used. Merges and splits can be built in similar ways. If an XOR merge is desired, each edge can be tagged with a selection condition that requires all other input edge values be equal to null. This way, the first edge that yields a value is selected as input. Also, context sensitive selection is implemented in the same way, e.g., an edge is not selected unless the temperature sensor has a value greater than 32.

4.4 Qualitative Comparisons

Recall that the three required features of the CiAN specification were: (1) easy fragmentation, (2) context-aware execution, and (3) support for the basic control flow patterns suggested by van der Aalst. Easy fragmentation was achieved by the use of an underlying graph model and self-contained task descriptions. Context-awareness was achieved through the use of selection conditions, and support for patterns through the combined use of selection conditions and accept sets.

The ability of the CiAN specification to support the control flow patterns is detailed separately in Appendix D. This section highlights the context-awareness features of the CiAN specification via the examination of three example scenarios, each of which progressively test the limits of existing approaches. For each scenario, the applicable workflow is specified using the CiAN specification and two other languages that are arguably the leaders in their respective spaces– BPEL [70] which is the most popular workflow specification language for commercial deployments, and YAWL which is one of the most mature systems to come out of academia. The cases and evaluations are shown below.

4.4.1 Case 1: Writing a Paper

The first case examines a scenario where an academic paper must be written collaboratively. The workflow shown in Figure 4.5 depicts the structure of this activity. Initially, the introduction is written to set the tone and message of the overall paper. Once this is done, several sections can be written in parallel prior to the evaluation and conclusion which must be written in order.

4.4.2 Case 2: Checking Out From an Online Store

The second case examines a scenario where a customer is checking out of an online store. When the checkout button is pressed, the shipping and the tax is calculated in parallel and applied towards the total for the order. If the total is greater than $

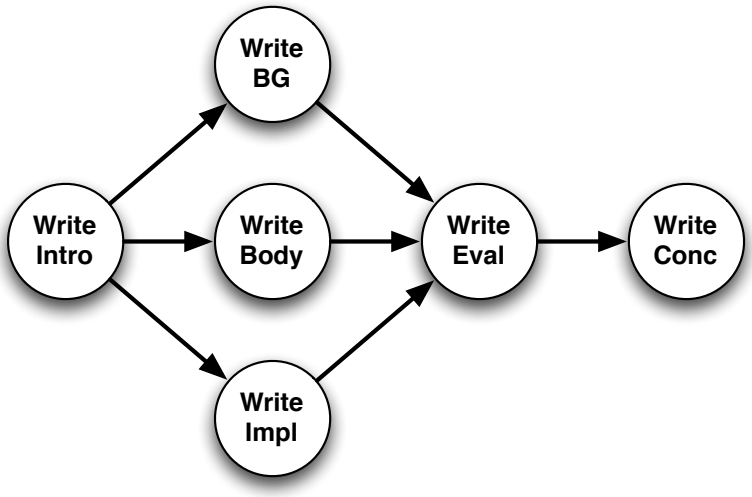

Figure 4.5: CiAN Specification Case Study 1: Writing an Academic Paper

100, then a coupon is included with the order as part of a promotion. The workflow structure is shown in Figure 4.6.

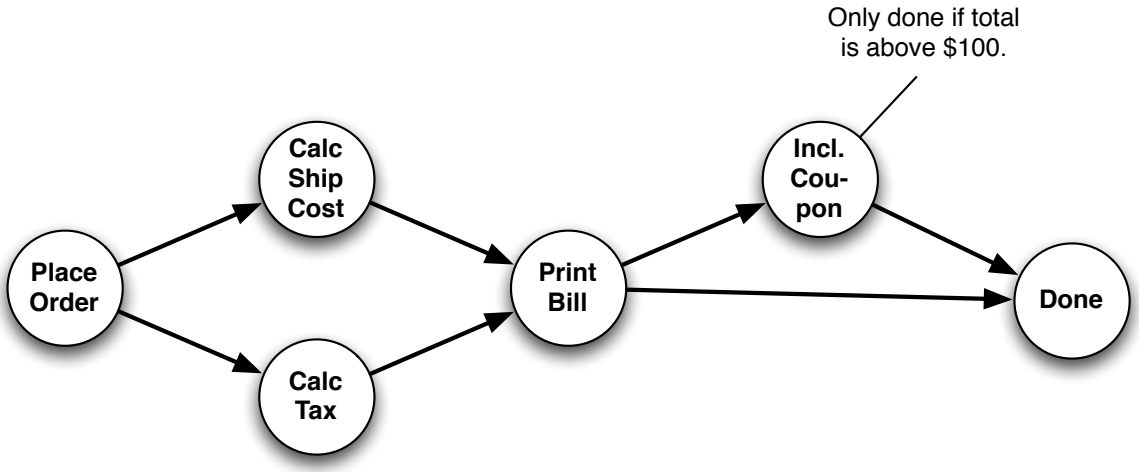

Figure 4.6: CiAN Specification Case Study 2: Checking Out From An Online Store

4.4.3 Case 3: Soil Sample Analysis

The final case examines a scenario where scientists are conducting soil sample tests in the vicinity of a chemical plant at a specified time (shown in Figure 4.7). Once the initial sample is taken, a basic analysis is done. Only if this shows a level of toxicity over a certain threshold is the second analysis done. If the second analysis shows a critically high toxicity level, then an emergency response is immediately called. If the level is marginal, a confirming analysis is done and emergency response called only if a high level of toxicity is confirmed. The preparation of a report and having the supervisor sign off on it is done regardless of the toxicity levels.

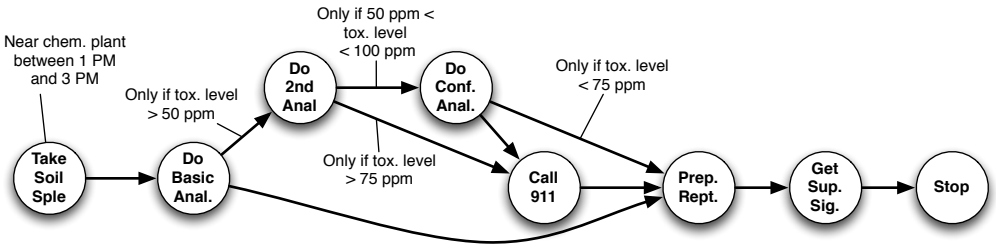

Figure 4.7: CiAN Specification Case Study 3: Soil Sample Analysis

4.4.4 Comparison

Figure 4.8 shows a summary view of the success of the CiAN specification versus the other two languages in specifying the workflows associated with the three cases. For case 1, where the workflow is fairly straightforward, all three languages do equally well, though the code for CiAN is much longer than the others. In the second case, with conditionals, again all languages do well except the amount of code for BPEL and YAWL expand to handle the two options. In the third, where there are several options and the actions taken depend on factors sensed during the workflow execution, CiAN is much better equipped to handle the situation and the other languages must resort to external solutions or syntactic manipulation to correctly specify the workflow. For example, in BPEL, to have flexibility, the standard structural constructs must be discarded in favor of the `link` construct and environmental variables can only be introduced as system variables using an external solution. Similarly in YAWL,

sensor values and environmental knowledge would need to be provided by an external solution and integrated with the core engine. As such, these languages are at a disadvantage to CiAN in mobile and rapidly evolving situations.

	CiAN	BPEL	YAWL
Case 1 Writing an Academic Paper	+ (code is longer)	+	+
Case 2 Checkout from Online Store	+	+	+
Case 3 Mobile Soil Analysis	+	X	X

Figure 4.8: Summary results of comparative analysis

The detailed code for each case for each language appears separately in Appendix E.

4.5 Chapter Summary

The CiAN specification has been designed from the ground up to be a workflow specification language that supports flexibility and adaptability of the workflow structure in response to mobility. In addition, the specification language contains means to describe spatiotemporal properties of tasks, an important feature when workflows are being executed in the physical world. Through a case-study-based qualitative comparison, it has been shown that CiAN can specify adaptive workflows for scenarios in which current approaches fail. This is a useful step towards a new generation of flexible workflow specifications.

Another aspect that was kept in mind when designing the CiAN specification was integration of the specification with the runtime system. System design concerns significantly influenced the design of the language resulting in a very tight and intuitive integration between the specification and the runtime components– sections of task

specifications translate directly into customized runtime components. The link between the specification language and the runtime system is further highlighted in the next chapter.

Chapter 5

CiAN: A Workflow Management System for MANETs

Workflow specifications written in the CiAN specification language are executed by the CiAN workflow engine. The CiAN workflow engine is a workflow management system (WfMS) targeted to mobile ad hoc networks (MANETs). As described in Section 2.4, a WfMS takes a workflow specification as input, allocates individual tasks to qualified hosts, and then manages the execution of these tasks across multiple hosts until all tasks are complete. In the case of the CiAN workflow engine however, there is an added requirement that all the aforementioned functions must work across a dynamic and volatile mobile environment. This chapter describes the design challenges, resulting software architecture, and implementation of arguably the first WfMS that is designed for operation in a fully ad hoc mobile network.

5.1 Motivation

The use of workflows to compose software services is a well-established concept, evidenced by the existence of workflow languages such as BPEL [70], YAWL [107], WfXML [102] and Workflow Management Systems (WfMSs) such as ActiveVOS (previously ActiveBPEL) [3], BizTalk [62], and the Oracle Workflow Engine [74], to name a few. WfMSs take a workflow specification, which describes a complex activity split into a structured set of smaller tasks as input. WfMSs choose software services to perform each of the tasks in the workflow specification and *orchestrate* their execution such that, working in combination, they are able to complete the complex

activity specified. While workflows are used in many applications today, arguably the most common use is in the specification and execution of complex, multi-step *business processes*, such as loan application processing, insurance claim adjustment, etc. Most of these business processes run across a stable network (wired or nomadic wireless) where the network topology is stable and connectivity among participants and WfMSs is assured. As such, most commercially available and deployed WfMSs today are designed to run in stable environments using centralized architectures and orchestration of services.

It should be observed however, that there is no intrinsic constraint in the workflow model that prevents it from being used in much broader contexts. Rather, it is the lack of a suitable WfMS that prevents workflows from being used away from the so-called network core. The goal in designing the CiAN specification and engine is to broaden the scope of workflow usage to mobile settings, where the scope of the workflow extends beyond computational tasks to tasks that are performed in the physical world and the entities that perform the task are either software services or humans with specific skill sets. In other words, the target is situations where a large number of people (assisted by a library of software services) need to collaborate, but where setting up a traditional WfMS over a temporary LAN, even if possible, is not the most desirable option. This motivation paves the way for a WfMS that runs on mobile devices over a MANET. Mobile workflow technology can be applied to a wide range of applications such as emergency response, outdoor hospitality events, and evidence-based practice, to name a few.

Migrating a WfMS from a wired or nomadic setting to MANETs is not trivial. In wired systems, the WfMS and all the services it exploits to complete tasks can coordinate with each other at any time due to the presence of the fixed network. A MANET environment does not offer this luxury. The dynamic network topology and unpredictable disconnections that are common in a MANET significantly reduce the ability of the participants to communicate with each other or with a central coordinating entity. In addition, the WfMS cannot execute on a single host because the loss of that host compromises the collaboration among all other hosts in the MANET. Hence, the workflow execution needs to proceed in a decentralized, *choreographed* fashion, with redundancies built in to accommodate errors due to the inability of hosts in the

network to communicate with each other. Solving these problems requires a redesign of WfMSs at the most fundamental level.

The remainder of this chapter describes the system design and implementation solutions that in combination make up a WfMS for MANET environments.

5.2 Related Work

At present, innumerable WfMSs are available as both commercial and open source software, including FLOWer [76], AgentWork [64], Caramba [19], Groove [61], and I-Flow [26]. ActiveVOS [3], JBoss [46], and the Oracle Workflow Engine [74] are just a few of the engines available today that run BPEL workflows, while BizTalk [62] supports XLANG. Each of these engines is designed for orchestrated operation in wired settings.

As a result of being designed for wired environments, a lot of the systems available today do not have the requisite flexibility for MANET environments. For example, in BPEL-based systems, transfer of data between tasks is done by centralized shared variables– an approach that is not practical in a MANET. A workaround to this was proposed in [14] which uses message passing to distribute data in a wired setting. However, this approach cannot handle the dynamism of MANETs. MoCA [90] is better suited to mobility due to its use of proxies. Mobile users maintain proxies on stable wired nodes which communicate with the MoCA directory service (DS), configuration service (CS) and context information service (CIS) over a wired link. These three modules coordinate the collaborations of multiple mobile hosts. However, since most of the MoCA infrastructure resides on a wired network, only nomadic mobility can be realistically supported as mobile hosts are dependent on the DS, CS, and CIS for coordination. In AWA/PDA [99], the authors adopt a mobile agent-based approach based on the GRASSHOPPER agent system. They define five types of agents, the Workflow Agent (WA), Process Agent (PrA), Task Agent (TA), Worklist Agent (WlA), and Personal Agent (PA). The PA is a daemon agent on a mobile device and is not logically mobile. The rest of the agents are logically mobile and can opportunistically move from one host to another. For example, a TA, which coordinates a single task can migrate to a PDA, work while disconnected, and then

report results when a connection is available to the WA. Since all necessary components of the system are logically mobile, this type of architecture is better suited to mobile environments. However, the dependence on a WA or WlA means that the hosts carrying these agents must always be within communication range. While this may be simulated by moving the process from one host to another this is computationally expensive and it may not be possible to guarantee connectivity with all hosts. Finally, Exotica/FDMC [6] describes a scheme to handle disconnected mobile hosts.

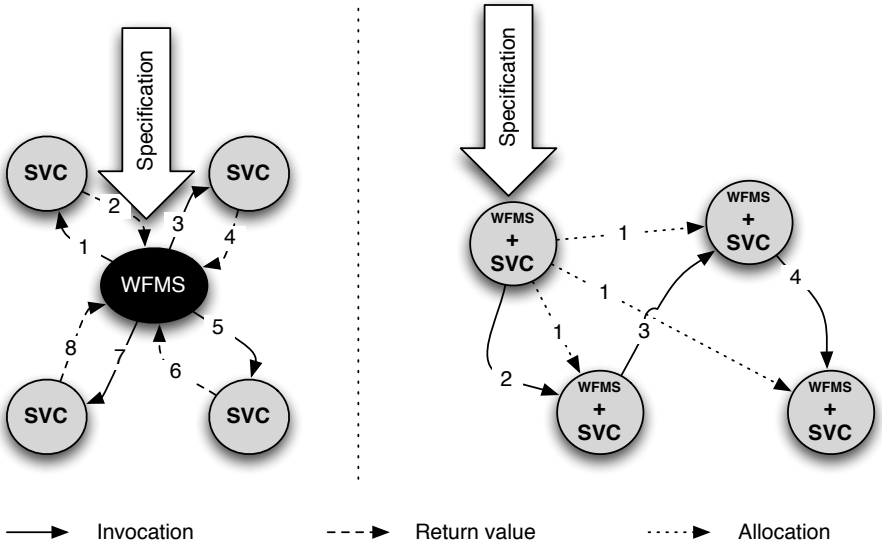

Figure 5.1: Orchestration vs. Choreography

WORKPAD [13] is designed to meet the challenges of collaboration in a peer-to-peer MANET involving multiple human users. WORKPAD works by augmenting the basic workflow specification with directives (mainly involving moving a host so that it can communicate with another) so that the tasks in the workflow can hand off data to subsequent tasks and thereby advance the execution of the workflow. WORKPAD's shortcoming is that it requires at least one member of a MANET to coordinate with a central entity that coordinates the mobile devices, manages disconnection, and augments the workflow specification with mobility directives. This dependence means that WORKPAD cannot survive in ad hoc mode for an extended time. The work presented in this chapter is targeted to an environment similar to that of WORKPAD. However, the approach is different. Rather than insert directives to move hosts from

one place to another, hosts are allowed to move pseudo-freely and other constructs are used to ensure successful completion of workflows as described in subsequent sections.

To achieve true distributed management, the system must support a choreographed style of execution (the differences between orchestrated and choreographed systems is shown in Figure 5.1). With a choreography-based system, a leading concern is the process by which a workflow is distributed across various participants and then executed. In [68], the authors describe the process by which a monolithic workflow specification can be fragmented and eventually distributed across multiple hosts while in [14], the authors parse a BPEL specification, discard all the structural constructs and use the *link* construct to build a more graph-like specification. Several systems exist that achieve partial choreography, a survey of which appears in [36]. OSIRIS [92] is one such system where individual nodes maintain a hyperdatabase (HDB) to which service execution requests are pushed by a set of global process repositories. The choice of to whom to push the request is handled by established load balancing techniques. ADEPT$_{Distribution}$ [9] describes a scheme for distributed execution of workflows such that the number of network messages is minimized. Additional efforts are ongoing to define protocols and standards for choreography such as in WS-CDL [112].

The CiAN workflow engine is a choreographed WfMS which uses an *allocation* process to coordinate division of responsibility among hosts and a combination of filters (described in Chapter 4) and host-agnostic publish-subscribe protocol for subsequent choreographed execution. It should be noted that this chapter touches only briefly on the allocation process, referring only to the software components needed to accommodate this function. The algorithmic aspects of allocation are treated separately in the next chapter.

5.3 System Design

The CiAN middleware is designed to be a *choreographed* WfMS. A choreographed execution model (as opposed to the *orchestrated* [109] models in common use at present) has two phases– a negotiation phase and a peer-to-peer execution phase . During the negotiation phase, the hosts participating in the choreographed execution determine

their pattern of interaction. This pre-determined pattern of interaction is then used to guide a message passing peer-to-peer protocol that advances the execution of the workflow. In CiAN, the *allocation* process described in the next chapter serves the function of the negotiation phase as it helps determine which hosts are responsible for which tasks. Once tasks have been allocated, the primary responsibility of the choreographed execution engine is to ensure that the tasks get executed without violating any structural or synchronization properties of the workflow as a whole. This is done in two phases as described below.

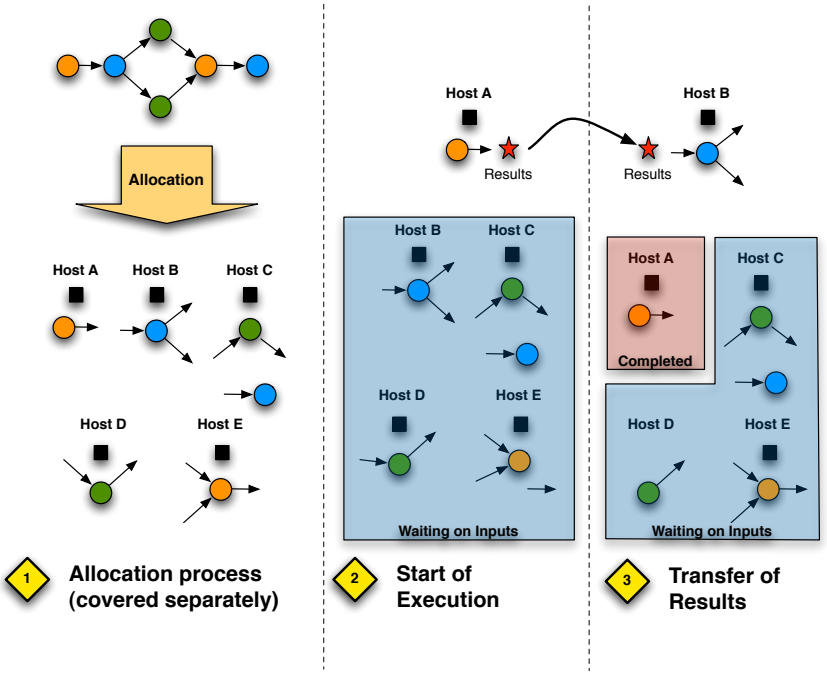

Figure 5.2: Steps in Executing a CiAN Workflow

Initial Installation. A *coordinator* allocates a task to a host by sending the target host a full specification of the task (shown in Step 1 of Figure 5.2). When a host receives such a specification, it installs it locally. The process of installation includes setting up the appropriate data structures and threads to evaluate the selection conditions and accept sets associated with that task (this process is described in more detail in the Section 5.5). Once the task is installed, it creates a *subscription* for each incoming edge to the task. A subscription is a request for the output of the tasks that are at the source of the respective edges. These subscriptions are disseminated into the network

using one of the protocols described in Section 5.4. At this stage, the host is ready to execute the task.

Task Execution. While the task is ready to execute once it has disseminated its subscriptions, it cannot move forward as long as it does not have a valid set of inputs. When an input value for the task is received by the host (from another host across the network), it evaluates the selection conditions and accept sets. As indicated in Section 4.3, this evaluation is the responsibility of the software filters that precede the task. If an acceptable combination of inputs has been received, the host executes the task (usually by invoking a local service or prompting the user to take an action) and produces outputs. The output data is then evaluated (by the output filter) in terms of the output selection condition and accept sets and is subsequently disseminated using the same protocols described in Section 5.4. The protocols ensure that the data from a task is delivered only to those tasks that have declared interest in having that data.

Given this model, a typical execution occurs as follows: the first task in a workflow has no inputs and can begin execution immediately (Step 2 of Figure 5.2). Once it is finished, it produces output data which are then transmitted over the MANET to hosts that are responsible for executing the next set of tasks. This data then becomes the inputs for those tasks, thereby triggering their execution (Step 3 of Figure 5.2). This process continues until the last task is done executing.

5.4 Communication Protocols

The piece that glues the individual hosts executing their allocated portions of the workflow into a cohesive distributed workflow execution is the communication protocol for exchanging data among hosts. Such a communication protocol must be *workflow-oriented*, i.e., it must be aware of and accommodate the semantics of the workflow structure and also be resilient to the dynamism of a MANET environment. This section shows how current approaches tackle the problem and which is then contrasted with the host-agnostic publish-subscribe-based approach of CiAN.

In current MANET communication protocols such as [83, 47, 84, 48], the approach is to assign an address to each host in the MANET, and then build routes between hosts that are either maintained continuously or created on an on-demand basis. As acknowledged in many papers, MANET routing protocols have an inherent disadvantage in that routes in a MANET do not last long and can be very expensive to maintain. In addition, for the purposes of CiAN, MANET protocols have an added disadvantage in that they are host-centric, i.e., addressing is done by host rather than by any property of the workflow.

The CiAN approach is built on the notion of *disconnected* routes [37], which do not need the entire route to be connected end-to-end like traditional MANET routes. Instead, disconnected routes use a store-and-forward approach where data is transferred between hosts when there is a direct (single hop) link between them. In addition, disconnected routes only require that the connection for the next hop be available for communication rather than the entire route. By exploiting these direct links between hosts in a temporally sequenced manner, disconnected routes can deliver data between pairs of hosts which are never directly connected with each other or connected end-to-end via a route.

While disconnected routes alleviate the problem of route maintenance in a MANET, they do not eliminate the fact that the addressing is still by host. To get around this, a publish-subscribe-based overlay layer is used by CiAN for communication. This layer is based on the principle that the hosts which have been allocated tasks will publish the results of those tasks and will subscribe for any required inputs to those tasks. By having the publish-subscribe layer deal in tasks, task input values, and task output values, the communication protocol is made to be workflow-oriented rather than host oriented, thus gearing it for the most stable feature of the system.

The publish-subscribe scheme works as follows: Each task in the workflow is numbered using a breadth first graph traversal algorithm. When a host is allocated a task or set of tasks, it creates an ordered vector consisting of the numbers of those tasks. Once this vector has been created, it removes the lowest number from that vector. This number then becomes that host's identifying number until the task corresponding to that number has been completed. At this point, the host chooses the next number from the vector and uses it as its identifying number. If the vector is empty, the host

assigns itself a null value. The null value is also assigned if the host does not have any tasks allocated to it.

Any data generated by the host contains two fields: the (current) number of the *task* generating the data and the number of the *task* that the data is targeted to. Any subscription similarly has the number of the task requesting the data and the number of the task from which the data is requested. An additional field is included in both publications and subscriptions to track the current routing status of the message. Observe that since hosts essentially number themselves based on the number of the tasks they have, there is a simple mapping from tasks to hosts without any host needing to be aware as to which specific host is executing a particular task. For example, a host may simply subscribe to the output of "Task 5" and the routing protocol then ensures that the request reaches the host who is publishing data that is the output of "Task 5."

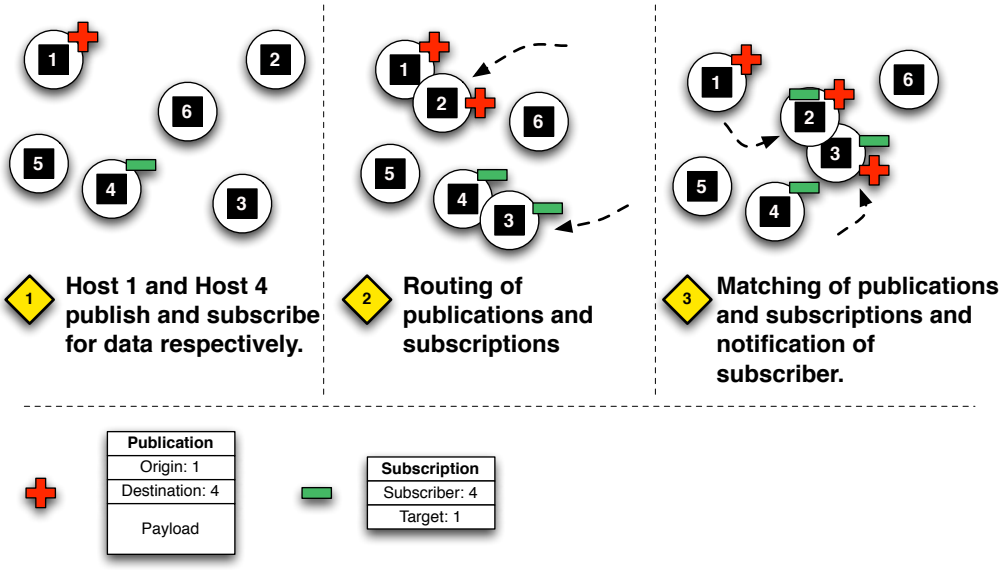

Figure 5.3: Steps in the Publish-Subscribe Communication Protocol

The data and subscriptions are routed among hosts using three distinct schemes, each of which have their own strengths and disadvantages. What is common among them is the restrictive flooding of data and subscriptions to a limited set of hosts in a way that improves their chance of reaching the recipient.

Routing Scheme 1. In Scheme 1, data and subscriptions are restricted to hosts that have numbers in the range between the originating host and the recipient host. Routing is also allowed to hosts that have a null value as their number. Data is routed to any host that has a number between the generating task number and the target task number or has no number in a strictly increasing fashion. Subscriptions are routed similarly but in a strictly decreasing function. Routing to a host with no number is considered neither a decrease nor an increase. In both cases, when the message is routed, the current routing status field is updated to reflect the number of the host to which it is routed (unless it is null in which case the value is not updated). This ensures that the message always makes forward progress towards the target host. A pictorial example of this scheme is shown in Figure 5.3.

Routing Scheme 2. Scheme 2 is a variation on Scheme 1 except that the allowable range is different. Data can be routed to any host with a number that is between the lowest possible task number in the workflow and the target task number. Subscriptions are routed to a host with a number between the highest possible task number in the workflow and the target task number. As with Scheme 1, routing is possible at any time to a host with a null value number.

Routing Scheme 3. Scheme 3 is the most flexible of all the schemes and is based on Scheme 1 with one exception. Data and subscriptions can be routed outside the permissible range but this triggers a counter. If the data or subscription moves to a host in range (as defined by Scheme 1) before the counter expires, the counter is reset, otherwise the data or subscription is destroyed. This makes it possible for messages to be routed to any host, but also ensures that messages that do not make sufficient progress are destroyed.

Scheme 1 generates the lowest number of messages in the network but is restrictive in the sense that the number of hosts that a message can be routed to is much smaller than the total number of hosts collaborating. Scheme 2 increases the permissible range but generates additional messages. Scheme 3 maintains the low range of Scheme 1 but allows limited transgressions which represents the most favorable trade-off between number of messages and number of hosts to which the message can be routed. The relative performance of the three schemes is analyzed in detail in Section 5.6.

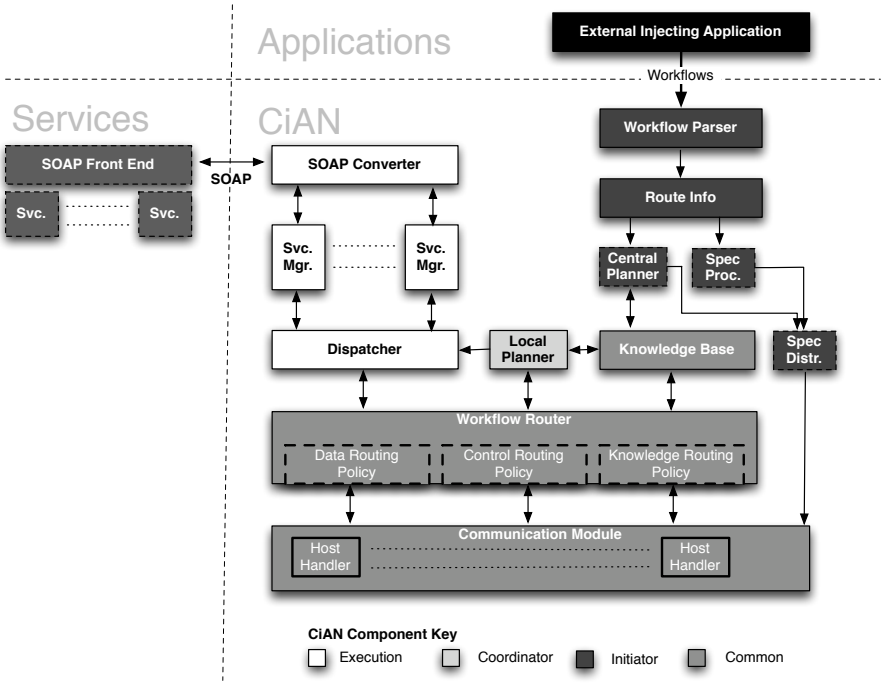

Figure 5.4: CiAN System Architecture.

5.5 Implementation

The CiAN workflow engine is implemented in Java SE 5 and uses two external packages– Sliver [35] and kSOAP [42]– to invoke Web services as part of workflow task execution. CiAN also uses the JGraphT library [21] to verify the acyclic graph structure of any workflow that is input to the system. This section is organized as follows. First, some common implementation details are presented followed by the description of the communication infrastructure and the publish-subscribe protocol that is used to exchange data among hosts. This is followed by a brief description of the task allocation infrastructure (included here for completeness but presented in more detail in the next chapter) before concluding with a presentation of the choreographed execution engine. A global view of the architecture of the system showing the various functional components appears in Figure 5.4.

Launching CiAN. The CiAN middleware is capable of running in three modes: *initiator*, *coordinator*, or *worker*. As mentioned in previous sections, the initiator injects the workflow into the system and fragments it, the coordinator allocates the tasks,

and the workers perform the tasks. When a user launches CiAN, he/she provides a host name (assumed to be unique), the mode in which CiAN is to be launched, and a runtime properties file that defines the policies that the CiAN instance should use (policies are covered in detail later in this section), and a host configuration file which contains information about the host's schedule and the services it possesses. If the initiator mode is used, the user also passes the name of the file containing the workflow specification to the system. All these pieces of information are passed to the system as command line parameters. Depending on which mode is used, CiAN will automatically perform certain actions. In initiator mode, it fragments the workflow and passes it to available coordinators. In coordinator mode, it stands by to receive task sets, and in worker mode, stands by to receive task solicitations and allocations.

Communication Infrastructure. The communication infrastructure of CiAN is encapsulated in the `CommunicationModule`, `WorkflowRouter`, and `HostHandler` classes. The `CommunicationModule` class implements a beaconing protocol that periodically broadcasts a host's information and listens for beacons from other hosts. When three beacons are received in a row from a host, the `CommunicationModule` creates a `HostHandler` which is responsible for setting up a socket connection to the newly discovered host and the object input and output streams. This creates a channel of direct communication between a pair of hosts. This direct communication channel is used during the process of allocating chunks of tasks to coordinators by the initiator and also during the allocation process that occurs between hosts and coordinators. Note that the direct connection approach is possible in both these cases because coordinators are required to be co-located with the initiator initially and hosts are also required to be co-located with coordinators when bidding on tasks.

The only function not covered fully by this style of communication is the data exchange between two hosts as part of the workflow execution process. As described in the previous section, when hosts are directly connected, they exchange publications and subscriptions according to one of three schemes. Whenever a new `HostHandler` is created, the `CommunicationModule` notifies the `WorkflowRouter`. The `WorkflowRouter` then assembles a list of data and subscriptions that must be transmitted to the host encountered (as defined by the routing scheme in use) and passes them to the `CommunicationModule` to send to the other host. Conversely, when data and subscriptions are received, the `CommunicationModule` passes them up

to the `WorkflowRouter`. The `WorkflowRouter` maintains lists of subscriptions and data that it has encountered thus far. If a newly received data or subscription results in a match, the `WorkflowRouter` instructs the `CommunicationModule` to send the data to the subscribing host using AODV [84].

It should be noted that the `WorkflowRouter` routes more than just data messages. It can also route control messages that are used to issue commands within the CiAN system and knowledge messages that are used to gossip knowledge about other hosts in the network. The `DataRoutingPolicy`, `ControlRoutingPolicy`, and `KnowledgeRoutingPolicy` are pluggable interfaces (which may have different concrete routing implementations for expandability). This flexibility has already been used to implement each of the three routing schemes as plug-able data routing policies.

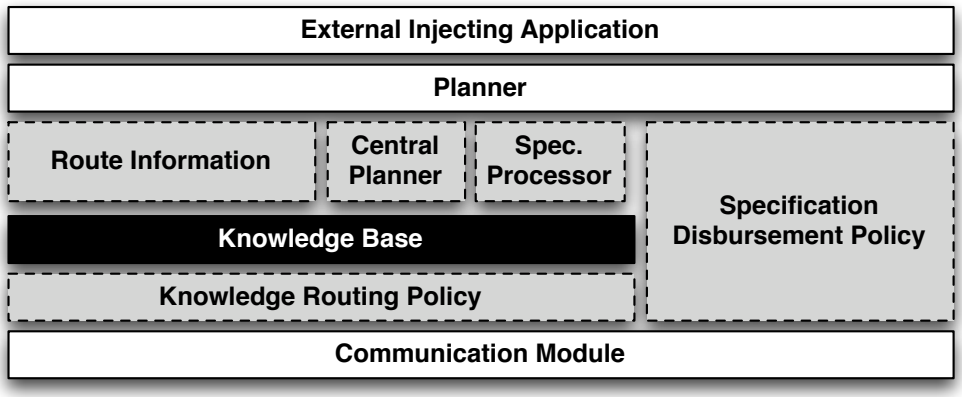

Figure 5.5: CiAN architecture for the Initiator and Central Planner

Workflow Initiation and Allocation Infrastructure. Workflows are injected into the CiAN system using the command line on a host that is running in initiator mode, the architecture for which is shown in Figure 5.5. The `WorkflowParser` reads the specification and converts it into executable objects. These objects are then passed to the `RoutingInfo` component which assigns `RouteInfo` to each task (the `RouteInfo` encodes the topological depth in the workflow graph as described in the previous section). Once the workflow has been augmented with this information, two subsequent actions are possible. If a centralized allocation scheme is being used, the workflow is sent to the `CentralPlanner`, which implements a centralized allocation algorithm and returns the workflow with each task annotated with the identifier of

the host that it is allocated to. If a distributed allocation scheme is used, the workflow is passed to the `SpecificationProcessor` which breaks it into smaller chunks using the k-min cut or geographic cut [97] approach. The chunks are then passed to the `SpecificationDistributionPolicy` to be sent to the coordinators as control messages via the `CommunicationModule`. It should be noted that the `CentralPlanner` and `SpecificationProcessor` implement a common interface and can be plugged into the architecture of the initiator as dictated by the type of allocation algorithm in use.

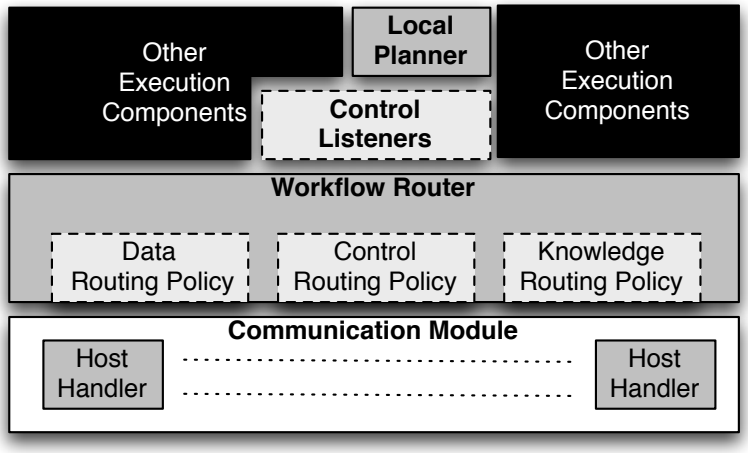

Figure 5.6: Subset of CiAN architecture for distributed allocation

On the coordinators, when a chunk of tasks is received by the `CommunicationModule`, it passes them on to the `ControlRoutingPolicy` of the `WorkflowRouter`. The `WorkflowRouter` allows other components of the middleware to register themselves as listeners for data, control, and knowledge messages. The `LocalPlanner` is one such component that registers itself as a listener for control messages. When the chunk of tasks is received by the `WorkflowRouter`, it passes it to all the listeners. The `LocalPlanner` receives the chunk of tasks and installs it locally. The functionality of the coordinator described in Section 6.3.2 is implemented within the `LocalPlanner`. It should be noted though that the `LocalPlanner` is an abstract class that can be extended by one or more concrete coordinator implementations. When a coordinator needs to send messages to a host, it does so by passing it to the `ControlRoutingPolicy` of the `WorkflowRouter` which in turn sends it out using the

CommunicationModule. Figure 5.6 shows the components of the CiAN architecture dedicated to distributed planning.

On the worker hosts, a similar setup exists, except that the implementation of the LocalPlanner reflects the host's end of the protocol rather than the coordinator's. When a solicitation is received (as a control message), it is passed up via the WorkflowRouter to the LocalPlanner which analyzes the solicitation and prepares a bid if appropriate. The bid is then transmitted via the WorkflowRouter and CommunicationModule back to the coordinator. Future communications between coordinator and worker host are handled in an identical manner.

Task Execution Infrastructure. When a coordinator allocates a task to a host, it sends the task specification as a control message to that host. This message is received by the LocalPlanner as described before. The LocalPlanner then passes this specification to the Dispatcher which creates a new ServiceManager. The ServiceManager is responsible for invoking the service associated with that task. Within the ServiceManager are the InputFilter and OutputFilter. These filters implement the input selection conditions and accept sets and the output selection conditions and accept sets respectively. CiAN provides a GenericFilter class which can be parametrized and customized to each task using the information in the task specification. The ServiceManager also contains information about the service to be invoked as part of the task execution. Once the ServiceManager has been created, it examines the input edge information of the task (found as part of the InputFilter instance it owns) and creates appropriate subscriptions, which it passes to the DataRoutingPolicy of the WorkflowRouter for dispersal. At this point, the host is waiting on its task inputs.

Inputs to a task are received as data messages routed by the WorkflowRouter to the Dispatcher. The Dispatcher determines which task the data is for and passes it to the appropriate ServiceManager. The ServiceManager applies the input data to its InputFilter. If the recently received input forms a valid set of inputs to the task (possibly in combination with other previously received inputs), then the ServiceManager invokes the service. This is done by passing the name of the service, method, and input parameters to the SOAPConverter, which translates this information into a SOAP message and sends it to the SOAPFrontEnd which forwards it

to the service itself (the invoking of the service is handled by Sliver [35]). Return values are returned to the `ServiceManager` via the `SOAPConverter`. When return values are received, the `ServiceManager` applies the result to its `OutputFilter`. If a valid set of outputs is generated, it creates data messages for those values and passes it to the `Dispatcher` which in turn passes it to the `DataRoutingPolicy` of the `WorkflowRouter` for dispersal.

Knowledge Management. Along with allocation and execution of tasks, knowledge management is a critical function which enables CiAN's context awareness feature. The knowledge management system is embodied by the `KnowledgeBase` (described in detail in Chapter 3), which maintains a host's awareness of other hosts executing the workflows. Each known host has a record in the `KnowledgeBase` containing its name, IP address and port, schedule, the `RouteInfo` associated with its last known task, and a list of services offered. Each host begins with only its own information present in a local knowledge base. Hosts exchange knowledge by gossiping, as described in Section 2.4.

The *host configuration* file is an XML document describing the host on which CiAN is executing. This file contains the three pieces of information— the host name, a schedule of times when the host is unavailable, and a list of services offered — as well as the port number for the Communications subsystem. This file provides the initial population of the `KnowledgeBase`.

The Knowledge subsystem gossips knowledge whenever another host is encountered. To accommodate the gossiping behaviors, the `KnowledgeBase` registers itself as a listener on the `WorkflowRouter` and `CommunicationModule`. The `WorkflowRouter` notifies the `KnowledgeBase` whenever new knowledge is received. Similarly, the `CommunicationModule` notifies the `KnowledgeBase` when a new host is within range so that the `KnowledgeBase` can send information to it. Each piece of information is time-stamped to allow decision making in the case of conflict. Conflict may occur because of the changing connectivity of the MANET, where a host's knowledge may be out-of-date because it was out of contact with a subset of the other hosts in the workflow. The knowledge base selects the newer information in such a case.

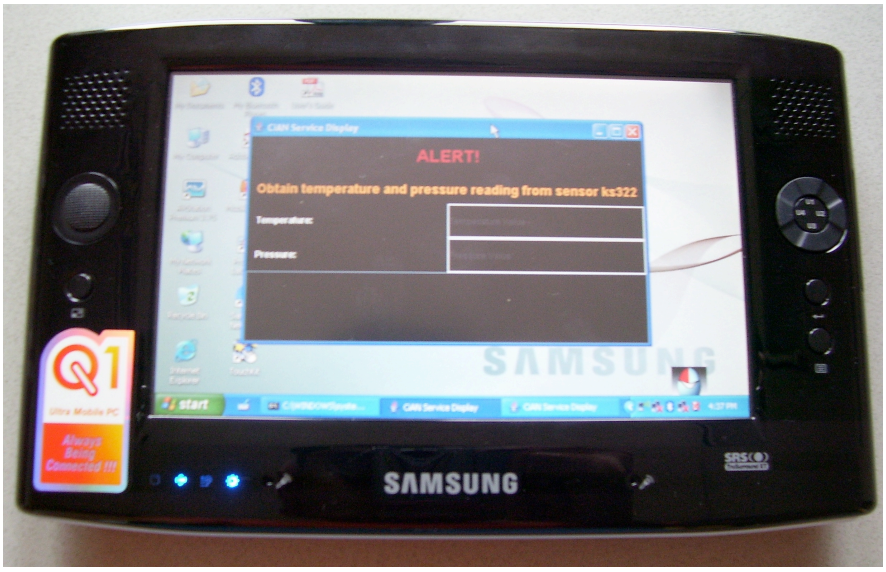

Figure 5.7: CiAN running on ultramobile PC

5.6 Results

Having presented the CiAN system architecture and implementation in the previous section, this section focuses on the effectiveness of the design and approaches. The initial portion of this section presents demonstration applications written and executed in CiAN while the latter portion covers the results of simulation experiments.

5.6.1 Demo Applications.

The capabilities of CiAN are highlighted via two demonstration applications. The first is a simple proof of concept which uses a simple workflow consisting of a sequence of tasks that involve a human being taking a reading from a specified sensor and reporting the value to the system. Figure 5.7 shows this application running on a Samsung Ultramobile PC.

The second demonstration application involves one coordinator and three worker hosts. The workflow for this application captures the process of collaboratively writing an academic paper (similar to Case 1 in Section 4.4). The application displayed

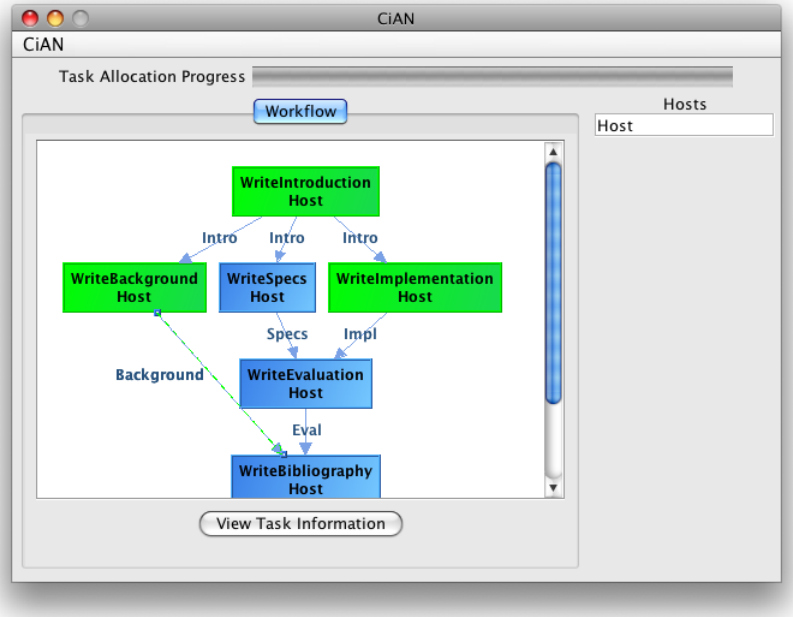

Figure 5.8: Screen shot with highlighted tasks showing allocation and completion status of the workflow

a graphical representation of the progress of the workflow (a screenshot from this application's execution is shown in Figure 5.8 and further screen captures appear in Appendix C). When worker hosts come within range of the coordinator, they receive solicitations and can submit bids (as part of the allocation process described in the next chapter). Once tasks are allocated, users receive prompts on their screens to write sections of the paper. When each task is finished, the text is transmitted to downstream tasks. At the end of the workflow, the application prints the completed paper consisting of each of the individual sections in the correct order.

These applications are just a small fraction of what CiAN is capable of. They serve as a proof of concept and a baseline from which more sophisticated applications can be built.

5.6.2 Simulation Experiments.

This subsection presents simulation results of the performance of the three schemes are used for routing data between hosts. The three protocols were simulated in the NS2 network simulator [1].

Simulation Setup.

The simulations were performed in a 200m x 200m space which represents the size of a large outdoor work area. The transmission range of the mobile devices was set to 25m using the 2-ray ground propagation model and the 802.11b MAC layer. Though the range of 802.11b can be higher than 25m, a conservative figure was chosen since experience with actual devices showed this to be the most reasonable range in the physical world. The experiments were performed with the hosts moving as per both the random walk and the random waypoint mobility model. In both cases, hosts moved with a uniform speed of 1.7 m/s which is close to human walking speed. In the random walk model, the hosts moved for a random amount of time between 1 minute and 5 minutes. In random waypoint, the host moved until it reached a waypoint. When the hosts paused, they did so randomly in the 1 minute to 5 minute range when ostensibly they were performing some task.

Randomly generated workflows were used for the tests. Several matrices in the range 5x5 to 60x60 were generated. The rows and columns of the matrices represented tasks. Using only the cells above the diagonal (to make the resultant graph directed), the cells were randomly marked to create an edge between those task pairs. The number of edges chosen was half the total possible number of edges. A single source and sink task were manually added subsequently. The tasks were then reordered to ensure they were in increasing order of position in the workflow.

Experiment 1: Overhead as a Function of Number of Hosts

The first experiment performed measured the overhead for executing the workflow while varying the number of hosts. Overhead is the time that was spent transmitting

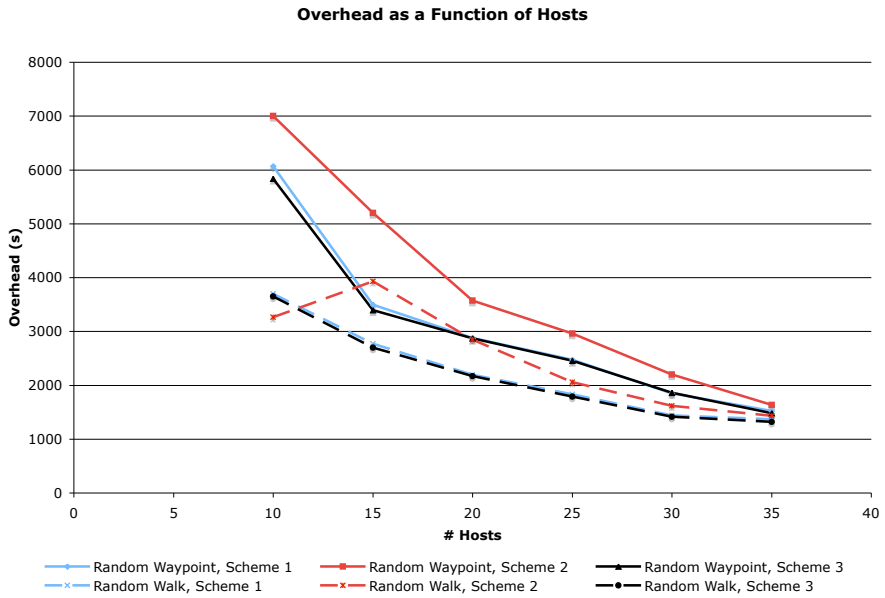

Figure 5.9: Overhead as a Function of Number of Hosts

data from one host to another or time that was spent waiting for inputs. As expected, the overhead dropped considerably with increased numbers of hosts. This is because more hosts equates to more routing options which results in faster data delivery. The results are shown in Figure 5.9. Each data point is an average of 30 runs with each run representing a different workflow. The overhead time for executing a workflow, while fairly large for small numbers of hosts was still significantly less than the time spent actually performing the tasks in the workflow and as such, considered acceptable. Also as expected, Scheme 3, which is the most flexible, performed better than the others. The kink in the trend for Scheme 2 in random walk was due to tasks being assigned to hosts in a small region, which allowed prompt communication and thereby lower overhead of execution.

Experiment 2: Overhead as a Function of Number of Tasks

The second experiment, shown in Figure 5.10, measured the effect of an increase in the number of tasks in the workflow on overhead. While overhead steadily increased for increasing numbers of tasks, the per task overhead remained fairly static when the number of tasks was below 30. However, above this number, the overhead per

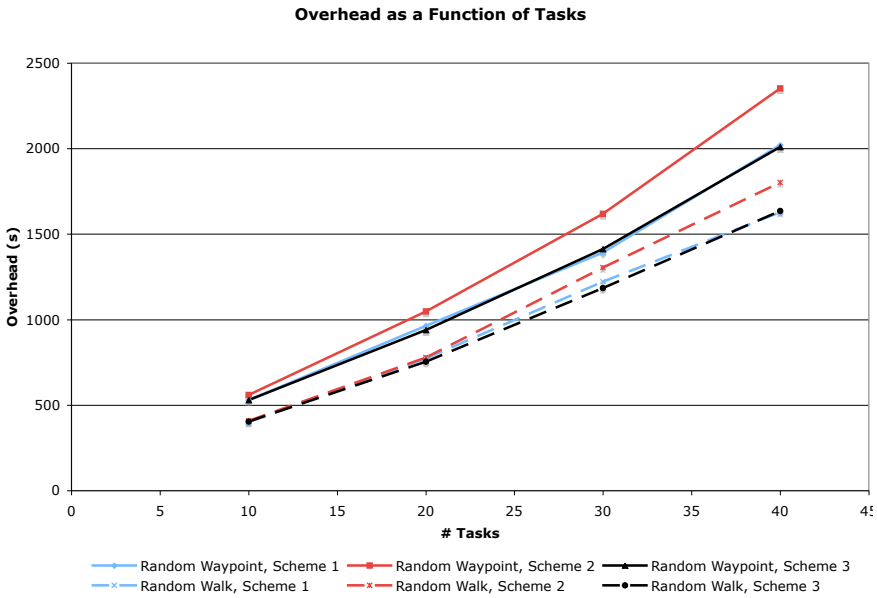

Figure 5.10: Overhead as a Function of Number of Tasks

task rose noticeably. This is attributed to the fact that this experiment was run by keeping the number of hosts fixed at 30. When the number of tasks exceeded 30, some hosts had multiple tasks assigned to them and since only one task is allowed to execute at a time, it was found that tasks that were ready in terms of having all their inputs were waiting for another task on the same host to finish, thereby increasing the overhead.

Experiment 3: Routing Scheme 3 Performance

The final experiment studied Scheme 3 in more detail. Trials were run to measure the overhead for a low number of tasks (15) and a high number of tasks (60) in both mobility models when the counter for exceeding the permissible range was varied. Each data point was an average of 20 runs using different workflows. It was found that changing the counter value reduced the overhead but not by a significant amount (shown in Figure 5.11). Given that higher counter values result in higher network traffic, keeping the counter low is desirable and this can be done without much performance penalty.

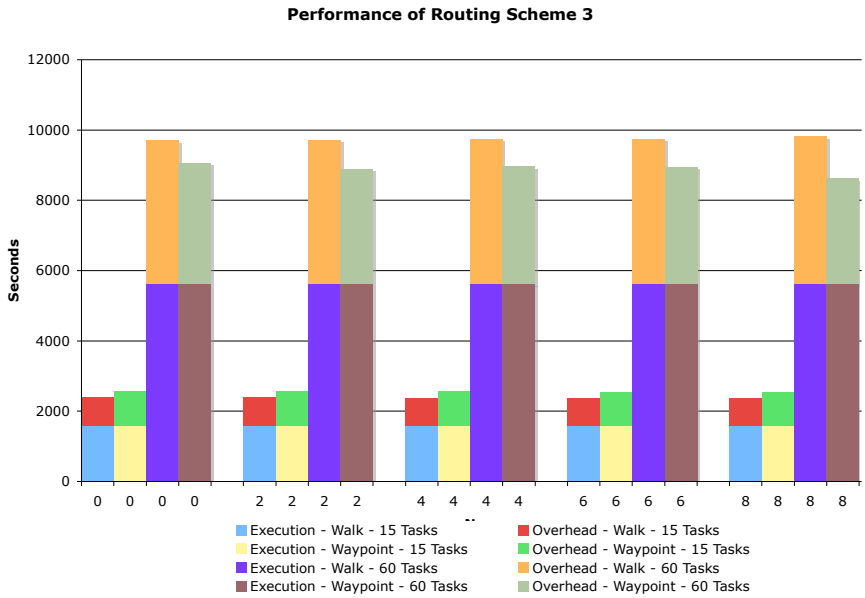

Figure 5.11: Routing Scheme 3 Performance

The results indicate that the routing protocols based on the task numbers has reasonable performance and a level of overhead that is not a hindrance to the progression of a collaboration. Most significantly in the trials, higher than 95% of workflows completed successfully despite numerous disconnections and interruptions. The small number that did fail were due to aberrant mobility patterns of one or two hosts that isolated themselves from the rest of the network and did not communicate with their peers, thereby preventing the progress of the workflow. While these results are encouraging, there remain further opportunities for optimization.

5.7 Chapter Summary

When WfMSs are ported to a MANET setting, most of the assumptions of stability made by current WfMSs are no longer valid making these systems ill-equipped to function in a MANET. This chapter addressed the problem of developing a WfMS for MANETs from the ground up. This consists of a middleware that executes workflows written in the CiAN specification. This middleware is designed to execute workflows in a completely decentralized fashion, relying on non-functional knowledge to make

decisions. Also described was a protocol for moving the result of a task from one host to another by exploiting transient communication opportunities among hosts in the MANET. The process of allocation, which was touched on only briefly in this chapter is covered in detail in the next, including algorithms to determine how tasks get allocated to hosts, which involves work in matching algorithms, as well as strategies such as an auction-based or a marketplace-based strategy.

Chapter 6

Allocation Algorithms

Allocation algorithms determine the manner in which hosts are given responsibility for executing tasks in the workflow. Allocation can be run *a priori*, i.e., before execution begins, or concurrently with the execution of the workflow on a just-in-time basis. This chapter describes two allocation algorithms, one centralized and one distributed, that allocate tasks to hosts using not only the host's capacity to perform the task as criteria but also its mobility pattern and other non-functional considerations.

6.1 Motivation

As indicated in previous chapters, the focus of this dissertation is on workflow management systems (WfMSs) that operate in dynamic settings such as a mobile ad hoc network (MANET), where hosts are physically mobile and the network topology evolves rapidly. There already exist several efforts to develop an execution engine for workflows in mobile settings such as [99, 65, 35]. However, relatively little attention has been paid to the equally-crucial process of allocation of tasks. An allocation process for mobile settings in combination with an engine for mobile workflows can enable applications that support collaboration among workers at a construction site, coordination among a team of geological surveyors in a remote region, or coordination between teams working at the scene of a toxic spill.

The approach presented in this chapter is a two step process. The first step involves the development of a centralized allocation algorithm for MANETs that takes into account the mobility of participating hosts when allocating tasks. While the use of

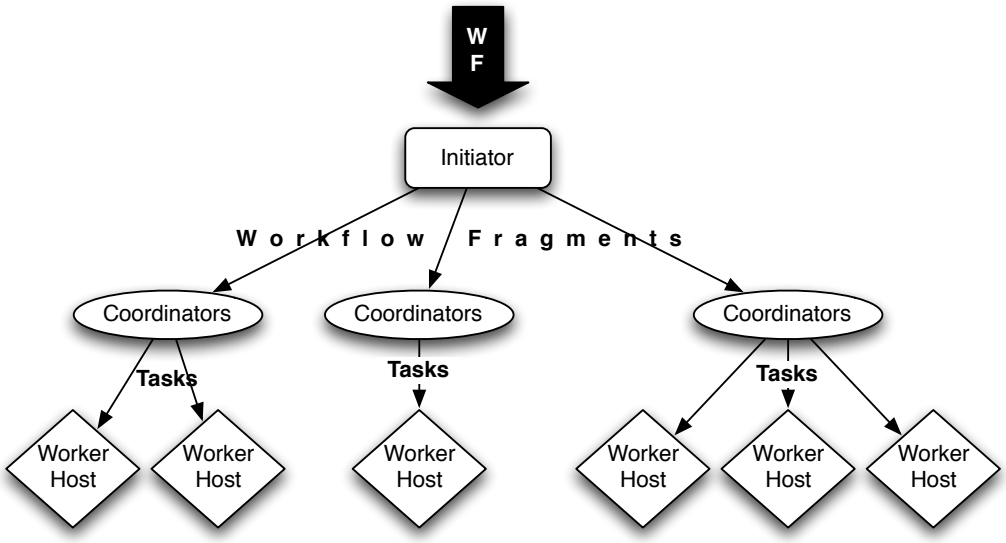

Figure 6.1: Hierarchy for the distributed allocation scheme

a centralized approach makes the algorithm simpler, the dynamics of the workflow execution environment create a new set of design challenges that are not faced when executing in a wired network. An algorithm that uses a set of heuristics to minimize the backtracking required when trying to determine a suitable allocation is described.

Since MANETs are not conducive to centralized solutions, the second step of the approach consists of taking the centralized allocation process consisting of a single coordinating entity and synchronous service calls and transforming it into a distributed and asynchronous process. A distributed allocation algorithm is described that takes a monolithic workflow and fragments it into smaller "sub-workflows" using a set of pre-defined rules. Each fragment is then assigned to a *local coordinator*, a special participant that is responsible for allocating a related subset of tasks that are assigned to it. The complete allocation hierarchy for the distributed scenario is shown in Figure 6.1. The allocation process is designed to work in a "just-in-time" manner, with tasks being allocated just before they need to be performed. The algorithm also examines constraints on motion of participants, and uses different policies for workflow partitioning and degree of workflow distribution to better accommodate future coordination efforts.

The decentralized approach is beneficial because: (1) it mitigates the problem of a single point of failure, crucial when operating in dynamic MANETs, (2) considering the motion of hosts within the algorithm reduces the chance of having allocations where the hosts cannot coordinate in the future, and (3) the just-in-time nature of the algorithm removes the requirement that all tasks be assigned *a priori*, which reduces the number of early (and potentially incorrect) allocation decisions.

6.2 Related Work

The fundamental problem is to develop a process by which tasks in a workflow are assigned to hosts that can perform them. This process not only must consider a host's capability to perform the task, but also must take into account whether the host can receive inputs from other hosts that perform the immediately preceding tasks, and successfully transmit results to hosts performing the immediately succeeding tasks.

Recently some systems have been developed to address workflows in mobile settings explicitly. A series of systems such as Exotica/FMDC [6], DOORS [86] and ToxicFarm [32], adapt workflow models for mobility by supporting workflows in the face of network disconnections. Clients in these systems hoard the needed data from a centralized server before they disconnect from the network. Clients may then continue to perform their task(s) while disconnected and the server merges any changes upon reconnection. These systems, therefore, rely on some fixed network infrastructure and assume disconnections are temporary. They also do not exploit the potential for collaboration among clients which are not connected to a central server but which may communicate directly with each other. Another approach to workflows in mobile settings has been through the use of mobile agent technology. The Agent-based Workflow Architecture (AWA) [99] consists of mobile Task Agents which can migrate to mobile devices to execute workflow tasks. The task execution may occur while the device is disconnected provided the Task Agent eventually has the opportunity to migrate back to a Workflow Agent which oversees the execution of the workflow. This agent-based approach is more flexible and appropriate for dynamic settings, but its single point of failure (the Workflow Agent) makes it undesirable for MANETs.

In MANETs however, neither is there the opportunity to have a centralized management architecture, nor are the participants always accessible due to wireless links that might break frequently due to host mobility. Hence the allocation process is much more opportunistic in nature, allocating tasks *a priori* whenever hosts are within communication range, which is a different approach from those in use today. Also, ensuring that a host can receive inputs and successfully transmit outputs becomes complex due to the mobility of participating hosts.

In a sense, the task allocation problem is similar to the Job Shop Scheduling (JSS) problem where a set of jobs is scheduled on a set of machines such that no machine executes more than one job at a time and the total duration for executing the jobs is minimized. In this work, the tasks and participants are analogous to the jobs and machines respectively.

The difference lies in the fact that the primary objective function here is to maximize allocation. Minimizing the time required to complete the jobs is only a secondary objective. Also, in addition to jobs being admitted during the scheduling process (the entire workflow may not available for scheduling up front), the possibility of additional machines being admitted during the scheduling process must also be accommodated. Finally, the approach must also take into consideration the constraints imposed by the physical mobility of hosts and the fact that the machines are heterogeneous (all jobs cannot be scheduled on all machines).

In [27], the authors describe a heuristic-based method for solving the basic JSS problem while [23] describes a genetic approach to solving the same problem. More pertinent to this work is [58], which considers the JSS problem with availability constraints, i.e., where the set of available machines on which to schedule jobs changes over time. This is analogous to the reachability of hosts changing over time in a MANET. Another closely related piece of work is reactive JSS [53] where the schedule is not computed *a priori* but rather over a period of time. More recently, researchers have used neural nets to solve the JSS problem [117].

Another related area is robot task scheduling. In [30], the authors propose a taxonomy of multi-robot task assignment problems. The work presented in this chapter is closest to the extended time assignment variants of the problem proposed therein. Solutions

to this problem involve using a market-based economic model [118] and an auction-based approach [29] that uses concept of task utility and fitness of a robot to perform a task to make allocations. Incorporation of spatiotemporal considerations including the formation of organizations and a reward scheme is described in [18] while a scheme for fault tolerant coalition formation is described in [81]. Similar approaches have been used to allocate resources in wireless sensor networks [55].

6.3 Algorithm

6.3.1 Centralized Approach

This subsection describes the centralized allocation algorithm. This algorithm is different from those employed by workflow systems in wired settings in the following ways: 1) the allocation of tasks is done *a priori* and in a batch (all tasks are allocated before the workflow execution begins) as opposed to in an on-demand fashion at runtime, 2) hosts are evaluated on the basis of their functional capabilities *as well as* their spatiotemporal behavior, and 3) the allocation process is partitioned into sub-problems with backtracking capabilities built in. It should be noted that in mobile environments, the allocation process assumes greater significance since a poor allocation may not necessarily be able to be corrected at a later stage and can result in the workflow not executing to completion due to non-functional circumstances such as situations where a host has completed a task but cannot communicate the completion to the next host in the workflow because that host is not reachable.

Before allocation of tasks to hosts can begin, any relevant constraints that ensure that an undesirable allocation is not computed must be taken into account. Two types of constraints are possible– *host allocation constraints* prevent certain hosts from being allocated to a task, or require a particular host to be allocated to a task, e.g., that a host must be allocated to both task X and task Y, or that task X cannot be allocated to the same host as task Y, etc. Such constraints may be specified as a supplement to the workflow specification. *Spatiotemporal constraints* prevent allocations that are in conflict in the spatiotemporal domain, e.g., a host should not be allocated to two tasks whose start and end times overlap, a host should not be allocated to two

tasks if they are separated by time t and distance d, if the host's maximum speed is lower than $\frac{d}{t}$, and two different hosts should not be allocated to two sequential tasks if the host executing the second task cannot receive the results of the first task (either directly or via a disconnected route) before the second task begins. The existence of spatiotemporal constraints for tasks and hosts can be determined by calls to the `is_spatiotemporal_constraint` and `is_meeting_possible` operations on the knowledge base described in Chapter 3. The knowledge base can provide information on such constraints because it contains the motion information of all relevant hosts (due to it having been gossipped by the other hosts which are all assumed to be co-located initially). Spatiotemporal constraints are especially important because they abstract the effects of mobility and represent them as a simple constraint set to the allocation algorithm.

Constraints are represented as 3-tuples of the form $< t_1, A, t_2 >$ which indicates that host A cannot be allocated to task t_2 if it has been allocated to task t_1. Note that the algorithm is agnostic to the cause of the constraint which may be host-driven or spatiotemporally driven. For simplicity, it is assumed that constraints are symmetric. Once all constraints are established, the data structures that represent the initial state of the allocation algorithm are built. In this phase, a table is created for every task in the workflow as shown in Figure 6.2. The first column in each row represents a host that has the *functional* capability to perform the task and is not subject to a host constraint associated with that task. The matching of the functional capability of a host with the requirements of a task can be easily computed [51] if hosts' capabilities and tasks requirements are expressed using a uniform ontology such as OWL-S [57]. The second column in each row is a list of tasks which cannot be allocated to the host (in the first column) if that host were to be allocated the current task. This information can be obtained from the constraints assembled previously– for each constraint $< t_1, A, t_2 >$, the table is checked for task t_1 to see if it has a row for host A. If it does, then t_2 is added to the corresponding list in the second column.

task 1	
host A	2, 3
host B	3, 4
host D	3, 4, 5

task 2	
host A	1, 4
host B	3, 4
host C	4, 5

Figure 6.2: Examples of constraint tables

The tables for each task provide two pieces of information– the list of hosts to which that task can be allocated, and the list of future allocations made impossible by that decision, e.g., according to the first table in Figure 6.2, task 1 can be allocated to hosts A, B, or D. Allocating task 1 to host A however, makes it impossible to allocate tasks 2 or 3 to host A.

Allocation Algorithm.

The tables assembled in the previous step form the input to the core allocation algorithm, the pseudo-code for which is shown in Figure 6.3. The algorithm first sorts the tables in ascending order of the number of rows in the table. Then, within each table, it sorts the rows in ascending order of the number of elements in the list in the second column. Thus the tables are sorted according to the number of hosts that can perform a task, and the rows are sorted according to the number of allocation conflicts the allocation decision causes.

The algorithm begins with the task represented by the first table. It selects the first host in that table. If this host has no conflicts, then it can allocate that host without any conflicts, i.e., this host can be allocated without affecting *any* other allocation decisions. Hence, the host is allocated to that task and the algorithm proceeds with the next task. If the first host (call it host A) has at least one conflicting task, then by allocating the task to the first host in the table, some future allocations are made impossible. To establish whether this decision is the correct one, the algorithm tries recursively to resolve the conflicts. For this, it creates a stack to keep track of its allocation decisions as shown in Figure 6.4. It marks host A's row in the table, and pushes a token onto the stack reflecting this marking. Next, it collects the list of conflicting tasks from the table. For each conflicting task, its grays out the row with Host A in the corresponding task's table, and pushes onto the stack a marker that represents this change. When it later visits these tasks, it will disregard all the rows that have been grayed out, since they reflect decisions that would violate a constraint. This process continues until all conflicting tasks have been recursively allocated at which point it returns to the original list of tasks and continues allocating them sequentially as before.

```
boolean heuristicAllocatetask(tasks, A, allocation)
    for each row (G, conflicts) in A, ordered by |conflicts|
        if |conflicts| = 0
            allocation := allocation ∪ (A, G)
            return true

        myToken := new AllocationToken(A, G)
        push(stack, myToken)
        allocation := allocation ∪ (A, G)

        for each C in conflicts
            if C ∉ allocation
                push(stack, new GreyRowToken(C, G))
                disableRow(C.table, G)

        for each C in conflicts, ordered by |conflicts|
            if C ∉ allocation
                if not heuristicAllocatetask(tasks, C, allocation)
                    do
                        token := pop(stack)
                        undo(token)
                    until token = myToken

                    push(stack, new GreyRowToken(A, G))
                    next row

        return true
    return false

map enhancedAllocate(tasks, hosts)
    allocation := ∅
    createConstraintTables(tasks, hosts)

    for each A in tasks, ordered by |A.table|
        if A ∉ allocation
            heuristicAllocatetask(tasks, allocation, A)
    return allocation
```

Figure 6.3: Psuedo-code for heuristic allocation algorithm

At some point, the algorithm may encounter a task with no capable hosts left (due to them having been grayed out as an effect of previous allocation decisions). This means that one of the earlier decisions was undesirable, and that it must roll its state back to that decision point. It does this by popping elements off the stack, undoing the changes that they represent, until it reaches a change to a table that marked one of at least two remaining rows. This indicates a place where it made a decision that may have been incorrect. It un-marks the host chosen at this point, and grays out its row so that it doesn't try that host again (it also pushes a token onto the stack for the row that has just been grayed out). Finally, the algorithm attempts to re-allocate the task to the next un-grayed host in the table.

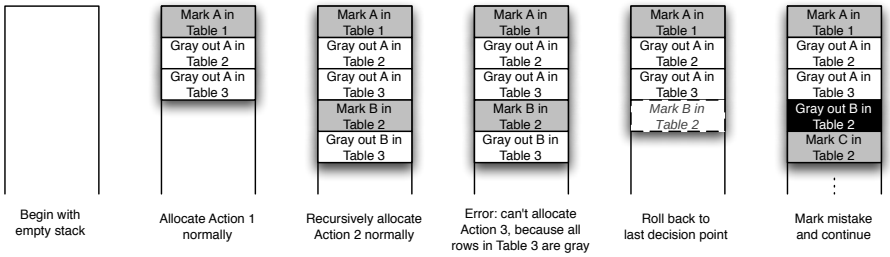

Figure 6.4: Allocation stack for tracking and rolling back changes

This algorithm has two key features. First, rather than allocating tasks in an arbitrary order, it first allocates the tasks that are hardest to satisfy and allocates them to hosts that will cause the fewest conflicts later. This reduces the amount of backtracking that the algorithm must do, since it will first consider the paths that are least likely to cause irresolvable conflicts. Second, the algorithm recurses through hosts' conflict lists, effectively dividing the allocation of the workflow into sub-problems. Due to this recursive process, it is guaranteed first to consider the entire "conflict closure" of a task, i.e., all the tasks that recursively conflict with it. Since by definition tasks in one closure cannot conflict with tasks in another closure, they are allocated completely independently of each other. So, once a closure has been fully allocated, the algorithm will never revisit any of the tasks in it, which greatly reduces the scope of backtracking.

Note that the algorithm does not consider the actual data flow when computing a well-formed allocation. This has two implications. First, the constraint that two hosts must "meet up" before exchanging data becomes more complex to describe when one host must receive results from multiple predecessors. This constraint can be simplified by requiring that all tasks corresponding to nodes that join to a common node in the graph must take place in the same physical location. This behavior can be enforced by adding "move to a common location" tasks to all the paths immediately before the join point. Second, the allocation is conservative: it is assumed that all tasks in the workflow will be executed, even though the workflow may split into multiple, mutually-exclusive paths. Thus, valid allocations may exist which do not execute all tasks, and which the algorithm will not find. This shortcoming can be worked around by enumerating all possible traces through the workflow and attempting to allocate each trace individually until one feasible allocation is found. As shown in Section

6.5, the cost of running the algorithm is low enough to make this approach feasible. Nevertheless, future work may consider ways to incorporate data flow information into the algorithm's decisions.

Before concluding this section, a brief discussion of the complexity of this algorithm is presented. Since the algorithm involves backtracking, in theory its worst case complexity is exponential. However, in practical use this is rarely likely to occur. Consider a workflow of 'n' tasks which must be allocated to 'h' hosts. Theoretically, there are 'h' options for each of the 'n' tasks leading to a worst case complexity of n^h. However, saying that there are 'h' options for each task, is in effect saying that all 'h' hosts are capable of doing any of the 'n' tasks. If this were true however, it would be much easier to find suitable hosts, and the amount of backtracking would be reduced significantly, reducing the overall complexity. Another possibility is that the number of hosts that are able to do a task is significantly less than 'h'. Therefore 'a', the average number of options $<<$ 'h' which means that $n^a << n^h$. This complexity assumes a brute force approach of trying all options in a random order as in the naïve algorithm.

The algorithm presented here does not allocate tasks in a random order. Rather a task that has the least number of possible hosts to service it is allocated first. In the process of allocating this task, other opportunities for hosts to perform a task (due to conflicts) are cancelled out which brings down the value of 'a' as the algorithm progresses, making it more efficient at the tail end. Finally, once a task is allocated, it is followed by the allocation of its conflict closure. This ensures that the backtracking is over a subset of the tasks rather than the entire workflow because a task and its conflict closure by definition do not cause conflicts with other tasks and hence once allocated, need not be revisited as part of any backtracking. As such the algorithm makes it possible to allocate tasks to hosts while taking into account their mobility patterns at a lesser computational cost than a naïve approach.

6.3.2 Distributed Approach

In contrast to the centralized approach described in the previous subsection which performs *a priori* allocation under the assumption of initial co-location of hosts, this

subsection describes a distributed approach which requires initial co-location of only the initiator of the workflow and a select group of hosts called *coordinators* that execute the actual allocation process in a decentralized manner. The approach is divided into three phases: 1) pre-processing steps, which occur prior to the actual allocation process, 2) the core allocation process that is agnostic to whether it is run in a centralized or distributed fashion, and 3) additional resources that distribute the core process and manage the mobility of participating hosts. Each phase is described in turn.

Pre-processing Steps. The first pre-processing step is to assign each task a time-dependent utility value that represents how critical it is that the task be allocated. For a task T at time t, its utility $U_T(t) = 1/(E_T - t)$ where E_T is the earliest starting time for task T, and t is the current time. Thus, the further in the future a task's start time is, the lower it's utility to the progression of the workflow *at the current time*. The next step is to fragment the monolithic workflow into k pieces so that tasks can be allocated by k coordinators in a distributed fashion. Two fragmentation techniques are used:

k-Minimum Cut. Using a graph traversal algorithm, tasks are sorted into buckets according to their depth in the graph. The k-minimum cut approach considers the combined size of adjacent buckets in turn. Cuts are made between the k bucket pairs that have the lowest combined value. The exception to this rule is that cuts are note made that would result in a fragment having lower than f tasks, where f is a user-defined parameter with a value less than N/k where N is the total number of tasks in the workflow. In such a case, the next higher cutting point is chosen.

Geographic Cut. The area in which the workflow is to be executed is divided into k equal sized zones. Using a graph traversal algorithm, the task locations are examined and the task specifications are put into buckets that correspond to zones containing the tasks' locations. Thus, the tasks in any fragment are geographically related, i.e., they are in a subset of the total area.

The k-Minimum Cut technique keeps blocks of contiguous tasks under the responsibility of one coordinator, which is useful when recovering from localized errors (not covered in this dissertation). The geographic cut technique allows geographically related tasks to be handled by one coordinator. Since each coordinator is responsible

Given an host with capabilities C, schedule SC, and capable of a maximum velocity $maxV$ that is analyzing a solicitation list S:

```
AnalyzeSolicitations(C, SC, maxV, S)
    B ← ⊥
    for each s ∈ S
        if capabilities[s] ⊆ C then
            if Available(SC, start[s], deadline[s]) then
                precT ← GetPrecedingTask(SC, start[s])
                succT ← GetSucceedingTask(SC, deadline[s])
                precC ← GetCoordinator(precT)
                succC ← GetCoordinator(succT)
                precAvail ← deadline[precT] + |location[precT] − location[precC]|/maxV
                succAvail ← deadline[s] + |location[succC] − location[s]|/maxV
                precV ← (|location[precC] − location[s]|)/(start[s] − precAvail)
                succV ← (|location[s] − location[succC]|)/ (start[succT] − succAvail)
                if precV ≤ maxV and succV ≤ maxV then
                    capFrac ← |capabilities[s]|/|C|
                    bid ← {capFrac, precV, succV, maxV,
                            GetDeadline(SC, succT)}
                    Insert(B, bid)
    Transmit(B)
```

Figure 6.5: Procedure for bid formulation by worker hosts

for a specific sub-area, the geographic cut allows correlation between the location of the coordinator and the tasks they are allocating.

Core Allocation Process. When a coordinator receives a fragment of the workflow, it places the tasks in that fragment into a *task list* sorted in descending order according to their utility. The time parameter t to the utility function of the tasks is the start time of the workflow. Each task is assigned the status `NO_ATTENTION`. The coordinator then runs the allocation process continuously until all tasks are allocated. The allocation process is split into two phases: (1) distributing solicitations to perform tasks to worker hosts and generating bids and (2) allocating a task provisionally to a host and then revisiting the allocation decisions over time. Each step is described in detail below.

Distributing Solicitations and Generating Bids. For each task, the coordinator formulates a solicitation which is a 6-tuple of the form <*String:taskName, List:capabilities, Location:taskLocation, Time:duration, Time:start, Time:deadLine*>. Each of the six pieces of information in the solicitation can be obtained from the task specification [97]. When a worker host comes within communication range of the coordinator, the solicitations are sent to that worker host (the detection of the presence

of a worker host is done by the `CommunicationModule` using the procedure described in 5.5). When a worker host receives the list of solicitations from the coordinator, it analyzes them to determine whether it is suited to perform any of the tasks advertised using the procedure shown in Figure 6.5. If so, it submits bids for the tasks it can perform.

For each solicitation, the algorithm checks whether the capabilities required by the task is a subset of the host's capabilities. If so, it checks that host's schedule to ensure that the host does not have any previously scheduled commitments at the time that the task described in the solicitation needs to be performed. This is done using the AVAILABLE function on the host's schedule which returns a boolean value. If this check is successful, then the host is qualified *and* available to do the task. Finally, the travel time is factored in. For this, the details of the tasks that would immediately precede and succeed the task under consideration (were it to be assigned to this host) are obtained. This is done using the GETPRECEDINGTASK and GETSUCCEEDINGTASK functions respectively. The velocity at which the host would need to travel from the coordinator to which the results of the preceding task need to be delivered to the location of the task under consideration, and then from the location of the coordinator to which the current tasks's results must be submitted to the location of the succeeding task is computed. The reason the coordinator locations are used rather than the task locations is due to the conservative planning done by the algorithm (described later in this section). If both the preceding and succeeding velocities are lower than the maximum velocity capability of the host, then it is eligible to submit a bid for that task.

Before the submission, it calculates the fraction of its capabilities that it will use in performing the tasks. It then creates a bid, which is a 5-tuple of the form *<double:capabilityFraction, double:precedingVelocity, double:succeedingVelocity, double:maxVelocity, Time:deadline>*. The deadline is computed by the GETDEADLINE function of the host's schedule, which determines the latest time at which the host must leave the current location so as to have sufficient time to travel to the designated location of any previously made commitment in time. If the host has no other commitments, this deadline value is infinity. This bid information is then added to a set. Once all the solicitations are considered, the TRANSMIT function sends all the bids to the coordinator.

Provisional Allocations and Re-allocations. During the allocation process, two things happen in parallel– the allocation of tasks to hosts and the submission of bids by hosts. The process of bid submission is covered first followed by the allocation process. When a coordinator receives a bid, it is placed in a *bid list* that corresponds to the task that it was submitted for. The corresponding task is marked as `NEEDS_ATTENTION` in the task list.

The bids in each bid list are sorted in descending order of the capability fraction of the bid. The capability fraction indicates whether a host is specialized for the task or not. A "jack of all trades" would use fewer of its capabilities for a task than an host that is specialized for the task in question. Sorting the tasks in this manner biases the algorithm to choose more specialized hosts before choosing hosts with broader capabilities, the rationale being that it is desirable to have hosts with broader capabilities available for tasks which may not have specialized hosts. To break ties between bids, the average of ratios of the preceding and succeeding velocity to the maximum velocity is used, which indicates how good a fit the task is in the schedule of the host. Higher ratios indicate a more constrained time slot and therefore a better fit in the schedule.

When bids are initially inserted into the lists, they are marked as `NON_CONFLICTING` indicating that the host that submitted the bid is not provisionally allocated to a conflicting task. In the future, as hosts are provisionally allocated for a task, the other bids belonging to the host that are in conflict (due to spatiotemporal constraints or schedule constraints) with that particular provisional allocation are marked as `ARE_CONFLICTING`.

The allocation process iterates over the task list every p seconds and moves tasks to the *outstanding task queue* if they are either (1) marked `NEEDS_ATTENTION` due to a new bid being submitted or (2) their earliest start time or host imposed deadline is within $minT$ (a parameter to the algorithm) seconds of the current time. The outstanding task queue is processed in parallel as follows (see Figure 6.6): The coordinator removes the first task from the outstanding tasks queue and looks at the bid list for that task. The first bid that is marked as `NON_CONFLICTING` in the bid list is the best qualified host that has no other conflicts. This host is provisionally allocated to perform the task, and all other bids submitted by the host that conflict with this allocation are

```
Given a set of sorted bid lists with bids B, an outstanding task queue O,
a task list T, a minimum threshold minT, and re-evaluation period of n

ALLOCATE(O, B, T, minT, n)
    while O ≠ ⊥ do
        WAITONEMPTY(O)
        t ← REMOVEFIRST(O)
        bid ← REMOVEFIRSTNONCONFLICT(B, t)
        if bid ≠ ⊥ then
            if alloc[t] = ⊥ then
                alloc[t] ← bid
                COLORASCONFLICT(B, host[bid])
            else
                COLORASCONFLICT(B, host[MAX(alloc[t], bid)])
                COLORASNONCONFLICT(B, host[MIN(alloc[t], bid)])
                alloc[t] ← MAX(alloc[t], bid)
        if ((start[t]− GETSYSTEMTIME() ≤ minT or alloc[t].
            deadline ≤ GETSYSTEMTIME()) and alloc[t] ≠ ⊥) then
            NOTIFYHOST(alloc[t])
        else
            color[t] ← NO_ATTENTION
            INSERT(T, t)
```

Figure 6.6: Distributed allocation algorithm

marked `ARE_CONFLICTING`. If the task under consideration already has a provisional allocation, the algorithm chooses the better bid using the same criteria that is used to rank bids. If there is a change in the provisional allocation, the conflicting bids are updated accordingly with the new bid's conflicts being marked as `ARE_CONFLICTING` and the old bid's conflicts reinstated to `NON_CONFLICTING` status. At this point, the coordinator checks whether the current time is within some $minT$ of the earliest starting time of the task or whether the deadline set by the host submitting the winning bid has arrived. If either of these is the case, then it makes the allocation final by notifying the host of its newly allocated task. If the current time does not fall within the $minT$ of the earliest start time or the deadline has not approached, the coordinator marks the task as `NO_ATTENTION`, indicating that it does not need further attention at this time and re-inserts it into the task list. If no bid is available at the time of the initial evaluation, then the coordinator re-inserts the task into the task queue.

The above scheme ensures that tasks are considered and re-considered by the allocation process every time a new bid comes in, if their start time is approaching, or if the deadline imposed by the host with the winning bid is approaching. This ensures that the allocation decision is made as late as possible and is the best option available

at the time. Note however, that there is a greedy element to the approach which is that if a host submits a bid and the deadline imposed by that host is approaching, then we commit to that host rather than risk waiting for another, potentially better host to come along. The complexity of the algorithm is shown in Figure 6.7.

Accommodating Physical Mobility. To accommodate mobility, it must be considered when allocating tasks to hosts because mobility affects the ability to transfer results after it has finished the task to the host(s) that have been assigned subsequent tasks in the workflow. The preferred method is to transfer the results directly to the intended recipient via a publish-subscribe based protocol described in Section 5.4. However, this may not always be possible due to the lack of a disconnected route [37] (a spatiotemporal series of store and forward hops) between the two hosts in question. In such cases, the source host can attempt to transmit the results to the coordinator using the same publish-subscribe based protocol. If this too is not possible, the source host must physically return to the coordinator and transfer results. The coordinator, once it receives results, stores them until the recipient of those results is within range and then transmits the results to that host. Since the existence of disconnected routes between hosts cannot be known *a priori* without knowing their motion profile [96], in the allocation planning always assumes that the worst case scenario will occur, i.e., the host will need to return to the coordinator. This additional travel during the allocation process is factored in to the bid submission process (shown in Figure 6.5) to ensure that even while hosts are physically mobile, there is a reliable way for them to exchange data.

Distributing the Allocation Process. The transition from a centralized allocation algorithm to a distributed one is made possible by using multiple coordinators to allocate a workflow. This transition requires two key changes: (1) splitting the workflow into discrete pieces and (2) modifying the behavior of the coordinators. These are described in detail below.

Dividing the Workflow among Coordinators. By definition, the workflow for any activity is a monolithic entity. It is assumed that initially, the *initiator* has the specification of this monolithic workflow. The initiator is responsible for fragmenting the workflow using either the k-min-cut or the geographic cut approach. If the k-min-cut approach is used, fragments are assigned randomly to the coordinators. If

the geographic cut is used, fragments are assigned such that the coordinators are responsible for allocating tasks in the geographic area in which they operate. Each coordinator, in addition to receiving a fragment of the workflow also receives a table of tasks in the workflow that are not in the fragment allotted to it, along with the name of the coordinator responsible for assigning each of those tasks. Once a coordinator receives its fragment of the workflow, it can immediately begin executing the core allocation process as described earlier.

Changes in Coordinator Behavior. From the coordinator's point of view, the transition from a single coordinator to multiple coordinators requires only two relatively minor changes. The first relates to the bid submission process. If a coordinator has a bid from another host that is better, it immediately rejects the host's bid, which allows it to leave the locality (described in the next paragraph). Second, when calculating the travel time of hosts to return results to the coordinator, the coordinator now has to be aware whether the subsequent task that the results are destined for will be allocated by itself or another coordinator. If the task is allocated by the same coordinator, there is no change from the case of the single coordinator. If the task is allocated by a different coordinator, the destination of the travel is changed to be the coordinator that is allocating the subsequent task. From the worker host's perspective, factoring in which coordinator must be travelled to is handled as part of the bid submission process shown in Figure 6.5.

Controlling Worker Host Motion. In a mobile setting, especially that of a MANET where hosts are physically mobile, the effectiveness of the allocation process depends heavily on host motion. If worker hosts move in an adversarial manner, they can ensure that very few tasks get allocated. Even if worker hosts are not adversarial, e.g., if the worker hosts exhibit random movement, they can still affect the performance of the allocation process. Simply put, when worker hosts move randomly, it becomes a matter of chance whether they come within range of the coordinator and have tasks allocated to them. It was found that by imposing minor constraints on worker host behavior, it is possible to make the system more consistent and reliable, especially when host and task densities are low.

The constraints imposed on worker host motion are as follows. When a worker host has time slots in its schedule that are free, it gravitates towards the *nearest* coordinator. Upon coming within range of a coordinator, the worker host checks the solicitations as before. However, if there are no tasks for which the worker host is suited, it immediately leaves and goes to the next closest coordinator that it has not yet visited. If it does submit bids, the coordinator checks whether the bid is better than the current best choice. If not, the coordinator notifies the worker host immediately of the failed bid. This ensures that a worker host is not waiting at a coordinator while other coordinators have tasks that it could perform. These constraints are non-intrusive and in fact replicate the command and control structure of many collaborative activities to which this work is targeted, i.e., it is analogous to a worker returning to find a supervisor to be assigned additional tasks. As such, such constraints are deemed to be reasonable. The performance of the approach with and without these constraints is discussed in Section 6.5.

When the system is operating in a fully distributed manner, coordinators are usually stationary and responsible for a fragment of the workflow. Worker hosts gravitate towards a coordinator and remain there if there is a task that needs to be allocated that it can perform competently. If the task is allocated to the worker host, it leaves, performs the task and then returns to the coordinator. If the task is not allocated, the worker host may move to another coordinator in search of tasks. The process of searching for tasks and then performing them goes on until all the tasks are completed. At this point worker hosts have no more work to do and they gather around the coordinators. Coordinators then transmit a termination signal that indicates that its portion of the workflow has been completed and shuts down. Eventually all coordinators shut down, indicating to the worker hosts that the workflow is complete.

Guarantees. In MANETs, it is difficult to provide guarantees due to the unpredictable motion of hosts. However, when some constraints are imposed on how worker hosts can move, as described above, it becomes easier to offer guarantees. The allocation scheme does not guarantee to allocate all tasks on time. However, it does guarantee that all tasks will eventually be allocated as long as there are worker hosts available with the requisite services. It is possible to make this guarantee because the motion constraints require each worker host to visit each coordinator looking for work. Thus, if a coordinator has a task requiring a service that is on a particular

Step	Comp-lexity	Total
PREPROCESSING (on Initiator)		
Determining Parallel Tasks	O(n)	
Fragmenting the graph		
- Graph traversal to make buckets	O(n+ e)	**O(n+e)**
CORE ALGORITHM (on Coordinator)		
Sorting Tasks by Utility with BubbleSort		
- Complexity is n^2 only when all tasks are in parallel with every other task (pathological case)	$O(n^2)$	
Formulating solicitations	O(n)	
Inserting bids into sorted buckets	O(hn)	
- Assume each host submits a bid for each task (pathological case)		
Allocation and re-allocation		
- Each task is evaluated initially and re-evaluated every time a host submits a bid	O(hn)	**$O(hn^2)$**
PROCESSING BIDS (on Worker Hosts)		
Analyzing solicitations + submitting bids	O(n)	
- One solicitation per task in workflow		
Compute travel time for each solicitation	O(n)	**O(n)**

Figure 6.7: Computational complexity of the allocation process

worker host, it is guaranteed to encounter that worker host eventually. The eventually semantics of the allocation scheme is sufficient for workflows where the tasks are required to be executed according to some partial order. In fact, most workflows today impose this requirement. Where the approach is not as efficient is when there are strict deadlines for each task. In such cases, the delayed allocation affects the remainder of the workflow, which is undesirable. However, as shown in Section 6.5, the scheme performs fairly well in allocating tasks on time even with strict timing constraints.

Analysis of computational complexity.

The allocation process described in this section consists of several small algorithms that run in concert to make the task allocations to hosts. To make the analysis of the computational complexity of this process more simple, each of the individual algorithms are analyzed separately and subsequently combined to yield a combined complexity for the process. The summary of the various complexities appears in Figure 6.7.

Pre-processing Stage. The first stage of the allocation process is the pre-processing stage in which the workflow is split into multiple smaller pieces using the k-min or geographic cut approach. For the k-min approach the first step is to determine the degree of parallelization. To do this, the algorithm starts at the root of the workflow graph, and moves through it visiting each node exactly once (a flag ensures that if two nodes have a common child, the child is not checked twice). Hence this step has a worst case complexity of $O(n)$ where n is the number of tasks. Next, the tasks are organized into buckets. For k-min, the buckets correspond to the depth in the workflow. This is done with a standard traversal algorithm that has complexity $O(n+e)$ where e is the number of edges in the graph. Finally, the minimum cuts are determined by iterating over the buckets organized by depth which in the worst case will be $O(n)$ for the case where there is a bucket corresponding to each and every depth in the graph. Thus the overall complexity for the preprocessing using k-min is $O(n+e)$. If geographic cut is used, it eliminates the first traversal to determine the degree of parallelization but the overall worst case complexity for the bucketing process remains the same, i.e., $O(n+e)$.

Core Algorithm. The next stage is the core algorithm execution. Here, the first step is to sort tasks by utility which in the worst case can take $O(n^2)$ time using a BubbleSort. This worst case is in fact a pathological workflow where every task is in parallel with every other. In a typical workflow, sequential tasks are already in utility order. The only tasks that may need re-arranging are the ones that occur in parallel, which in most cases is much lower than the total number of tasks n. The formulation of solicitations is constant time per task and hence has complexity $O(n)$. When hosts submit bids, they are inserted into the correct bucket containing bids for the task (the buckets themselves are sorted). In the worst case, searching for the bucket takes $O(n)$ time (one bucket is maintained per task) and inserting it takes a worst case $O(h)$ time where h is the number of hosts (each host submits a bid for that task). This yields a total complexity of $O(hn)$. For the actual allocation, each task is evaluated every time a new bid comes in or if its start time is approaching. Each task in the worst case therefore gets evaluated $h+1$ times where h is the number of hosts. In addition, each time a task is allocated, the algorithm must gray out other bids which may take a worst case $O(n)$ time (if that host had bid on all tasks in the workflow). Hence, the combined complexity of this stage is $O(hn^2)$. It should be noted however, that these complexities come from worst case pathological cases. All

hosts would rarely bid to do a particular task because it is unlikely all of them would be qualified *and* available to do it. A host would not bid on all tasks because it would likely not have all the necessary qualifications, and also because all tasks would not be allocated by a single coordinator. Hence, the practical values of h and n are much smaller than their worst case values. Note however, that if a task is not allocated on time, it will be re-evaluated every p seconds after its start time has passed. If the task is not allocated soon after, the number of re-evaluations could dominate the overall complexity.

Bid Submission. On the worker hosts, the analysis of each solicitation and the computation of travel can be achieved in constant time with the use of proper data structures and multiple indexes and hence the analysis of solicitations takes $O(n)$ time. In the bid collection, the analysis is done based on the assumption that every host will submit a bid for every task, an unlikely thing to happen given hosts may not have the capabilities to do all tasks.

It should be noted that each of the stages described above occur on *separate* hosts. Hence, the initiator and worker hosts have a complexity of $O(n+e)$ and $O(n)$ respectively. The bottleneck is the coordinator which has a complexity of $O(hn)$ over the life of the allocation process. However, since this process is spread out over a long duration (corresponding to the execution of the workflow), this bottleneck is purely theoretical and does not affect practical executions of the system.

6.4 Implementation

Both the centralized and the distributed algorithms have been implemented as plugins to the CiAN engine. This section presents the implementation of the centralized algorithm followed by a description of the distributed algorithm.

Centralized Allocation Algorithm

CiAN is designed to accommodate both centralized and distributed algorithms. When central planning is used, CiAN designates a special host as the *initiator* which runs the centralized planning software. Figure 6.8 shows the architecture of this software.

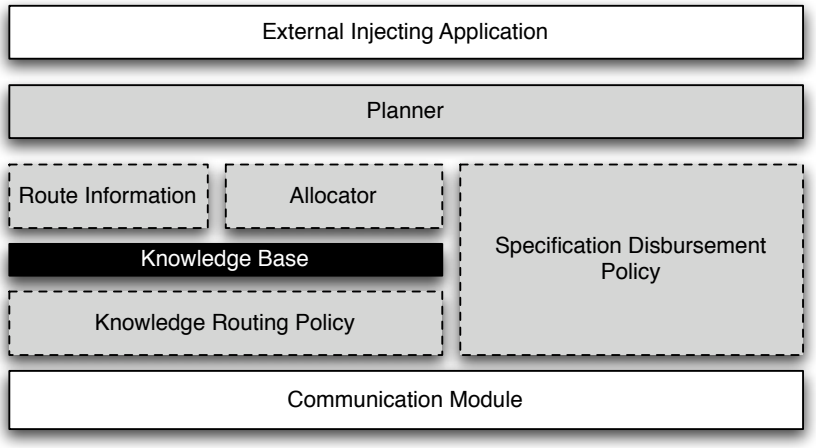

Figure 6.8: CiAN planning architecture

When the initiator is performing centralized planning duties (as opposed to fragmenting the workflow for a distributed allocation algorithm), an `Allocator` is used which implements the centralized algorithm.

The centralized allocation algorithm allocates each task in the workflow to a *suitable* host, where a suitable host is defined as a host whose capabilities are a superset of the capability requirements of the task, and whose motion pattern allows it to be at the location at which the task needs to be performed at the time it needs to be performed. The external application injects the workflow specification (encoded in the CiAN specification) into the planning system by way of the `Planner`. The `Planner` feeds the specification to the `RouteInformation` unit, which augments the specification with task numbers (used by the communication protocol that routes data between hosts). This augmented specification is then returned to the `Planner`. The `Planner` then passes this specification to the `Allocator`, which runs the centralized allocation algorithm to determine the hosts that are assigned each task in the workflow. It then annotates the specification with these allocations and returns it to the `Planner`, which subsequently forwards it to the `Specification Disbursement Policy` module, which breaks the workflow into its constituent tasks using a custom parser developed using the Java XML Processing API (JAXP), and sends each task specification to the host to which it has been allocated using the `CommunicationModule`.

The components with dotted borders are interfaces, i.e., they can be realized by different plug-able policies, e.g., the `SpecificationDisbursementPolicy` can be realized by the `SendAllTasksToAllHosts` policy or the `SendOnlyRelevantTasks` policy as long as they obey the relevant interface. The `Allocator` uses the `KnowledgeBase` as a resource for information about hosts, which are used to determine task allocations. The `KnowledgeBase` is populated using a gossiping protocol [96].

Distributed Allocation Algorithm

The distributed allocation algorithm consists of a centralized initiation phase followed by a distributed allocation phase. Each of the phases is implemented in CiAN as follows:

Initiation Phase. During the initiation phase, a workflow is input on the *initiator*. This host annotates the workflow with the metadata needed for routing purposes and then fragments the workflow into multiple pieces as indicated in the previous sections. The fragmentation algorithms are implemented as the `RouteSpecification` module which substitutes for the `Allocator` shown in Figure 6.8. Note that this in effect replaces the centralized planning algorithm above because the fragmentation is the only part that needs to be done centrally. Once the workflow is fragmented, the fragments are passed to the `SpecificationDisbursement` unit. Here, a policy is implemented that distributes these fragments to the coordinators (as opposed to the hosts as in the centralized algorithm). At this point, the distributed allocation phase begins.

Allocation Phase. The software for the allocation phase executes within the `LocalPlanner` shown in Figure 5.5 in Section 5.5. However, it should be noted that the algorithm is part of the `LocalPlanner` only on those hosts that are designated as coordinators. In effect, the `LocalPlanner` implements the "coordinator policy" which encodes the allocation phase of the algorithm.

The workflow fragments from the initiator are received by coordinators as `ControlMessages`. Since the `LocalPlanner` registers with the `WorkflowRouter` as a `ControlListener`, it is notified when the `ControlMessage` containing the workflow fragment is received. When the fragment is received, the `LocalPlanner`

sets up the necessary data structures and creates the solicitations which it packages within a `ControlMessage`. When a host comes within communication range, the `WorkflowRouter` detects this event and reports it to the `LocalPlanner`. The `LocalPlanner` then transmits the `ControlMessages` containing the solicitations via the `WorkflowRouter` and the `CommunicationModule`.

A *different* "worker host policy" is implemented within the `LocalPlanners` of the worker hosts. This policy receives solicitations as `ControlMessages` and responds with bids, also encapsulated within `ControlMessages`. The manner in which the `LocalPlanner` on the worker hosts receive the `ControlMessages` is identical to the coordinator. The `LocalPlanner` on the worker host is also responsible for any other communication with the coordinator such as accepting the final allocation of a task.

When hosts submit bids to the coordinator, the `LocalPlanner` processes these bids and runs the allocation process. As and when necessary, the `LocalPlanner` notifies the hosts with the winning bids of its allocation decision.

6.5 Results

6.5.1 Centralized Algorithm

The centralized allocation algorithm was evaluated by designing a simulator written in Java. A series of random workflows were generated and the time taken to find an allocation was measured. For comparison, a naïve algorithm was implemented that allocated tasks in a random order while the algorithm presented in this chapter employs the heuristic of allocating the tasks with the most constraints first.

Randomly generating a set of *realistic* workflows is difficult, mainly because the "realism" of workflows is hard to quantify. Instead, the random workflow generator generates a *diverse* range of workflows based on several parameters:

- r, the number of requirements that actions may draw from
- a, the number of actions in the workflow

- g, the number of agents in the system
- p_r, the probability that an action has a specific requirement
- p_c, the probability that an agent has a specific capability
- p_o, the probability of an agent having a constraint between actions

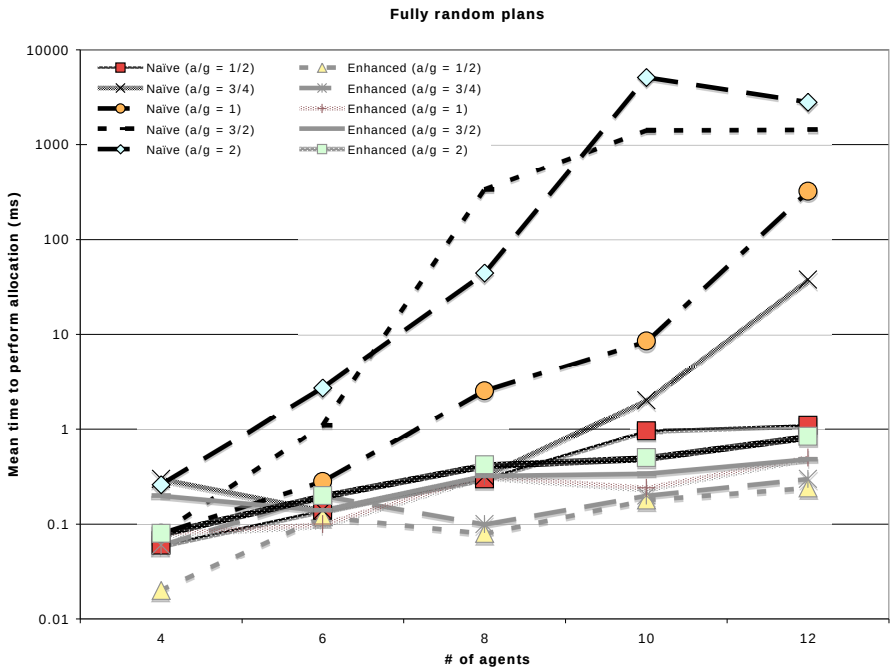

Figure 6.9: Algorithm performance when $p_o = 0.1$ for a mix of allocatable and un-allocatable workflows

Varying these parameters can help determine the effect that certain properties of workflows have on allocation performance. For the sake of simplicity, there is no distinction between spatiotemporal constraints and agent constraints. Rather, a set of constraint tuples are generated directly, without first generating a set of causes for those conflicts.

50 allocations of fully random workflows were performed using a wide range of values for these parameters, and the time taken for each version of the algorithm either to find an allocation or to determine that the workflow was impossible to allocate

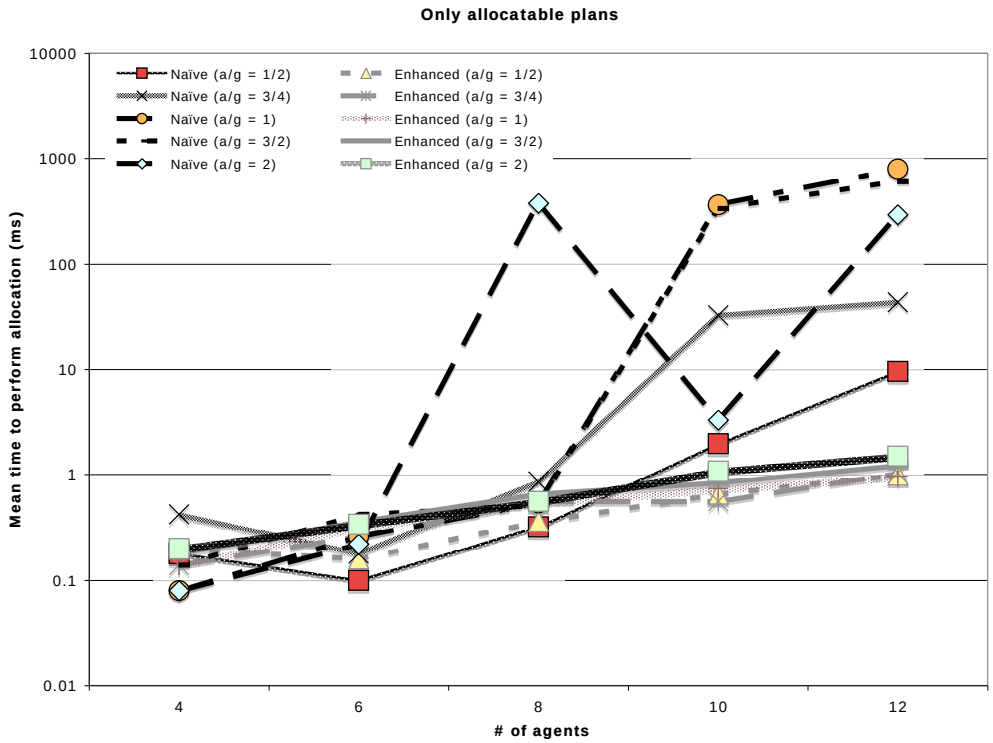

Figure 6.10: Algorithm performance when $p_o = 0.1$ for allocatable workflows only

was recorded. For comparison, this procedure was repeated with 50 more random workflows that were first filtered to ensure that an allocation existed. Since the decision space that the naïve algorithm traverses quickly becomes intractable as the number of actions and agents increases, an upper-bound of 30 seconds was enforced to find an allocation. In the interest of brevity, all results of all combinations of parameters are not presented here. However, it should be noted that the parameters that had the greatest effect on algorithm performance were the number of actions in the graph, the ratio of actions to agents, and the probability of conflicts. Figures 6.9 and 6.10 show the effect of varying the first two of these parameters with $p_o = 0.1, r = 8, p_r = 0.1$, and $p_c = 0.1$; Figures 6.11 and 6.12 show the effect of repeating these experiments with $p_o = 0.3$ and the other parameters unchanged. It should also be noted that the algorithm required no more than 10 ms to allocate any workflow of up to 24 actions, whereas the naïve algorithm frequently required more than 30 seconds to allocate the same workflow.

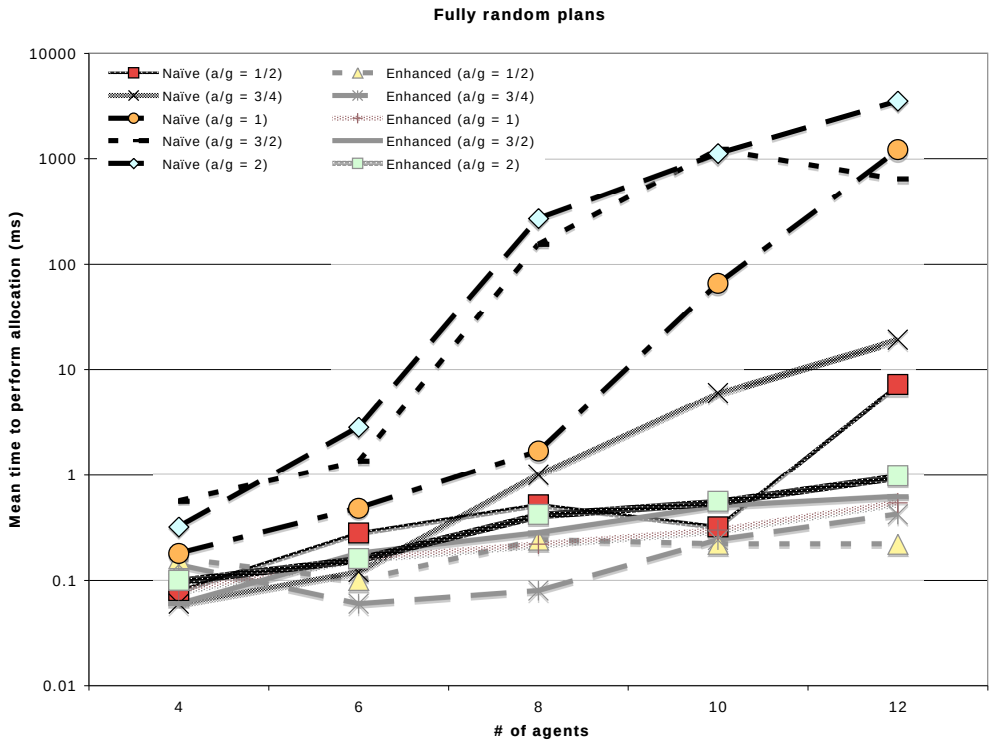

Figure 6.11: Algorithm performance when $p_o = 0.3$ for a mix of allocatable and un-allocatable workflows

The algorithm shows a significant performance improvement over naïve approaches because in the event of a wrong decision, the heuristic algorithm only has to explore the relatively small decision space of that sub-workflow before revisiting the incorrect decision. Furthermore, the algorithm does not need to traverse the entire decision space before it can conclude that a workflow is not allocatable.

6.5.2 Distributed Algorithm

The approaches described in this chapter were evaluated via simulation experiments using the workflow simulator for MANETs that is described in Chapter 7. The experimental setup is described first followed by details of individual experiments.

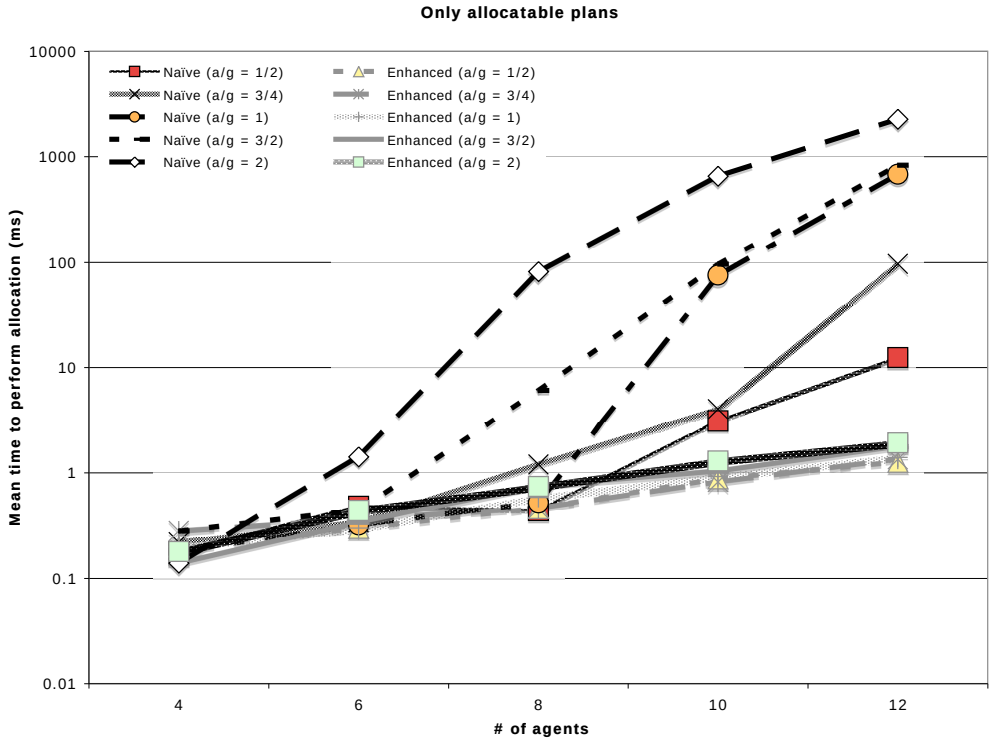

Figure 6.12: Algorithm performance when $p_o = 0.3$ for allocatable workflows only

Simulation Environment Setup. The simulation environment consists of a well-defined, non-discrete space of 100 x 100 area. This area is occupied by a number of hosts generated using the procedure detailed in Section 7.2. A host can move a maximum of 1.0 units in one time step though it may occupy locations with fractional values, e.g., (0.32, 6.71). Multiple hosts may occupy the same location simultaneously. Each host has a default communication radius of 2 units. Hosts are placed in the area at random locations initially. They move away from these initial locations as dictated by their mobility profile (in the cases presented here, the mobility is dictated by the controlled motion of the allocation algorithm in Section 6.3.2). The random workflows are generated using the algorithm described in Section 7.2. The set of workflows and hosts used are consistent across all experiments presented in this subsection. Workflows are split (when required) in a random manner. During allocation, the algorithm is held to strict timing constraints for each task which is more rigorous than the simple ordering requirement of traditional workflows. In addition, it is

ensured that every service required by a workflow is provided by at least one host in the simulation. However, it is not guaranteed that all workflows are allocatable.

E1: Comparing Centralized and Distributed Approaches. This experiment the distributed allocation scheme to the centralized allocation scheme presented in 6.3.1 (see Figure 6.13). For the distributed scheme, 4 coordinators were used. Each data point is an average of 300 workflows, 100 each with 10, 25, and 50 tasks. The experiments were conducted for cases where each host had a 0.1 and 0.5 probability of having a particular service required by the workflow. As expected, the centralized algorithm outperformed the distributed case because it has perfect knowledge of every host's capabilities and motion and therefore represents the optimal possible on-time allocation percentage for a given configuration. The distributed algorithm which is greedy and works in a just-in-time manner was able to allocate as many as 94% of the tasks on time compared to the centralized algorithm when a sufficient number of hosts was used. Greater numbers of hosts available to do tasks improved allocation numbers due to additional options to choose from. This same reason is responsible for the difference in numbers between the cases where the probability of the service being available on a host is 0.1 vs 0.5. Thus, for reasonable host densities and appropriate service distributions across hosts, the distributed approach which operates with partial knowledge of the environment can achieve an on-time allocation success rate close to the optimal success rate. It should be noted that the distributed algorithm allocated the remaining tasks successfully eventually, but just not within the timing constraints required.

E2: The Effect of Distributing the Allocation Process.

Experiment 2 shows the effect of different degrees of distribution on the allocation process (see Figure 6.14). Each data point is an average of 3000 different workflows. These workflows were executed by between 5 to 25 hosts with service probabilities of 0.1, 0.2, and 0.5. Once again, the most centralized approach (using the distributed algorithm with just 1 coordinator) performs best. However, the distributed case for the 2 coordinators is within 7% of the centralized case. Using a greater number of coordinators reduces performance because each coordinator has a small set of tasks and hosts spend a lot of time going from coordinator to coordinator to look for work. Also, with multiple coordinators and large number of tasks, it is possible a

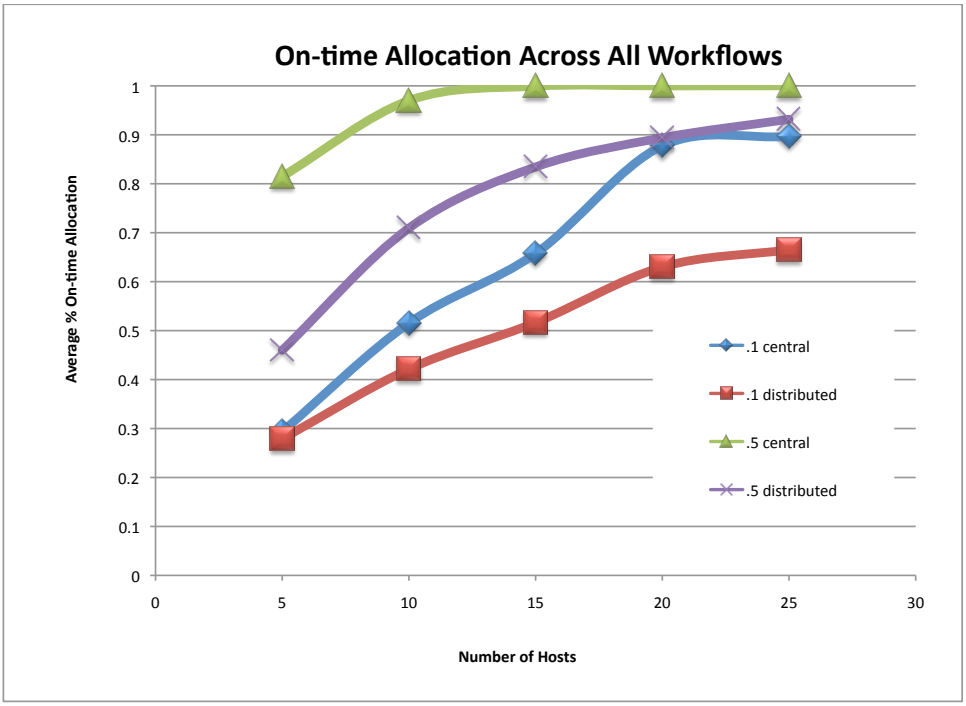

Figure 6.13: E1: Centralized vs. distributed allocation

host is waiting for an allocation from one coordinator when it is needed urgently at another. This indicates that the relation between the number of tasks, hosts, and coordinators must be rational. Simply adding more coordinators or hosts will not improve numbers. As can be seen, 2 coordinators sufficed for 50 workflow tasks and 50 hosts. However, multiple coordinators fare better in other circumstances described in following experiments.

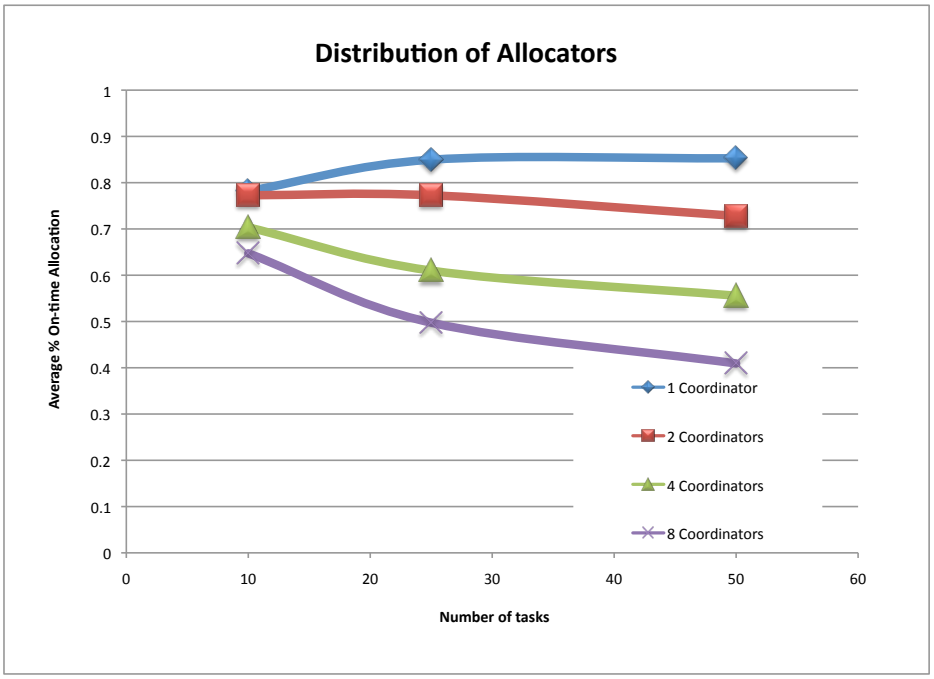

Figure 6.14: E2: Effect of using multiple coordinators

E3: The Effect of Controlling Host Motion.

The distributed allocation scheme assumes that a host initially seeks out a coordinator to look for available tasks and repeats this procedure whenever it is not working on a task. It also assumes that hosts move from coordinator to coordinator in a round robin manner if its initial choice of coordinator does not have any suitable tasks available. If both these assumptions are relaxed, and hosts move randomly, the allocation performance drops off if all other factors are constant. As shown in Figure 6.15, for the 4-coordinator case, when hosts move randomly, the percentage of tasks allocated on time drops off by about 10%. Since controlling host motion is a tradeoff between freedom of mobility of the hosts and the execution and allocation performance

of the workflow, this experiment helps quantify to a degree the implications of the tradeoff. Results are shown for the case when a host has a 0.1 probability of having a service and a 0.5 probability of having a service.

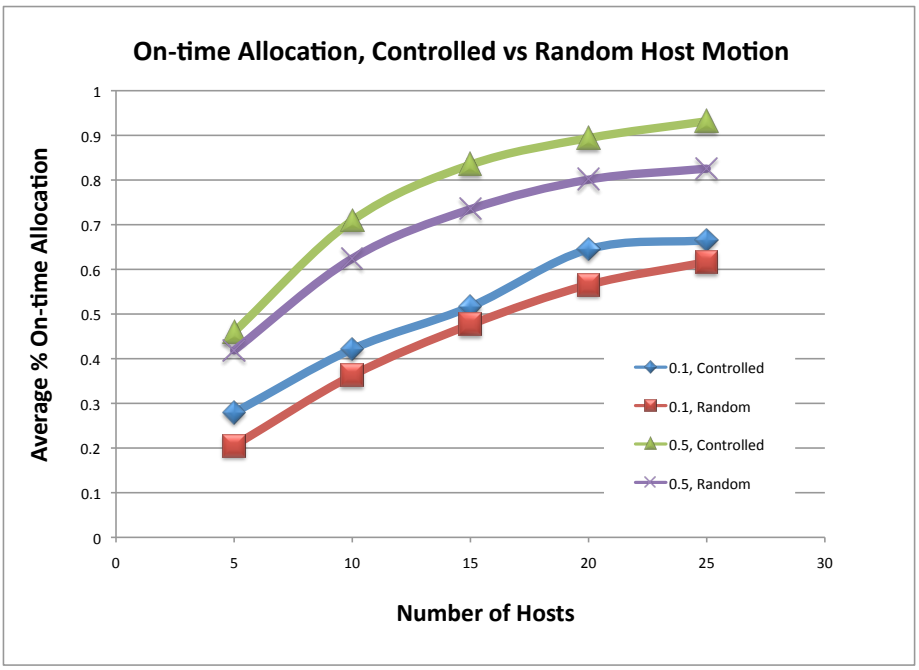

Figure 6.15: E3: Random vs. controlled host motion

E4: Enlarging the Area of the Simulation.

Figure 6.16 shows the effect of enlarging the area in which the workflow takes place. For the experiments thus far, a 100 x 100 space was used for the simulations. In this experiment, the trials of Experiment 3 were repeated using smaller and larger areas (small = half the length of the edge while larger = double the length of the edge) while continuing to use 4 coordinators. For comparison purposes, the results from E3 are shown as the "default size area" trend lines. As shown in the figure, the performance differential for random motion is significant when the area becomes larger. This is simply because the random motion pattern relies on pure chance to encounter coordinators and bid on tasks. When the area is larger, the probability of finding a coordinator goes down which results in fewer options for the coordinator and therefore lower allocation numbers. For controlled motion, while there is a difference, it is not as pronounced as in the random case. The discipline of the controlled motion

approach ensures that it does not suffer in larger areas. The performance drop off of the controlled motion cases is primarily due to larger distances being covered and therefore hosts not being as available as they would be in smaller areas. It is also worth noting that the controlled approach in a smaller area performs worse than the random approach because in this situation, the tasks and coordinators are in such close proximity to each other that visiting them in a systematic manner takes more time than just visiting one encountered randomly.

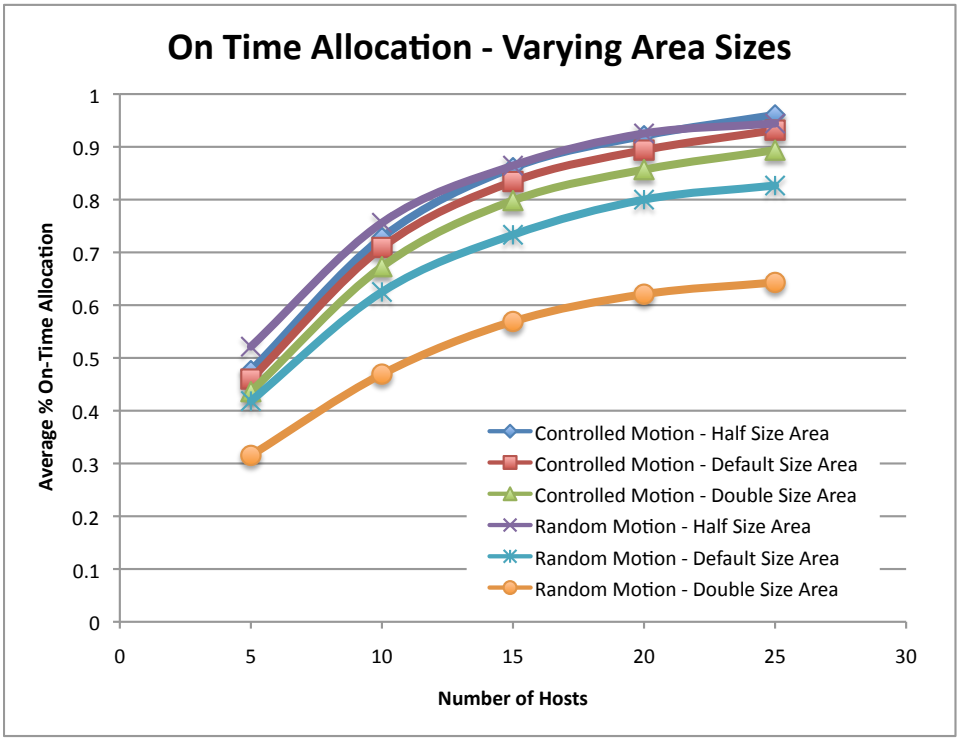

Figure 6.16: E4: Large vs. Small Areas of Simulation

E5: Using Geographic Distribution for Allocation.

In E2, the workflow was partitioned and tasks were distributed randomly to each coordinator for allocation. This experiment contrasts the random distribution with a localized or geographic distribution. In this scheme, each coordinator is responsible for an area around itself and tasks falling within that area are distributed to that coordinator for allocation. The results are shown in Figure 6.17 with the results

from E2 included for illustration. Using the geographic distribution method shows an improvement across the board in on-time allocations.

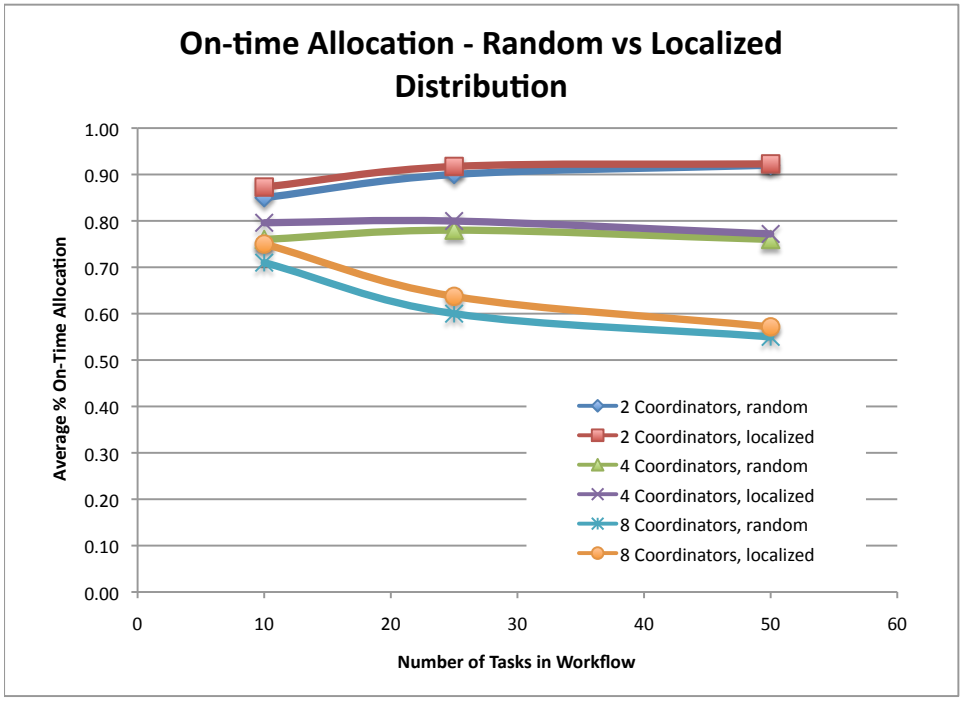

Figure 6.17: E5: Random vs. Geographic Distribution of Tasks

Discussion and Summary. The results shown in the experiments exhibit a wide range of allocation performance from as high as 95% to as low as 40% for the distributed algorithm depending on parameters. The results support the conclusion that there is no single recipe for ensuring high percentage of tasks allocated on time. Rather, configurations should be chosen carefully depending on the size and complexity of the task at hand. For example, in small areas with a large number of tasks to allocate, there is no need for constraints on motion whereas in a large area with a lower task density or host density, the motion constraints are a useful feature. In addition, the degree to which the allocation process is distributed should depend on the number of tasks to be allocated and the number of hosts available. Overall, the distributed algorithm with partial knowledge of the configuration performed no less than 82% as well as the optimal solution which was done offline with complete knowledge of the configuration. This number is especially encouraging given that the experiments were conducted with strict timing constraints. If the more traditional

ordering constraints were used, the algorithm allocates 100% of tasks eventually if the constraints on motion are applied. To summarize: (1) the distributed allocation performance is close to a centralized approach for reasonable configurations of hosts, workflows and coordinators, (2) motion constraints on hosts are useful in situations where task, host, and coordinator density is low, and (3) the use of geographic distribution of tasks helps allocation performance for large numbers of coordinators.

6.6 Chapter Summary

In a MANET environment, the process that assigns responsibility for workflow tasks to hosts must be aware not only of the capability of the hosts to do the tasks but also of their mobility pattern which will dictate whether they can receive inputs and communicate results to other hosts. This chapter described two allocation processes. The first is a centralized approach that takes into account mobility of hosts and the second is a process by which a monolithic workflow is divided into smaller pieces and then allocated in a distributed fashion by multiple coordinators in a MANET setting. The approach combines a bidding scheme with measures of utility and fitness to make allocation decisions. Experiments indicate that the approach works well when there are reasonable levels of host and task density where it allocates as many as 94% of the tasks on time. If minor constraints on host motion are applied, the algorithm eventually allocates all tasks in all situations.

Chapter 7

Mobile Workflow Evaluation Tools

The specification, engine, and allocation algorithms presented in the preceding chapters are designed to support collaborations involving many people in the physical world. Unfortunately, it is not always possible to test the system with large numbers of participants due to the lack of access to a large number of mobile devices. Hence, in addition to the demonstration applications, the solutions were tested using simulated experiments. Simulating this work required a *workflow-aware* simulator for mobile ad hoc networks (MANETs). While network simulators such as NS2 [1] and Omnet++ [2] can simulate MANETs to an extent, these packages do not support a tight integration between host mobility and the applications that execute on them. The mobile workflow simulator presented in this chapter is targeted to testing various approaches that require the simulation of large numbers of workflows executed by numerous hosts in mobile settings. The simulator provides the following salient features: (1) generation of randomized workflows, (2) generation of hosts with randomly selected attributes, and (3) simulation of all functions of a workflow management system (WfMS) via the execution of arbitrary plug-able code.

7.1 Introduction

A workflow management system (WfMS) is responsible for executing properly specified workflows. In the work presented in this dissertation, a workflow can embody any collaborative activity that has a fixed structure and acceptable workflows can range from those that describe the writing of an academic paper to those that coordinate the activities of workers at a factory. These workflows execute across multiple mobile

hosts that are capable of physical mobility and communicate with each other across a MANET.

Given that the hosts are essentially mobile devices carried by people, it is possible that certain actions within the workflow could prompt the user to change his/her mobility pattern or behave in a different manner. There is therefore a strong coupling between the actions of the workflow management system and the behavior of the hosts. Also, in contrast to network simulators, where the network performance is of key interest, for this work the focus is on the performance of the workflow management system. As such, this simulator is designed to be used to evaluate the various aspects of a workflow management system. The simulator offers the following features:

- **Random Workflow Generation.** The simulator generates random workflow structures, tasks, and synchronization semantics while obeying restrictions such as the number of tasks in the workflow, the degree to which the workflow is parallelized, the number of services the workflow requires, etc. For this, a special generation algorithm was formulated that preserves randomness while allowing categorization of workflows for comparison and repeated use in experiments.
- **Programmable Hosts and Applications.** The simulator provides generic hosts that can move freely around the environment. The programmer of the simulator can also load arbitrary applications on top of these generic hosts to define their behavior (the WfMS can be implemented as a series of applications). These applications can manipulate aspects of host behavior using hooks provided for the purpose. A fully-functional messaging system is also provided so that hosts can exchange messages with each other.
- **Physical Environment.** The simulator provides a well-defined physical environment in which the workflow takes place. In addition to providing a physical space, the simulated physical environment supports randomized placement of resources, hosts, and any other required entities.

The following sections describe the design of the random workflow generator and the simulator.

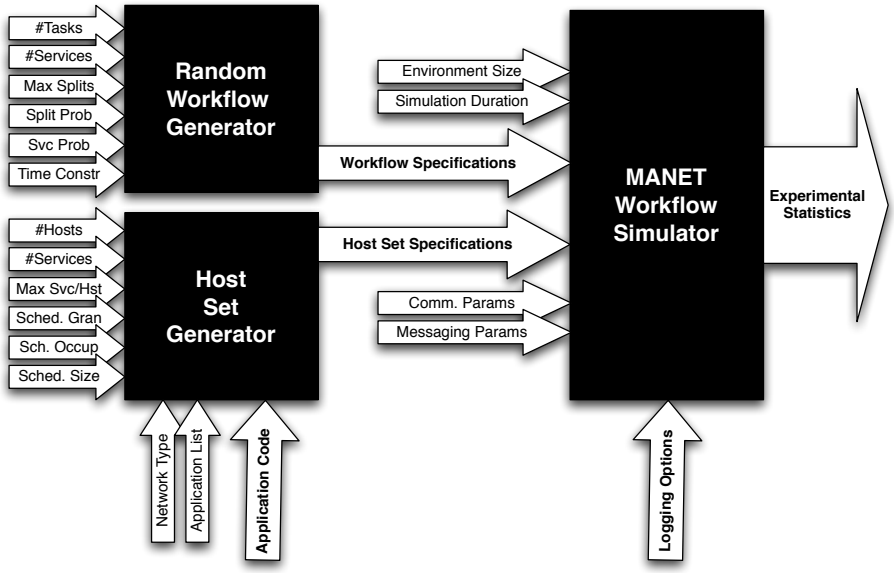

Figure 7.1: Inputs to and outputs from the workflow simulator

7.2 Simulator Design

A simulation of workflows across MANETs requires as inputs a set of workflows, a set of hosts that will cooperatively execute these workflows, and environmental parameters for the region in which the workflow is going to be simulated. Figure 7.1 shows the inputs to each step of the simulation and the relation between the various components of the simulator. Each of these components is described in detail in this section.

7.2.1 Random Workflow Generation

The goal in using the simulator is to be able to show that the developed solutions work on a wide range of workflows that have different structures and task characteristics. As such, having a large set of workflows whose structure and characteristics are randomized are crucial. However, at the same time, it is important that these workflows have certain degrees of similarity. For example, it is not reasonable to compare

the allocation time for a workflow with 10 tasks to a workflow with 100 tasks. As such, the algorithm takes in the following parameters as input:

- *Number of tasks.* The number of tasks in the workflow. All generated workflows have exactly the specified number of tasks.
- *Maximum splits.* The maximum number of edges that can come out of a single task node. This number must be less than the number of tasks in the workflow.
- *Split probability.* The likelihood of a task having more than one output edge.
- *Subset probability.* An operational parameter that dictates the degree to which the workflow has edges that are "long", i.e., that bypass several tasks.
- *Number of services.* The size of the set of services that can be used as requirements for tasks in the workflow.
- *Service probability.* The probability that a task will require a particular service.
- *Timing constraints.* The maximum duration of a task as well as the maximum possible gap between tasks.
- *Master service set.* The set of all possible services that can be included in a workflow.

Given these inputs, the workflow generation algorithm proceeds as shown in Figure 7.2. Initially, the algorithm creates a source task and a sink task, which are the anchors of the workflow. For each of the source and sink tasks, and for any subsequent tasks generated, a location is chosen randomly which represents the place where the task will be performed. In addition, a random subset of service requirements is chosen from the master set of services available. The size of the subset is dependent on the *number of services* and *service probability* parameters to the algorithm. Since workflows by definition are lattice structures, the cases of multiple source tasks or multiple sink tasks need not be handled. The newly created source and sink tasks are added to a `Workflow` data structure. A `Collection` called `left` is then created, to signify the expansion of the left side of the workflow, i.e., the source side. Initially, the source task is added to the `left` collection. The workflow is then built up recursively.

The recursive function first checks if the total number of tasks in the workflow has reached the number needed, in which case it returns the contents of the `left` collection. The main function (`GenerateRandomWorkflow(...)`) adds edges from all tasks in `left` to the sink task, effectively forming the "right" portion of the workflow. If the total number of tasks is not reached, the recursive function generates a new task to insert into the workflow. It then chooses a random subset of the tasks in `left` and creates edges between those tasks and the newly created task. It then checks each of those tasks in the randomly generated subset and removes them from `left` if they have reached the maximum number of outgoing edges or if they are randomly chosen for elimination. This recursion continues until the requisite number of tasks have been created.

Next, time values are generated for tasks. A random start time and duration is chosen for the first task. To get the deadline for this task, the duration of the task is added plus a random *buffer* (subject to the *timing constraints* parameter to the algorithm) to the start time of the task. For each of its children, a start time is assigned that is a random amount of time after the deadline of the root task (once again, subject to the *timing constraints* parameter). Note that this random time is generated separately for each edge. The duration and deadline are generated in the same way as the first task. When tasks have several incoming edges, they use the deadline of the task that finishes the latest to generate a start time. The workflow is split at this stage for applicable experiments (see Section 6.3.2). The workflows are written to a file which is in turn passed in as an input to the simulator.

7.2.2 Random Host Generation

Workflows are just one part of the simulation. The tasks in the workflow must be executed by hosts that can move around the environment. The host generation process generates sets of hosts that have certain characteristics in common to ensure that the experiments do not compare between two situations involving hosts that are not similar. The host generation process takes in the following parameters:

- *Number of hosts.* The number of hosts in the set to be used for a particular workflow execution.

```
GenerateRandomWorkflow(numTasks, maxSplit, splitProb, subsetProb)
    sourceTask ← GenerateNewRandomNode()
    sinkTask ← GenerateNewRandomNode()
    wf ← new Workflow(sourceTask, sinkTask)
    left ← ⊥
    Add(left, sourceTask)
    left ← InsertTaskAndEdges(wf, left, numTasks, maxSplit, splitProb, subsetProb)
    for each t ∈ left
        if GetSuccessors(t) == ⊥
            edge ← new Edge(t, sinkNode)
            AddEdge(workflow, edge)


InsertTaskAndEdges(wf, left, numTasks, maxSplit, splitProb, subsetProb)
    if Size(wf) ≥ numTasks
        return left
    task ← GenerateNewRandomNode()
    oldTasks ← GetRandomSubset(left, subsetProb, numTasks, Size(wf))
    for each t ∈ oldTasks
        edge ← new Edge(t, task)
        AddEdge(workflow, edge)
        if Size(GetSuccessors(t)) ≥ maxSplit or GetRandomNumber ≥ splitProb
            Remove(left, t)
    Add(left, task)
    return InsertTasksAndEdges(wf, left, numTasks, maxSplit, splitProb, subsetProb)
```

Figure 7.2: Random workflow generation algorithm

- *Maximum number of services.* The maximum number of services that a host can have.
- *Number of services.* The total number of distinct services across all hosts (this must match the number used to generate any workflows that are executed with this host set).
- *Schedule granularity.* The duration of a single entry in the host's schedule.
- *Schedule occupancy.* The fraction of a host's schedule that is marked as busy initially.
- *Schedule size.* The number of entries the host's schedule can hold.

In addition to these input parameters which generate "basic" hosts, i.e., hosts which have no programmatic behavior, the following additional parameters can be specified that give the host its behavior within the simulation.

- *Network type(s).* The name(s) of the network(s) that the host is a part of and what type they are (ad hoc, wired, etc.).
- *Application list.* A list of the names of the applications that will run atop the simulated host.
- *Application code.* The actual code (specified in terms of the names of Java class files) of the applications to be loaded on the host.

These parameters are all used to generate randomized hosts (by varying the value of the parameters up to the maximum allowable). The host objects are written to a file which is then passed in as an input to a simulation instance.

7.2.3 Workflow Execution Environment

The execution environment of the simulator has been designed in a generic manner such that only the most basic resources are provided as part of the simulation system. All experimental functionality that is to be tested using the simulator is integrated into the simulation using plug-able modules. This design allows the same basic physical environment and hosts to be used to test different protocols and strategies without having to generate a new environment for each. An initial set of plugins is provided, which implement basic protocols and strategies to help the programmer of the simulator get started in a reasonable period of time. Simulations occur in a lock-step fashion involving 4 steps– move, update communication tables, communicate, and process, relating to the physical motion of hosts, exchange of messages across the network, and the local processing of those messages by applications. The choice of the lock-step mechanism as opposed to continuous, parallel simulated execution of hosts was due to the fact that minuscule timing differences between the approaches are not of interest. Rather, observing if a certain sequence of events in a particular order results in a successful execution is the primary goal of running these simulations.

Figure 7.3 shows the architecture of a virtual simulated host in the simulation. At the bottom, the host has access to a network controller which encapsulates several network interfaces. Network interfaces not only encapsulate the *type* of network but also the *instance* of network. Therefore, each network interface has a type as well as

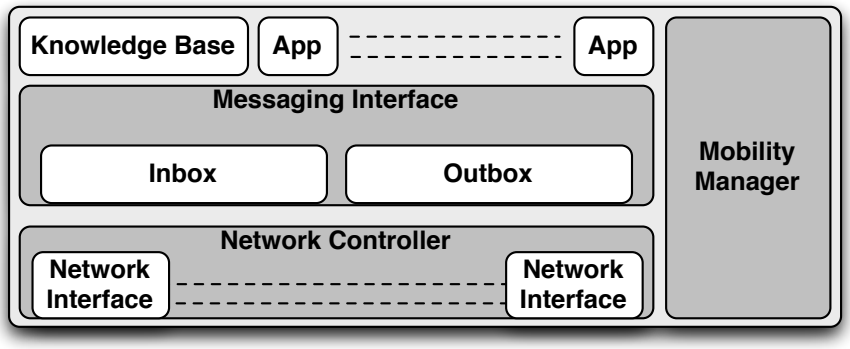

Figure 7.3: Architecture of a host in the simulator

a name. Each type of network has associated with it several policies that dictate the rules for two hosts to be in communication with each other, routing methodologies and other such details. By creating a network interface of a particular type, a host is bound to obey the rules associated with that network type when sending and receiving data through that network interface. Currently, the types of networks provided are *wireless ad hoc* with a static communication range and *wired* with routing. The network interface architecture is itself completely plug-able and therefore can support user-designed custom network types beyond the three types provided.

Above the network controller sits the messaging interface that is responsible for sending and receiving messages between hosts. The messaging architecture is set up using mailboxes rather than direct delivery of messages to decouple the applications that reside above it from the network below. The messaging interface provides an inbox and an outbox for incoming and outgoing messages respectively. Both of these use mutual exclusion constructs to ensure that applications cannot access them at the same time as the network controller. Applications can register for notifications when certain types of messages (that are of interest to them) are placed in the inbox by the network controller (as a result of it having received them from another host in the simulation). It is up to the application to then subsequently retrieve the message from the inbox. Messages placed in the outbox are transmitted to the target host unless that host is not reachable from the current host (as defined by the appropriate network interface). In such cases, the message is retained in the outbox and a retransmission is attempted periodically until the message can be delivered.

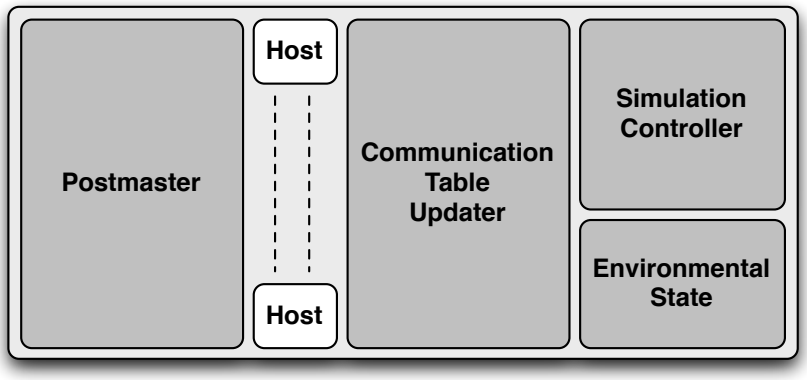

Figure 7.4: Architecture of the execution environment of the simulator

Atop the messaging interface sits the knowledge base which captures non-functional information about hosts. It is similar to other applications with the exception that it is automatically instantiated during a simulation and is made available as part of the simulation code. This decision was made in support of the view that knowledge is essential to execution across mobile networks. In the case that the programmer does not need the knowledge base, he/she can disable it by setting a flag. The structure of the knowledge base is a slightly simplified version of the knowledge base described in Chapter 3. Applications that run on hosts encapsulate the logic that the host carries. A host can support multiple applications and the programmer of the simulation can choose to have facilities that allow applications to interact with each other. The programmer can create these applications by simply extending an abstract class that provides the hooks needed for notification. Beyond that, the programmer can place arbitrary code in the applications.

The mobility manager is responsible for guiding the host through the environment. Depending on the policy used by the programmer, the mobility of the host can be autonomous or influenced by messages or any application. As with all other components in the host, the mobility manager can be customized with plug-able policies to create custom motion patterns and rules.

Figure 7.4 shows the architecture of the simulator execution environment. The key components are the list of hosts and the environmental state component. The environmental state component maintains a record of the size of the environment, the

attributes of each location in the environment, and any other details that are considered to be part of the state of the environment. The host list is simply a list of hosts (as described earlier in this section) that have applications, motion policies, and one or more network connections. This host list is populated from the file generated by the host generator which forms an input to the simulator.

The `Simulation Controller` is the "brain" of the simulator and manages the simulation in lock step. The `Simulation Controller` works as follows: (1) It iterates through the list of hosts and requests each of them to move to their "next" location. The next location of a host is determined by its mobility controller depending on the policy it is implementing. (2) After all hosts have moved, the `Simulation Controller` invokes the `Communication Table Updater` to update the list of possible communication links based on the new locations of all the hosts. (3) Once the links have been updated, the simulation controller invokes the `Postmaster`. The `Postmaster` iterates through the outboxes of each host, takes each message and places it in the inbox of the target host (subject to their being a valid communication link between them - for this it consults the `Communication Table Updater`). (4) Once the messages have been delivered, the `Simulation Controller` iterates through each host and requests them to run each of their applications for one step. Once this is complete, the simulation goes back to step 1 and repeats the process. The simulation runs for a prescribed number of steps which is specified when the simulation is initialized.

7.3 Implementation Details

The workflow simulator for MANETs is implemented in Java 2 SE using the Eclipse development environment. This section describes the implementation details of the generic simulator followed by the implementation of some of the plugins were used to evaluate CiAN in the context of this simulator.

7.3.1 Generic Simulator

The generic simulator is a collection of core classes, most of which are abstract classes or interfaces. The intention is for programmers to extend the classes or implement the interfaces and add custom functionality according to the needs of their simulations. The classes in the generic simulator are described under three categories: (1) host, (2) environment, and (3) network.

Host

The `Host` class is used to create a host within the simulator. The `Host` class implements two interfaces `AppHost` and `EnvHost`. The `AppHost` interface defines the methods that a host needs to expose to applications while the `EnvHost` interface exposes the methods that the environment needs to manipulate on the host and perform actions such as notifications to move, message delivery, etc. While rare, in theory, a programmer could create hosts by implementing either the `AppHost` or the `EnvHost` interfaces alone. Implementing only the `AppHost` interface would create a host that would not be placed in the context of an environment. Implementing only the `EnvHost` interface would create a host that had no resident applications (potentially useful for simulating the outcome of different mobility patterns only).

The `AppHost` interface provides methods to put messages in the outbox, check the inbox for messages, add and remove `Applications`, add and remove network interfaces, and obtain neighbor lists. The `EnvHost` interface contains methods for putting messages in the inbox and checking the outbox, getting lists of the network interfaces the host has, and adding and removing neighbors from the host's neighbor list. The `Host` class maintains one instance each of the `Inbox` and `Outbox` classes as well as a `Collection` of `HostNetworks` (that encapsulate the network interfaces) and a `NeighborCollection`, which is a dynamic neighbor list. The `Host` class also provides accessors and mutators for the data members of the class and implements the generic behavior of the methods required by the two interfaces. Customized hosts can be created by extending the `Host` class.

Environment

The `Environment` class is the primary class that represents the physical environment and contains the logic for starting and advancing the lock-step simulation. The `Environment` class encapsulates a `Collection` of `EnvHosts` which represent the hosts in the simulation. To keep track of communication links, it maintains a communication table that is a mapping from the `ID` of an `EnvHost` to a `Collection` of `ID`s of other `EnvHost`s that the particular host can communicate with. This table is updated at every time step.

The `Environment` class also maintains the global notion of time for the simulation. Time is tracked as a `long` value that is incremented after each cycle of four steps as outlined in the previous section. Each `Host` is notified of the updated time each cycle and they in turn notify any resident `Application`s of the updated time.

In addition to static data members, the `Environment` class has two additional components, the `PostMaster` and the `CommunicationTableUpdater`. The `PostMaster` takes messages from the `Outbox`es of `Hosts` and delivers them to the `Inbox`es of the target `Host`. The `CommunicationTableUpdater` updates the communication table to ensure that the list of hosts that a given host can communicate with is consistent with the policies defined in the appropriate network interface.

Network

The simulator supports the existence of several networks within the physical environment. A network is embodied by the `HostNetwork` class which encapsulates a name and type for the network. Hosts that have `HostNetwork` objects with the same network name can communicate with each other subject to the constraints of the type of network. Two types of network implementations are provided - `WiredInternet` simulating a stable wired network and `WirelessAdHocBiDirectional`, simulating wireless ad hoc networks. These two classes can be used in combination to simulate nomadic networks.

The simulator also provides the `NeighborCollection` class that tracks the neighbor list of a given host dynamically in conjunction with the `CommunicationTableUpdater`.

A generic `Message` class is also provided which can be used as a generic wrapper for more specific messages sent between hosts.

7.3.2 CiAN Plugins

One of the motivations in developing this simulator was to test the various protocols, algorithms, and strategies that form the CiAN workflow management system. The simulation of CiAN within the simulator is briefly described as a case study for the successful use of the simulator.

The encoding of a CiAN workflow is presented first. For this, several classes were implemented that are Java embodiments of elements in the CiAN specification. The `CiANWorkflow` class encapsulates an entire CiAN workflow and contains several `CiANTask` objects. The `CiANTask` objects contain `CiANEdge` objects that specify the connections between the `CiANTask` object. The `CiANWorkflow` is passed into the simulation by a `CiANHost`, in a manner similar to a real execution of CiAN, where an initiator would inject the workflow into the system.

The `CiANHost` interface's concrete implementation is contained in the `CiANHostImpl` class. As with hosts in the simulation, the `CiANHostImpl` class can support arbitrary applications. The `CoordinatorTaskDistributer` class implements the functionality of the coordinator in CiAN while the `WorkerTaskBidder` class contains the logic for worker hosts. In addition to these two main classes, a `Schedule` class was implemented that is used to track the host's schedules. The host's motion logic is captured in the `WorkerMotionController` or the `CoordinatorMotionController` classes depending on the type of host (worker or coordinator).

The allocation process is contained within the `CoordinatorTaskDistributer` class. This class maintains lists of `TaskAllocation` objects that track the allocation status of various tasks. This application accepts `TaskBid` objects from worker hosts as bids to perform tasks.

7.3.3 Anatomy of Simulation Execution

A simulation of the CiAN system using the simulator works as follows. Before the simulation begins, the `LatticeGraphGenerator` is used to create several random workflows and the `CiANHostGenerator` is used to create the worker hosts and coordinators. The workflows generated are loaded onto one of the generated hosts (the initiator) and with the other hosts (coordinators and workers) is passed to the `MANETSimulator` class. The `MANETSimulator` class instantiates an `Environment` object passing in the list of hosts and configures all logging functions. At this point the simulation is ready to run.

The `MANETSimulator` class invokes the `start()` method on the `Environment` which begins the simulation. The `Environment` first invokes the `deliverMessage()` method on the `PostMaster` which ensures that the messages in each host's `Outbox` is delivered to the `Inbox` of the target host subject to the constraints in the respective policies that implement the `HostNetwork` interface. Then, the `Environment` calls the `updateCommunicationTable(...)` method on the `CommunicationTableUpdater` class to update the connectivity map. Note that since in CiAN host motion is dependent on the actions of the worker host and the coordinator, the `Applications` running on the hosts are responsible for actually moving the host.The `Environment` then iterates over all the hosts and invokes their `stepAllApplications(...)` method which makes each `Host` run their applications for one time step. This process is repeated until all applications on all hosts have terminated.

The logic of the CiAN system is captured in the `Application` class, more specifically the `CoordinatorTaskDistributer` and the `WorkerTaskBidder` classes. Each time these classes are instructed to execute a time step (via a call to their `step()` method) by the `CiANHost` (in response to the call from the `Environment` to each hosts `stepAllApplications(...)` method), they take a step of the allocation algorithm, execute tasks in the workflow, or simply move towards a coordinator or the location of their task. Thus, it is the various `step()` calls that result in the progression of the simulated CiAN system.

7.4 Chapter Summary

This chapter has described the design and implementation of a workflow simulator for mobile ad hoc networks. The simulator is designed in a generic fashion, providing basic entities such as hosts, environments, and messaging infrastructure. The remaining components are designed to be plug-able, allowing the programmer of the simulator to customize the behavior of each component. This level of customization makes it possible for a vast variety of protocols and strategies to be evaluated under very different simulated environmental conditions and mobility patterns. The simulator described in this chapter has been used for the experiments described in Chapter 6 and is in use by other projects for simulating strategies associated with more dynamic workflows.

Chapter 8

Future Work

The work presented in this dissertation addresses the challenges of building a workflow-based collaborative computing infrastructure across mobile networks. While the various algorithms and protocols presented serve a specific purpose, the software systems, in particular the CiAN workflow engine, have been engineered with future expansion in mind. As such, they serve as a platform upon which additional functionality and more sophisticated approaches can be built. A selection of potential extensions are listed here.

Fault Tolerance. Fault tolerance refers to a workflow management system's ability to recover from errors and continue the execution of the workflow in its original form or at the very least, complete the execution in a degraded mode of operation. In traditional workflow management systems (WfMSs), the source of errors are purely computational, i.e., software or hardware failure, and there exist various mechanisms to tackle these [89, 54]. WfMSs designed for mobile environments must be able to deal with such computational errors as well a whole new set of error types that arise from operating on mobile hosts across a physical environment. Such errors might be due to host mobility (host not within communication range), environmental factors (resources not being available or environmental parameters not being within tolerance levels), as well as random, unforeseen events. The problem is further exacerbated by the fact that in MANETs, it is difficult to detect faults in a timely fashion due to lack of communication guarantees and information about the errors cannot be propagated within a community due to the same reason. As such, a well-integrated fault tolerance and management system is required that can manage (potentially redundant) executions, detect and transmit fault information, encapsulate recovery

protocols, and work in conjunction with an *institutional knowledge base* to reduce the chances of a recurrence of the error.

Institutional Knowledge. In typical practice, a workflow specification is written once and then is instantiated by the WfMS for each individual *case*. Just as human beings learn from experience, there is a learning opportunity from executing a workflow multiple times for different cases. The benefit of using knowledge about the environment and the network in the execution of a workflow (for a single case) has already been demonstrated in Chapter 3. However, there does not exist a mechanism to transfer knowledge across *multiple* discrete executions of the same workflow. The *institutional knowledge base* concept seeks to address this issue by storing information about the execution of a workflow case that might be beneficial to subsequent case executions. In contrast to the knowledge base presented in Chapter 3, the institutional knowledge base would store primarily functional information that can include but is not restricted to situations in which errors occurred, the causes of errors, protocols for error recovery that had proven or previously demonstrated chance of success, hosts that are more reliable for doing work compared to others, and environmental factors that promote or hinder the execution of the workflow. This research effort would need to formulate representations for the information, mechanisms to automatically acquire and store this knowledge, and integrate the knowledge base itself with decision-making components of the system.

Security and Access Control. In any software system, security and access control are very relevant concerns. A security framework for WfMSs can be approached in two stages. In the first stage, an assumption can be made that there exists a group of participants that do not behave maliciously towards each other. If admission to this group is controlled carefully, there need not be any further security within the group. The development of such an access control scheme would require determining the credentials that a participant needs to present in order to be admitted into the group. For further fine-grained control within the group, a set of role-based permissions and access rights can be set up so that participants can only view the subsection of the workflow and its data according to the level of access granted, e.g., a worker cannot view other workers' evaluation forms but a supervisor can. This type of approach is suitable especially for workflows that execute in areas where electronic surveillance is at a minimum, such as a remote geological survey or a nature exploration activity.

In the second stage, no assumptions need be made and appropriate security and encryption frameworks can be used within the group. The challenge is to design a lightweight security framework that can function effectively even when the system components are fragmented and scattered over large physical areas.

Emergent and Unstructured Workflows. The workflow model has been demonstrated as being powerful yet simple enough to model any structured collaborative activity. However, all collaborative activities are not structured, especially a large selection of those that happen in the physical world. Consider the example of a toxic chemical spill where first responders must work together to rescue people and clean up the spill. Such situations evolve very rapidly creating different pressures on everyone's time. The collaborative nature of the activity is not well structured and emerges in response to various environmental factors. Workflows and workflow management systems need to be adapted to deal with dynamism in the network *as well as* dynamism in the workflow structure itself. This will allow workflow technology to reach broader definitions of collaboration.

Chapter 9

Conclusion

Mobile computing is clearly a wave of the future with increasingly more hardware and software capability becoming available for computing on the go. The unique nature of mobile devices and the fact that they are carried on the user's person have created an opportunity to engineer a new generation of software that has the potential to bring computing technology to aspects of life that have not been previously imagined. The work in this dissertation focuses on one such type of software - collaborative systems based on workflow management for mobile networks. This dissertation has made the following contributions:

- The design and implementation of a *knowledge management system* that gathers and maintains non-functional information about hosts in a mobile network. This information is used by other distributed software components to make communication decisions and offer weak guarantees.
- The formulation of the *CiAN specification language*, an XML-based specification language for choreographed workflows that are designed to execute in the physical world across MANETs.
- The design and implementation of the *CiAN workflow engine*, to my knowledge the first choreography-based workflow management system for MANETs.
- The formulation and evaluation of *algorithms for allocation* of workflow tasks taking into account individual participants' capabilities as well as their mobility and other spatiotemporal constraints.

- The design and implementation of a *simulator for mobile workflows* for evaluating the approaches and contributions of this thesis.

The results presented in the chapters of this dissertation show that for reasonable assumptions of non-adversarial host motion and properly structured workflows, the algorithms and software design stand up well to tests. However, these contributions are simply an initial step. As mobile hardware technology continues to advance, it will open the door for offering even more sophisticated functionality on mobile devices. The work presented represents a forward-looking foundation atop which such advanced approaches could be built.

Appendix A

Additional Experimental Data

The purpose of this appendix is to expand on the evaluation data shown in previous chapters for certain experiments. The data for the allocation experiments are presented first followed by data for the publish-subscribe protocol.

E1: Comparing Centralized and Distributed Allocation Approaches. Figures A.1, A.2, and A.3 show the data for the centralized allocation algorithm vs. the distributed allocation algorithm (with 4 coordinators) for 10, 25, and 50 task workflows, with each data point being an average of 100 workflows. It can be seen that holding the number of hosts and coordinators constant (as they are across the three figures), increasing the number of workflow tasks results in decreased performance, especially in the case where the likelihood of a host having a service is 10% as opposed to 50%. In other words, the more specialized the hosts are, the more the distributed approach struggles to allocate tasks. Also, increasing the number of tasks results in there not being enough qualified hosts to do the work which results in a drop in the allocation numbers. The fluctuations in the graph were due to certain workflows requiring services in a manner that caused conflict. In all the experiments, the outcome is reliant as much on the algorithm as on the distribution of services across hosts, the initial location of the hosts, and the order in which hosts reach the coordinator to be allocated tasks. The distributed approach numbers approach those for the centralized case when hosts are less specialized and the ratio between tasks to be performed and hosts available is reasonable.

E2: Using Multiple Coordinators. Figures A.4 through A.9 show the data for Experiment 2 in Section 6.5 from a slightly different perspective. Figure A.4 shows the percentage of tasks allocated on time as a function of the number of worker

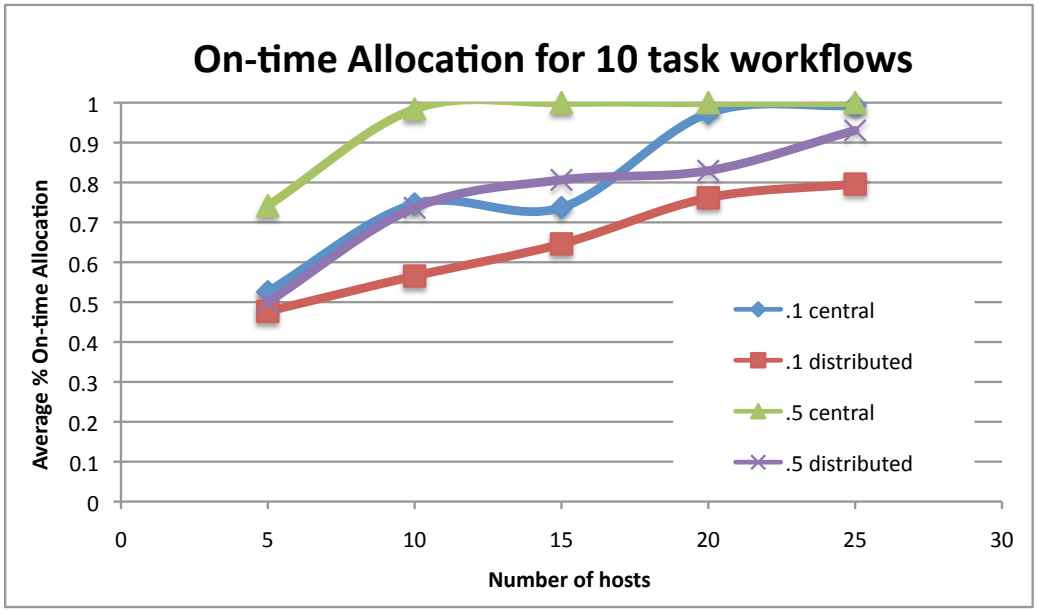

Figure A.1: On-time allocations for 10 task workflows

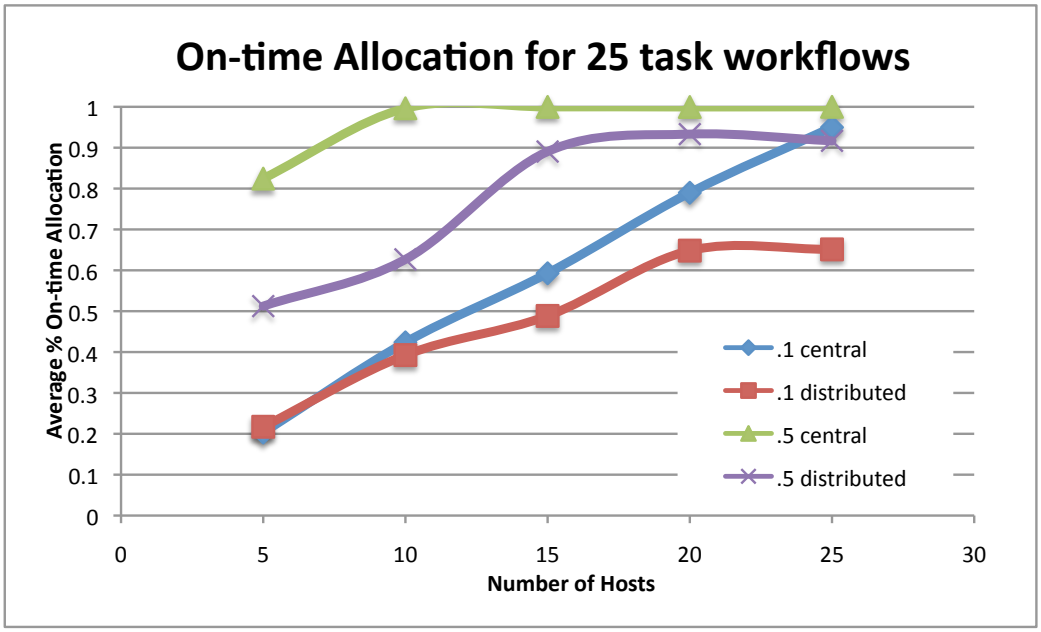

Figure A.2: On-time allocations for 25 task workflows

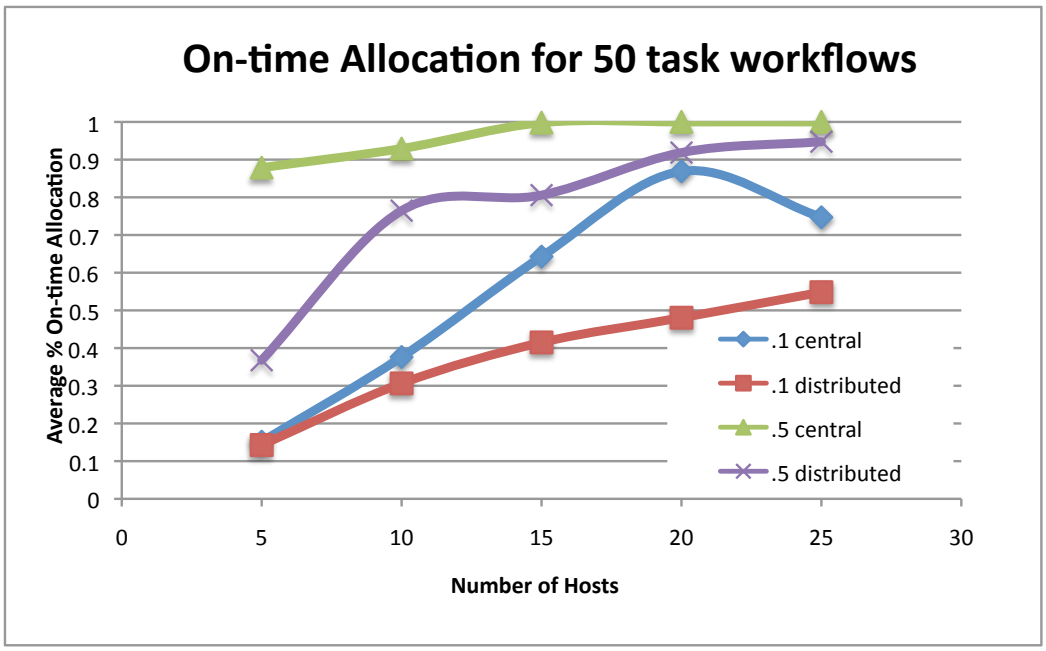

Figure A.3: On-time allocations for 50 task workflows

hosts used to complete the workflow. Observe that initially the numbers for multiple coordinators are worse than those for a single coordinator (i.e., with centralized allocation). However, as the number of hosts increases, the difference is decreased and at 25 hosts, the 2 coordinator case is better than the single case and the 4 coordinator case is comparable. This supports the conclusion that there must be a reasonable ratio of coordinators to hosts. Up to 20 hosts, the single coordinator does better but beyond, multiple coordinators are desirable. This is because there is a tradeoff between having one coordinator for all the tasks, which increases the amount of travel hosts have to undertake between task locations and the coordinator, and the use of multiple coordinators which increases travel between coordinators as hosts move from one to another looking for a suitable task. Having multiple coordinators also can lead to a host being locked at one coordinator when there is a more crucial task on another coordinator that it could do. It is for these reasons that for low numbers of hosts, fewer coordinators are preferred.

Figure A.5 shows the percentage of tasks allocated on time as a function of the probability of services being available on a particular host. The differences between the 1, 2,

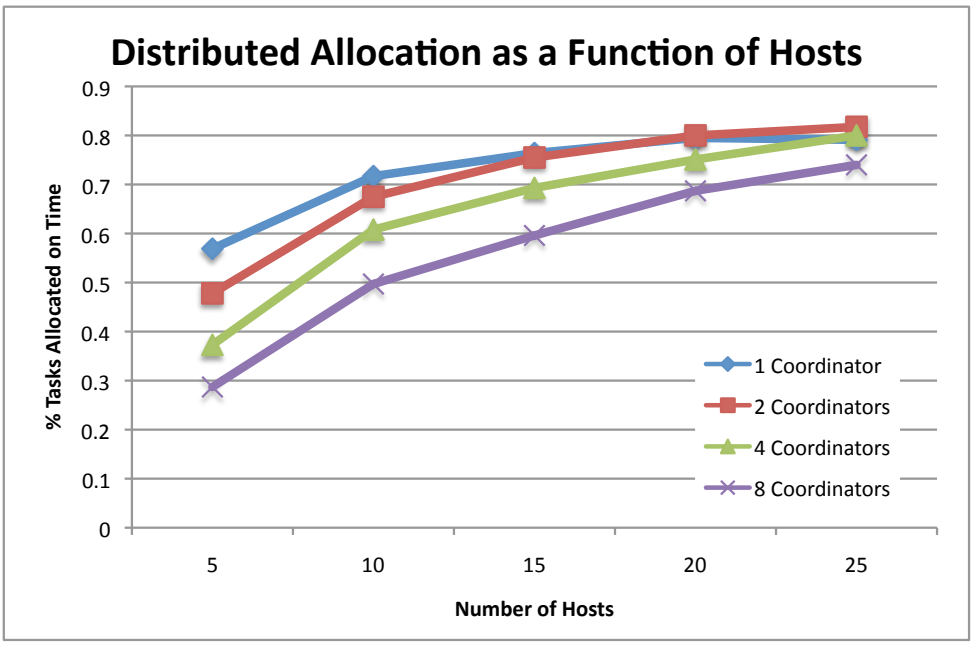

Figure A.4: On-time allocations as a function of number of hosts with different numbers of coordinators

4, and 8 coordinator cases have already been explained in Figure A.4. It can be seen that the performance when the chance of finding a service on a host is 50% is better than when the chance is only 10%. This is in line with expectations because a higher probability of finding a given service on a particular host translates into more options for the coordinator to pick from resulting in an even distribution of the workload. For the pathological case where every host can do every task, the algorithm does not perform as well because the provisional allocation and re-evaluation mechanism relies on comparisons between bids submitted by hosts to decide on an allocation, and in the initial stages all hosts are equivalent for this purpose. It should be noted that in this case, the need for allocation is in a sense eliminated since a simple greedy strategy of allocating tasks to hosts on a first come first serve basis would suffice.

Figures A.6, A.7, A.8, and A.9 show on-time allocations for 10, 25, and 50 task workflows. As explained earlier, having more hosts increases the allocation performance due to there being more allocation options. Typically, the performance did not differ greatly as a function of workflow size which is encouraging. In the 1 coordinator case, the workflow with 10 tasks had many parallel conflicting requirements and the poor

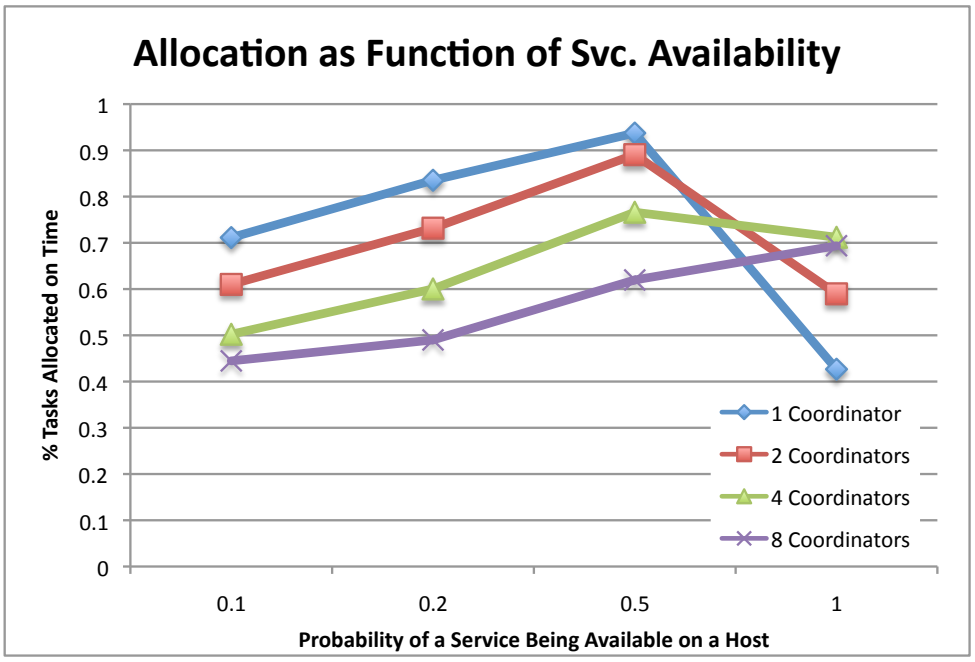

Figure A.5: On-time allocations as a function of probability of availability of services with different numbers of coordinators

performance was due to the coordinator "reserving" a qualified host for a particular task to the detriment of the conflicting task. In the 8 coordinator case, the difference is more marked due to the random distribution of tasks across coordinators which resulted in a lot of hosts going back and forth between coordinators looking for tasks, which ultimately made them unsuitable for being allocated tasks on time.

E3: The Tradeoff Between Random and Controlled Motion.

Figures A.10 and A.11 show additional data for Experiment 3 in Section 6.5. Figure A.10 shows that for high numbers of tasks in the workflow, the random motion patterns do better. This is because a high number of workflow tasks translates to a high number of tasks on each coordinator on average. Thus, a host moving randomly has a good chance of coming across a coordinator *and* the coordinator having something suitable to allocate to that host. In controlled motion, the host systematically visits each coordinator. This *in order* visiting pattern can delay the host's arrival at the coordinator which has tasks suitable for it and therefore causes a drop in allocation

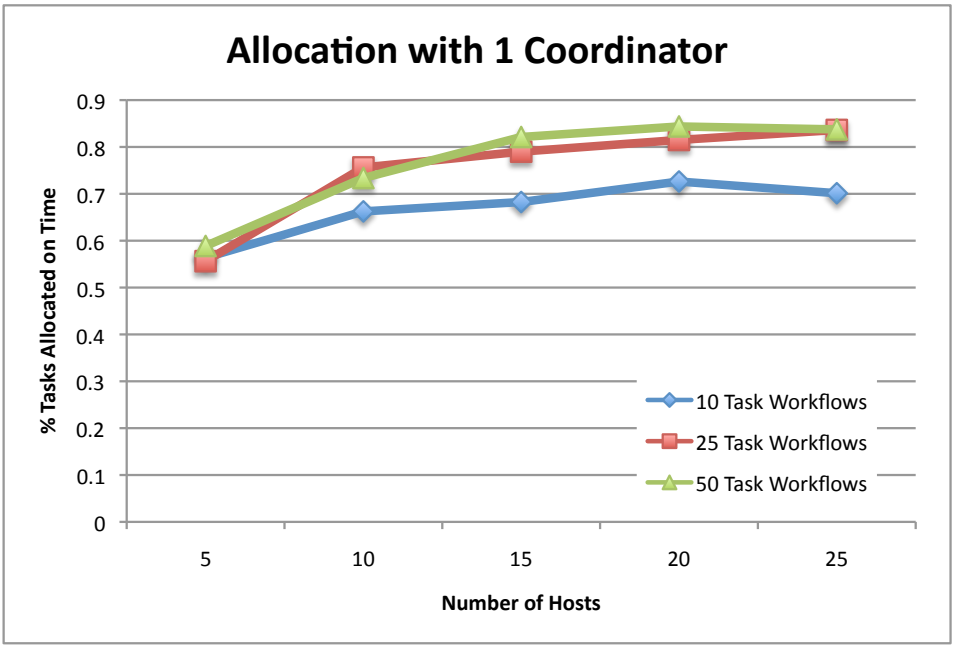

Figure A.6: On-time allocations with 1 coordinator for 10, 25, and 50 task workflows

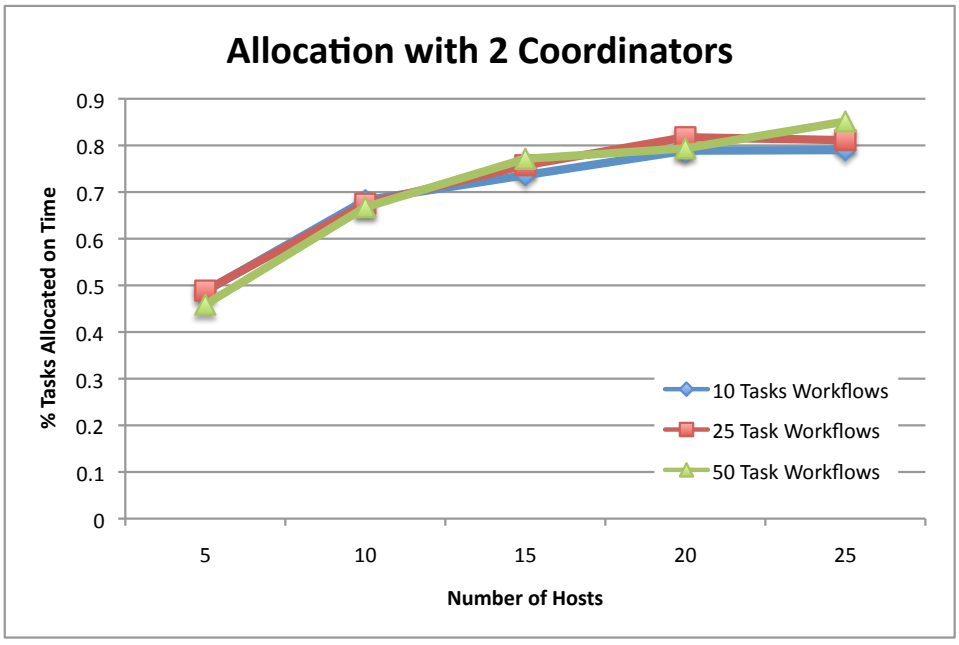

Figure A.7: On-time allocations with 2 coordinators for 10, 25, and 50 task workflows

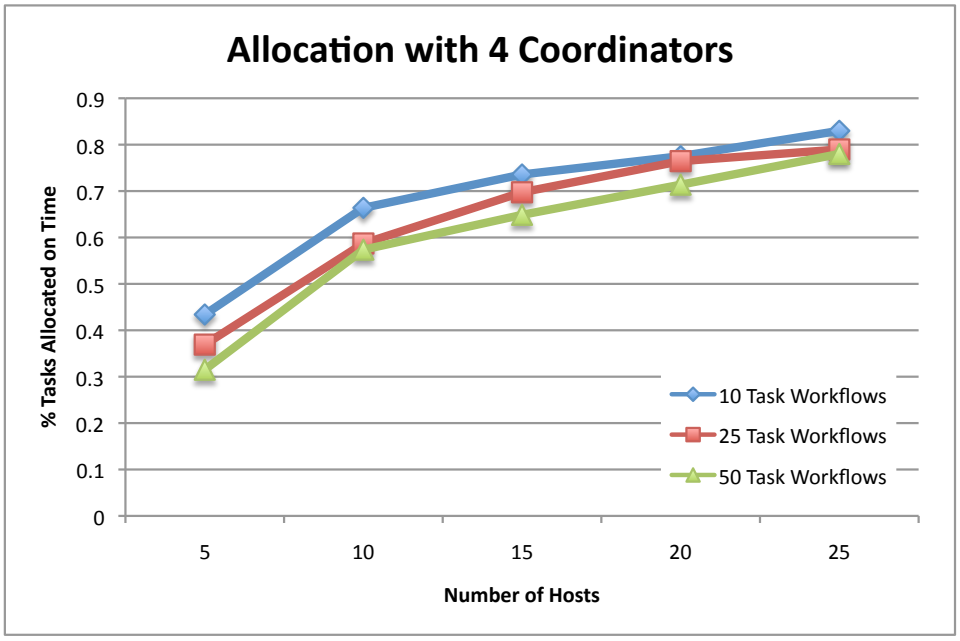

Figure A.8: On-time allocations with 4 coordinators for 10, 25, and 50 task workflows

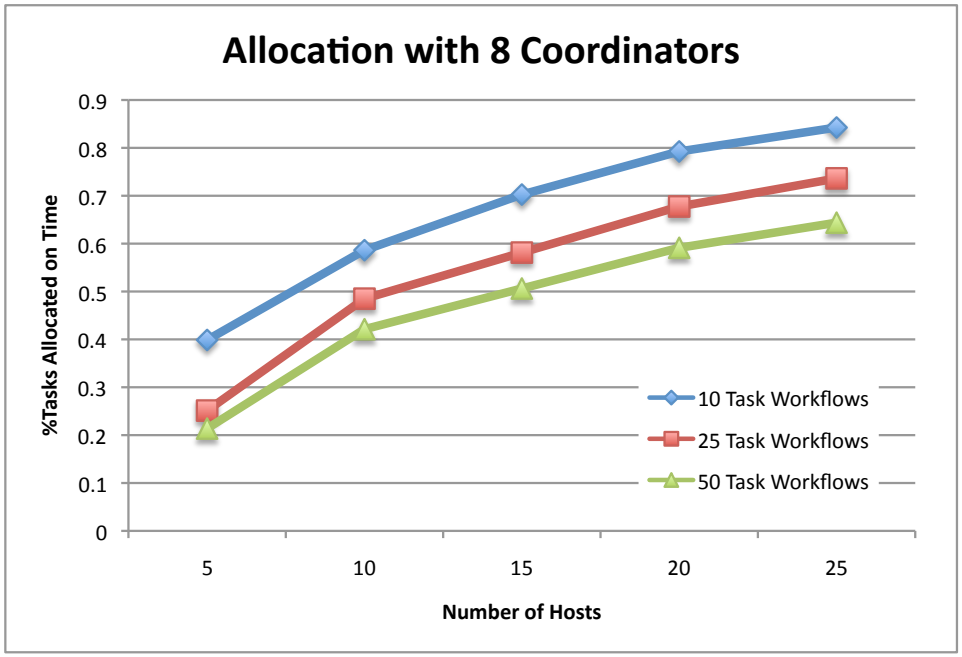

Figure A.9: On-time allocations with 8 coordinators for 10, 25, and 50 task workflows

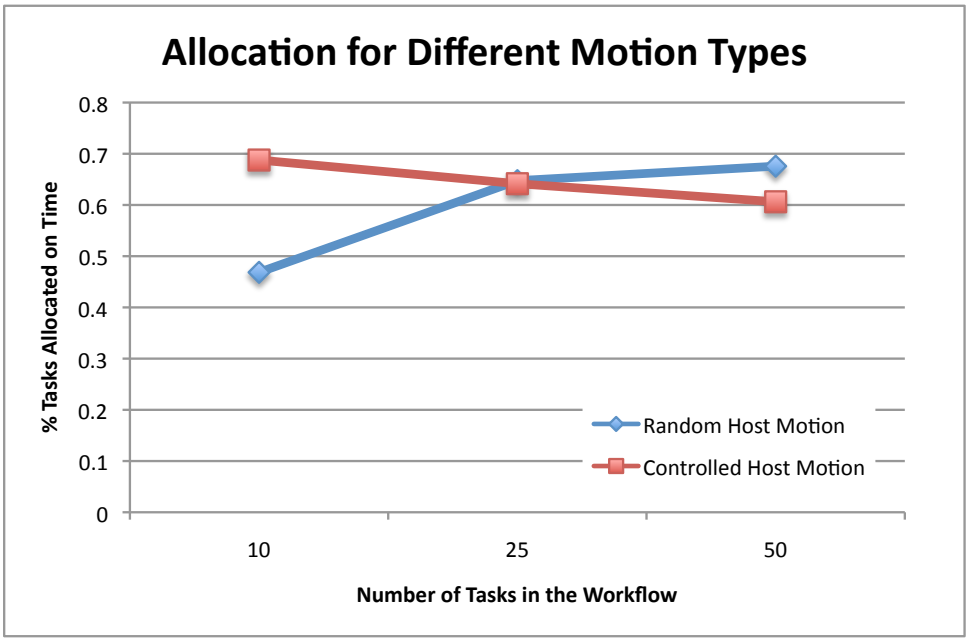

Figure A.10: On-time allocations for random and controlled motion as a function of number of tasks in the workflow

performance. Note however, that when there are fewer tasks to go around, the more disciplined controlled motion approach yields greater benefits.

Figure A.11 shows the on-time allocation numbers as a function of probability of availability of services. The controlled motion outperforms the random motion by a margin that remains constant as the probability of the service being available goes from 10% to 50%. This is because both approaches can exploit additional options to increase allocation numbers. It should be noted though that when all hosts have all services, the random motion of the hosts ceases to become a liability and outperforms the controlled scheme. Thus, while a random approach can work in the pathological case, for other cases, the controlled motion algorithm performs better.

E4: The Effect of Area Size on Allocation Performance

The experiments presented thus far were conducted using a 100 x 100 area. For this experiment, a smaller 50 x 50 area and a larger 200 x 200 area was used. Figures A.12 and A.13 show additional data for Experiment 4 in Section 6.5. The number of

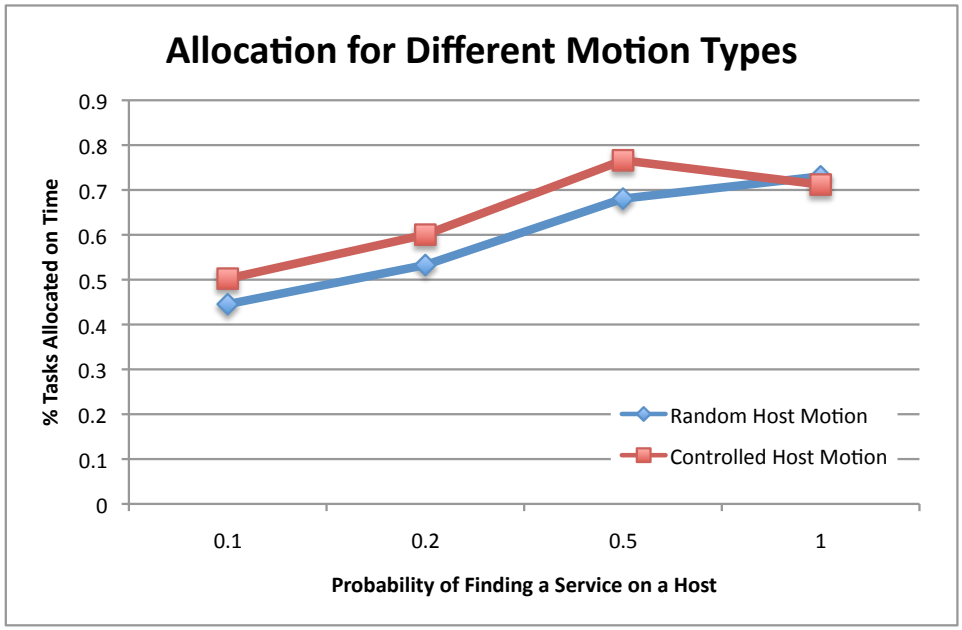

Figure A.11: On-time allocations for random and controlled motion as a function of probability of availability of services

coordinators used was 4. Figure A.12 shows the differences in allocation performance as a function of the number of tasks in the workflow. It can be seen that with controlled motion, there is not a large difference in performance when the size of the area of interest changes. In addition, performance is consistent in the larger area even with a smaller number of tasks in the workflow. Thus, the use of controlled motion is preferred when the spread of tasks in an area is sparse and discipline is required for hosts to travel to those tasks and return to coordinators for further task allocations.

Figure A.13 shows the data as a function of the probability of finding a service on a particular host. The tradeoffs between random motion and controlled motion patterns are clearly evident here. With random motion, the performance is best when every host can do every task (service probability 1.0) because any chance encounter can result in a successful task allocation. It can be argued that in such a case, an allocation process is not even necessary. However, in more realistic cases where each host can only do a subset of tasks, the controlled approach can hold its own. The advantages of the controlled motion approach can be mitigated when the area is smaller as shown

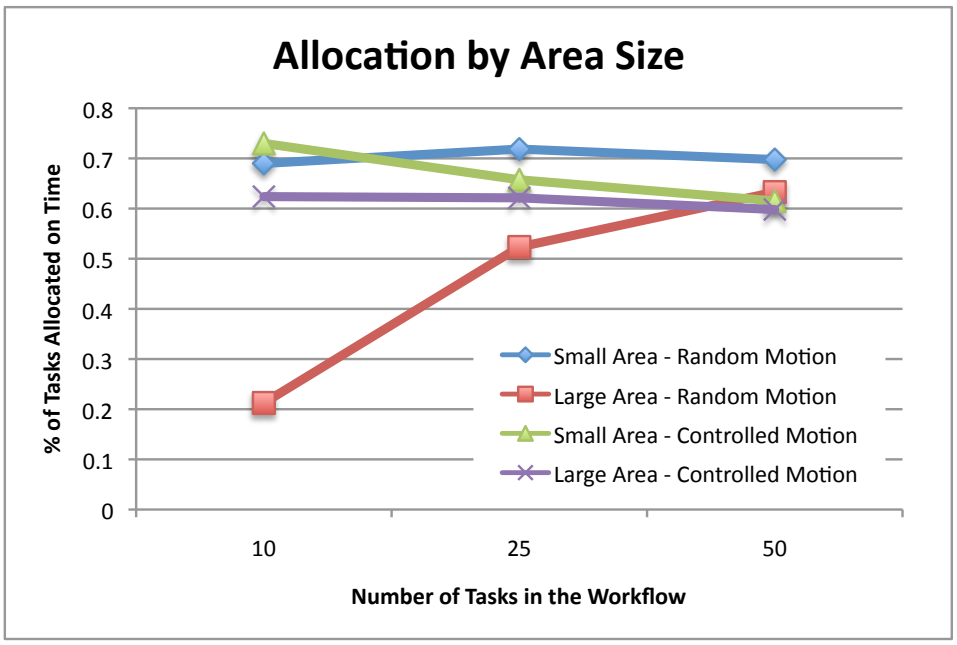

Figure A.12: On-time allocations for random and controlled motion with varying area sizes as a function of number of tasks in the workflow

but under the case of lower service probabilities and larger areas, the controlled motion approach dominates.

E5: The Use of a Geographic Workflow Partitioning Strategy

In experiment 3, tasks in the workflow were distributed across coordinators in a random manner. This distribution is done based on geographic proximity. Each coordinator is responsible for an area around it (such that no place is left uncovered by a coordinator and there are no overlaps). Tasks are distributed to the coordinator that is responsible for the location at which the task must be performed. Figures A.14 through A.18 show the data from this experiment. Figure A.14 shows the percentage of tasks allocated successfully as a function of number of hosts. Compared to the random distribution of experiment 2, the geographic distribution results in an approximately 4% increase for the 2 and 4 coordinator cases and an approximate 8% increase for the 8 coordinator case. This is due to the fact that hosts travel lesser distances to tasks from the coordinator and thus have more time available to do other tasks.

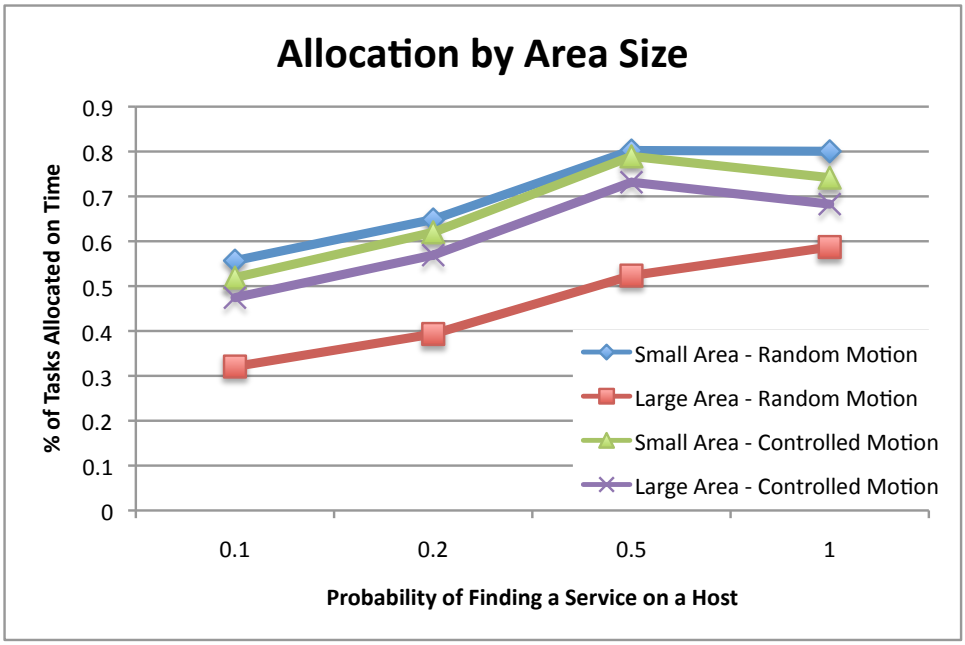

Figure A.13: On-time allocations for random and controlled motion with varying area sizes as a function of the probability of finding a service on a host

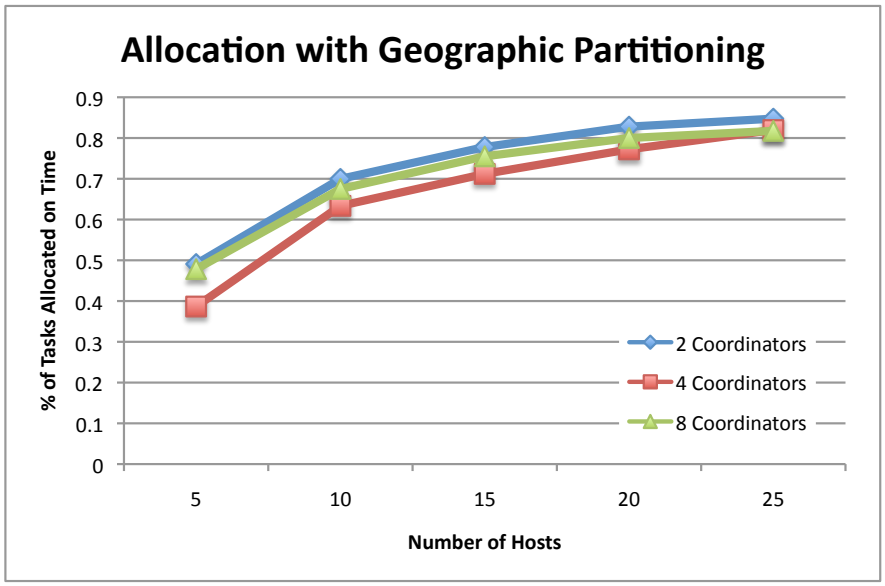

Figure A.14: On-time allocations as a function of number of hosts with geographic partitioning and different numbers of coordinators

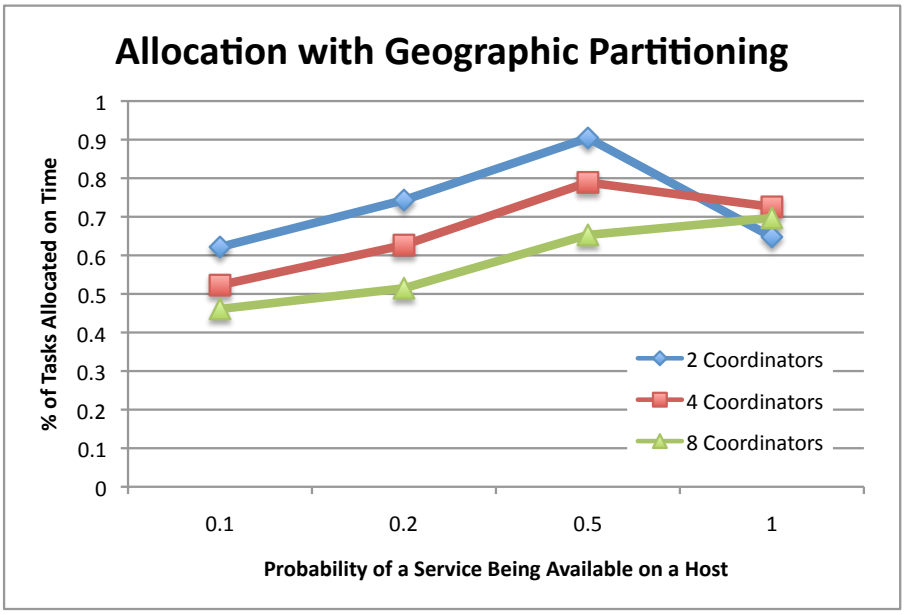

Figure A.15: On-time allocations as a function of the probability of finding a service on a host with geographic partitioning and different numbers of coordinators

Figure A.15 shows the data as a function of the probability of finding a service on a particular host. While the performance of the 2 coordinator case is slightly worse with geographic partitioning, the 4 and 8 coordinator cases do slightly better. For low numbers of coordinators, a situation arises where the balance of the tasks across coordinators is uneven and this leads to some coordinators being starved of qualified hosts resulting in a drop in overall allocation performance. With a larger number of coordinators, this problem is mitigated somewhat and the gains from having the hosts move smaller distances to their tasks overcomes any losses from the starvation of coordinators.

Figures A.16, A.17, and A.18 break down the results by the number of coordinators used and are placed here for completeness.

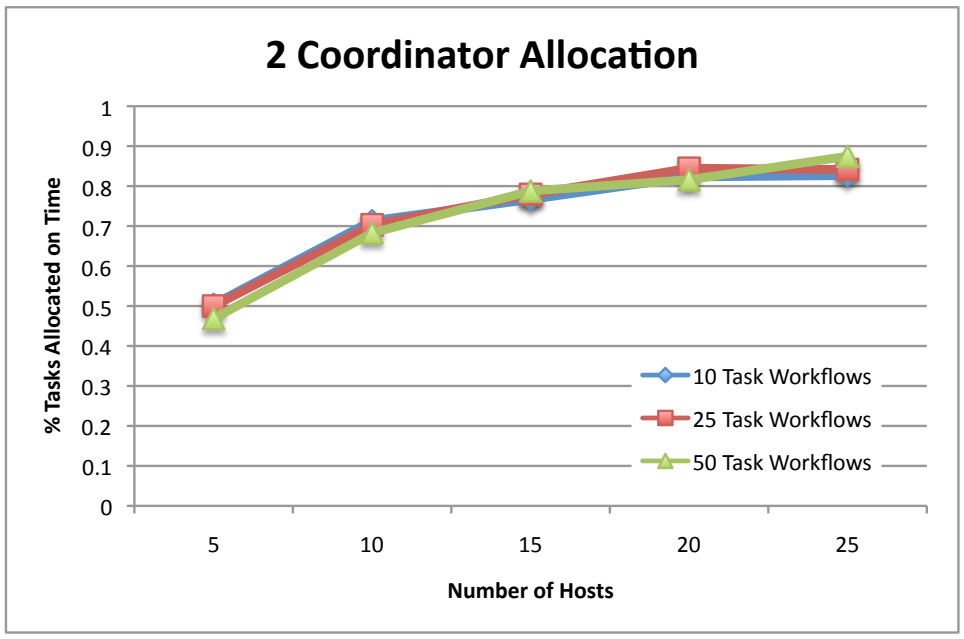

Figure A.16: On-time allocations as a function of number of hosts with geographic partitioning and 2 coordinators

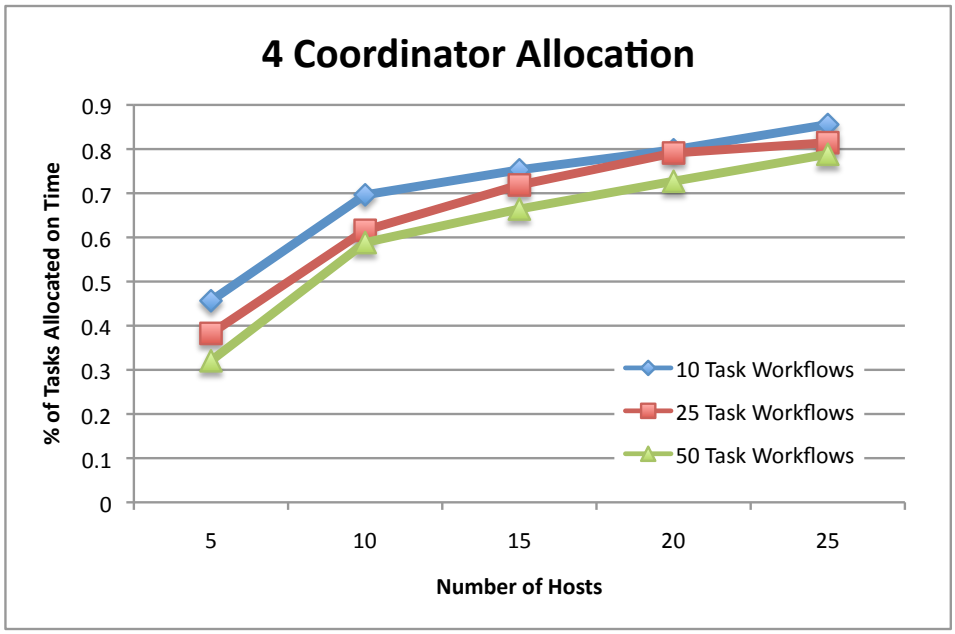

Figure A.17: On-time allocations as a function of number of hosts with geographic partitioning and 4 coordinators

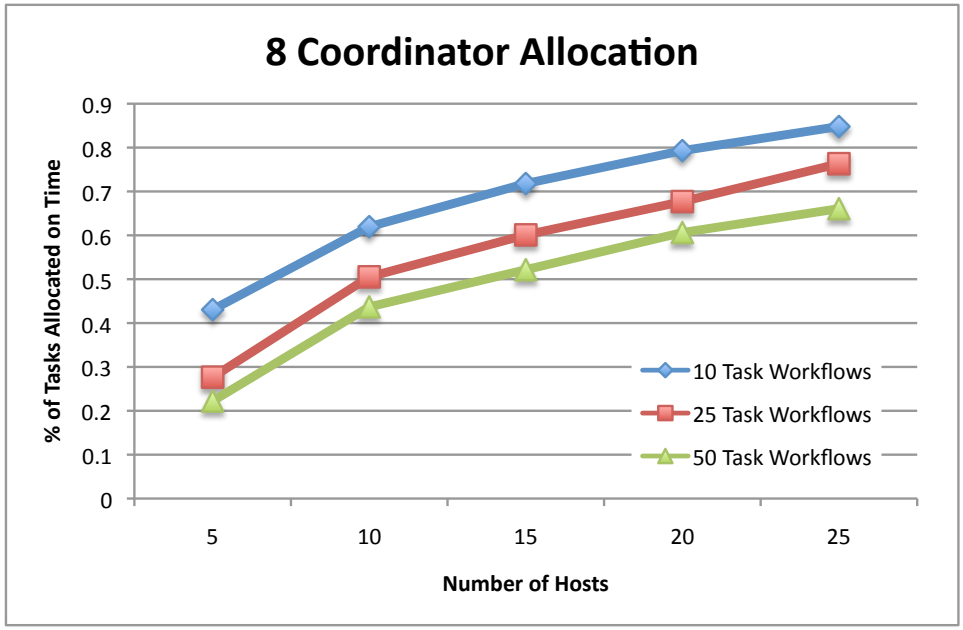

Figure A.18: On-time allocations as a function of number of hosts with geographic partitioning and 8 coordinators

Appendix B

List of Tags in the CiAN Specification

<collaboration> ... </collaboration>
The `collaboration` tags are the outermost tasks of the CiAN specification and are used to delimit a single workflow specification. Only one pair of collaboration tags can be used in a single file, thereby ensuring that each CiAN workflow appears in a separate file.

<knowledge-base> ... </knowledge-base>
The `knowledge-base` tags delimit the first of two subsections of the CiAN specification header. The knowledge base section specifies a list of knowledge base elements that the workflow depends upon for correct execution. The knowledge base referred to here is described in detail in Chapter 3.

<sensors> ... </sensors>
The `sensors` tags delimit the second of two subsections of the CiAN specification header. The sensor section specifies a list of sensors on whose values the workflow depends upon for correct execution. The software for actually interfacing with sensors is not part of the CiAN system. However, CiAN supports any text-based sensor readings.

<knowledge-var> ... </knowledge-var>
The `knowledge-var` tags delimit a single knowledge parameter that is required by the

workflow for correct operation. These tags are nested within the `knowledge-base` section. The values of these tags correspond to the name of the parameters as described in Section 3.4.3.

<sensor-var> ... </sensor-var>
The `sensor-var` tags delimit a single sensor that is required by the workflow for correct operation. These tags are nested within the `sensors` section.

<task> ... </task>
The `task` tags delimit the definition of a single task in a CiAN workflow. Every CiAN workflow is required to have at least one task in its definition. Multiple tasks can be listed in a single CiAN specification as long as their definitions are sequential and not nested.

<task-name> ... </task-name>
The `task-name` tags delimit the name of the task. This name is required to be unique in the scope of the workflow.

<loc-x> ... </loc-x>
The `loc-x` tags delimit the x-coordinate of the location at which the task must be performed. If GPS coordinates are being used, `loc-x` represents the longitude component.

<loc-y> ... </loc-y>
The `loc-y` tags delimit the y-coordinate of the location at which the task must be performed. If GPS coordinates are being used, `loc-y` represents the latitude component.

<earliest-start> ... </earliest-start>
The `earliest-start` tags delimit the earliest possible time that the associated task can start. Specifying an earliest start time does not guarantee that the task will start executing at exactly that time.

<deadline> ... </deadline>
The `deadline` tags delimit the deadline by which the task must complete. Depending on circumstances, the task may finish executing before the deadline but is guaranteed to have finished by the deadline.

<duration> ... </duration>

The `duration` tags delimit the maximum possible duration that the task can take to execute. The value of the duration may be equal to the interval between the task's earliest start time and deadline or smaller.

<inputs> ... </inputs>

The `input` tags delimit the section of the task description that describes the incoming edges, the conditions under which values along those edges are accepted, and sets of legal inputs to the task. The entire inputs section is optional and can be skipped if the task does not have any inputs (typically this is true only of the first task in the workflow).

<activity> ... </activity>

The `activity` tags delimit the section of the task description that describes the service that must be invoked to perform the task. This section also lists the names of the input variables to the service and the names of the output variables to which the service writes its return values.

<outputs> ... </outputs>

The `output` tags delimit the section of the task description that describes the outgoing edges, the conditions under which values along those edges are transmitted, and sets of legal outputs from the task. The entire outputs section is optional and can be skipped if the task does not have any outputs (typically this is true only of the last task in the workflow).

<edge> ... </edge>

The `edge` tags delimit the definition of a single incoming or outgoing edge from a task. Each edge in the workflow is defined twice. Once as an outgoing edge from the source task and once as an incoming edge to the sink task. Matching of the two is done by edge name.

<edge-name> ... </edge-name>

The `edge-name` tags delimit the name of the edge, required to be unique in the scope of the workflow.

<var> ... </var>

The `var` tags delimit the name of a variable. Depending on the context, the value enclosed within the `var` tag is the name of the variable that an edge should write the value transmitted along it to (in the case of input edges), the variable that the edge should read the value it should transmit from (in the case of output edges), or simply the name of an input or output variable to a `service`.

<partner> ... </partner>

The `partner` tags delimit the name of the task that is at the source (for input edges) and the sink (for output edges) of the edge.

<select-cond> ... </select-cond>

The `select-cond` tags delimit groups of conditions that must *all* be true in order for any value to be transmitted along the associated edge. An edge may have zero or more `select-cond` groups. Only one `select-cond` group (if multiple are specified) need have all its sub-conditions evaluate to true in order for a value to be transmitted.

<cond> ... </cond>

The `cond` tags delimit a single condition within a selection condition group.

<param> ... </param>

The `param` tags delimit the parameter that needs to be tested against a value to evaluate the condition. The value enclosed within the `param` tags must be of the form `knowledge:hostname:paramname` for parameters in the knowledge base (note the similarity of structure to the knowledge base described in Chapter 3), `sensor:sensorname` for sensor values and `edge:edgename` for edge values. The value for `paramname` must be chosen from one of the names indicated via `knowledge-var` tags in the `knowledge-base` section. Similarly the value for `sensorname` must be chosen from a value in the `sensors` section. The value of `edgename` must be chosen from the list of similar (i.e. input or output depending on the edge for which the condition is written) edges associated with the same task.

<comparator> ... </comparator>

The `comparator` tags delimit the comparator to be used in evaluating the condition.

<value> ... </value>
The `value` tags delimit the value to test the condition with.

<accept-set> ... </accept-set>
The `accept-set` tags delimit groups of edges that if all active, form acceptable inputs or outputs. An edge is considered *active* if at least one of its `select-cond` groups has all its sub-conditions evaluate to true.

<set> ... </set>
The `set` tags delimit names of edges (using the `edge-name` tag that if all active, form acceptable inputs or outputs. There may be more than one set in an `accept-set` block.

<activity> ... </activity>
The `activity` tags delimit the details of the service that must be invoked in order for the task to be completed.

<input-vars> ... </input-vars>
The `input-vars` tags delimit the set of names of input variables to the services. Individual variable names are enclosed within `var` tags within the `input-vars` section.

<service> ... </service>
The `service` tags delimit the name of the service that must be invoked in order for the task to be completed.

<method> ... </method>
The `method` tags delimit the name of the method on the service that must be invoked in order for the task to be completed.

<output-vars> ... </output-vars>
The `output-vars` tags delimit the set of names of output variables that capture the return values from a service execution. Individual variable names are enclosed within `var` tags within the `output-vars` section.

<date> ... </date>
The `date` tags delimit the date and is used within the `earliest-start` and `deadline` tags.

<time> ... </time>

The `time` tags delimit the time and is used within the `earliest-start` and `deadline` tags.

Appendix C

Screenshots of a Demo CiAN Application

A collaborative application that we have discussed at length in various chapters of this document is based on a workflow for cooperatively authoring an academic paper. This chapter presents a selection of screenshots (Figures C.1, C.2, C.3, C.4, and C.5) from this application to illustrate its functionality.

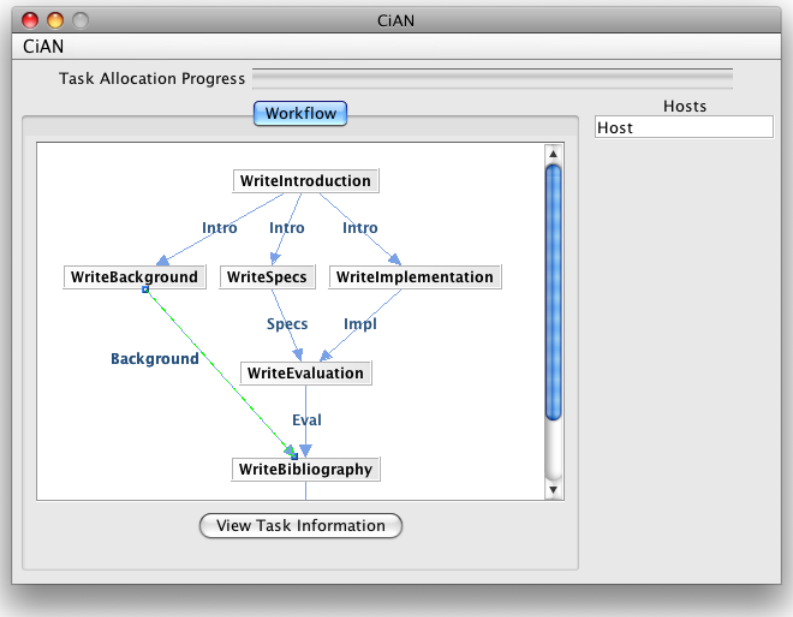

Figure C.1: Graphical representation of the workflow. Not neighbor list on the right

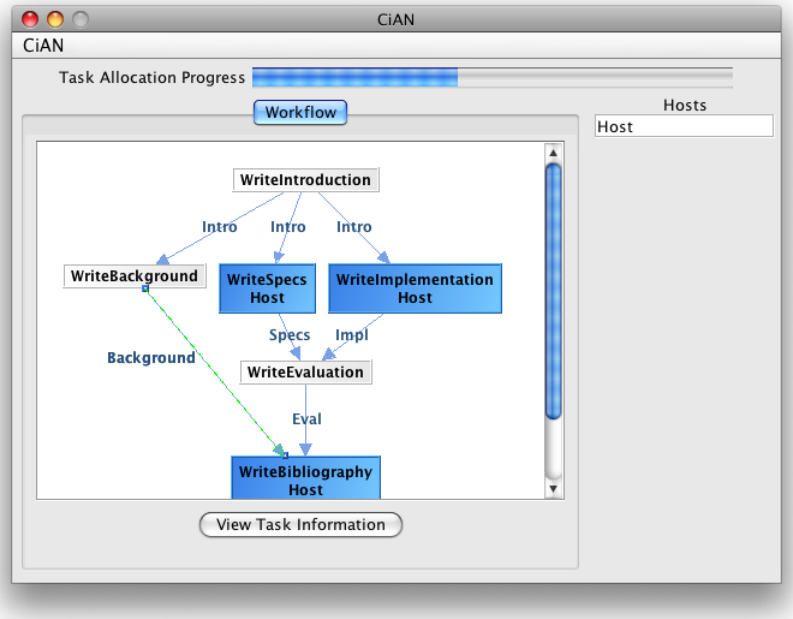

Figure C.2: Allocation in progress - highlighted tasks are allocated

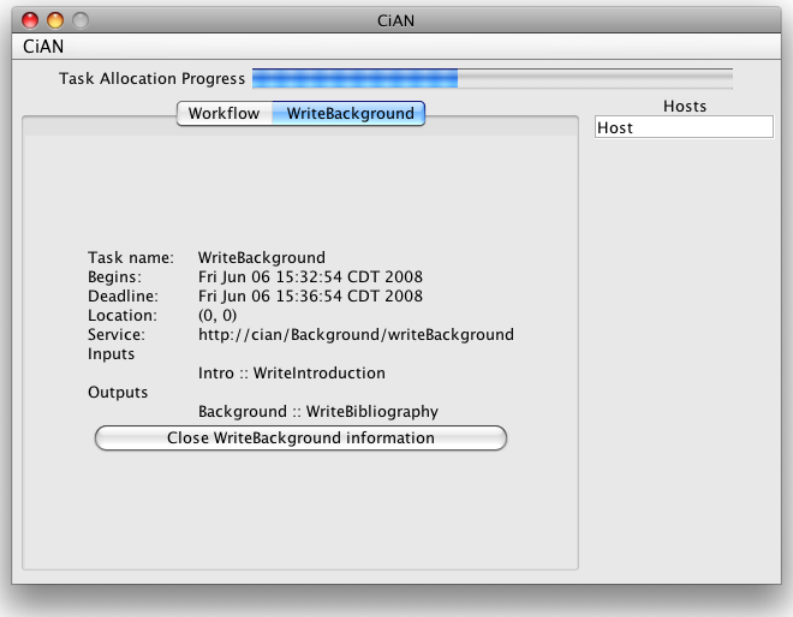

Figure C.3: Detail of a task

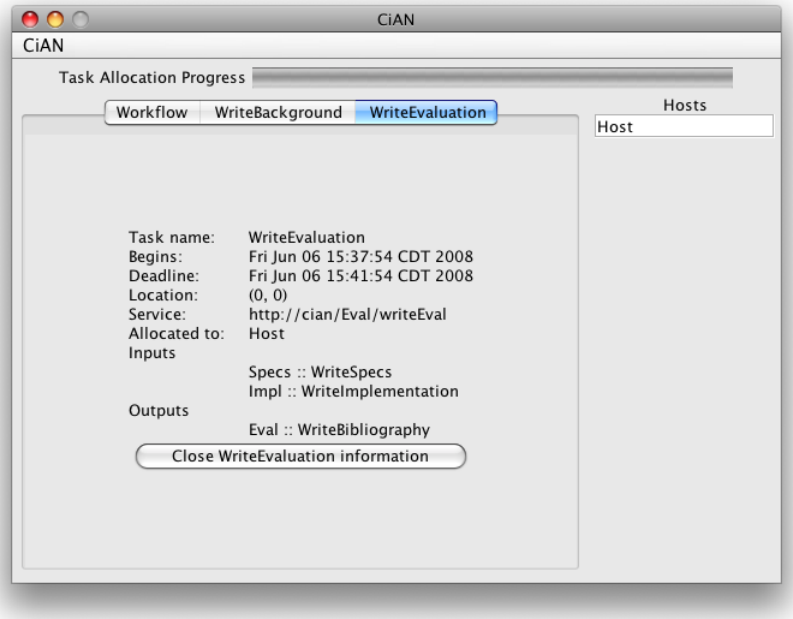

Figure C.4: Detail of a task with completed allocation indicator

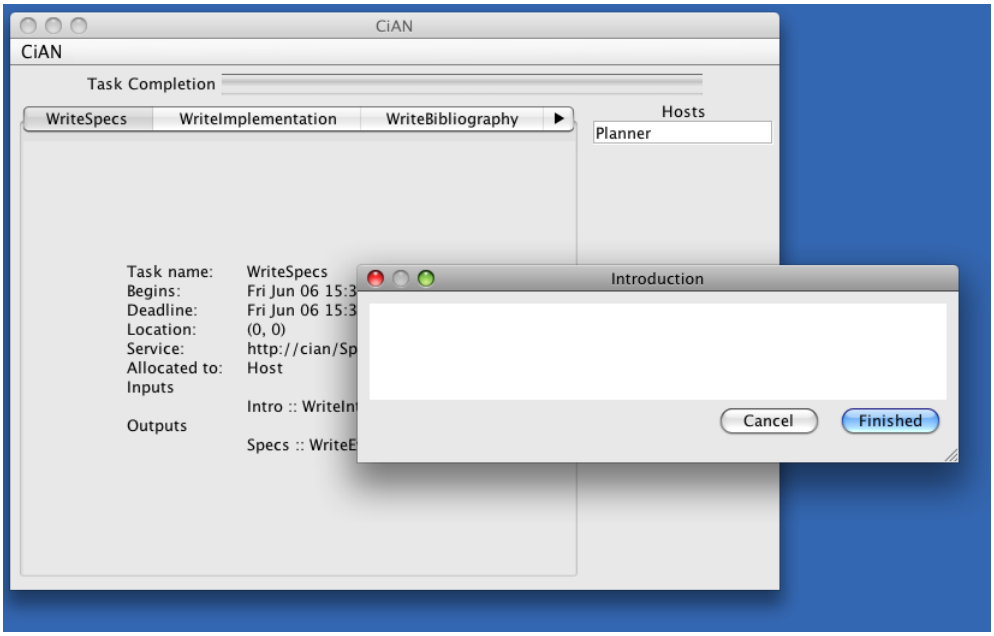

Figure C.5: Task execution: pop up window is for entering text for the introduction section

Appendix D

CiAN Specification Support for Workflow Patterns

According to van der Aalst [106], all workflows are a composition of a certain set of basic patterns. As mentioned in Chapters 4 and 5, CiAN provides support for all basic control flow and advanced synchronization patterns via its use of selection conditions and accept sets. This appendix outlines the code that must be written to create each pattern. For brevity, only the `input` and `output` sections of a task specification are shown as these are the only ones that are needed to illustrate the implementation of the patterns. The pattern numbers and names are taken from [106].

Pattern 1: Sequence

```
<task>
    <outputs>
        <edge><edge-name> Edge1 < /edge-name>< /edge>
        <accept-set>
                <set><edge-name> Edge1 < /edge-name>< /set>
        < /accept-set>
    < /outputs>
< /task>
```

Pattern 2: Parallel Split

```
<task>
    <outputs>
        <edge><edge-name> Edge1 < /edge-name>< /edge>
        <edge><edge-name> Edge2 < /edge-name>< /edge>
        <accept-set>
                <set>
                    <edge-name> Edge1 < /edge-name>
                    <edge-name> Edge2 < /edge-name>
                < /set>
        < /accept-set>
    < /outputs>
```

```
< /task>
```

Pattern 3: Synchronization

```
<task>
    <inputs>
        <edge><edge-name> Edge1 < /edge-name>< /edge>
        <edge><edge-name> Edge2 < /edge-name>< /edge>
        <accept-set>
                <set>
                    <edge-name> Edge1 < /edge-name>
                    <edge-name> Edge2 < /edge-name>
                < /set>
        < /accept-set>
    < /inputs>
< /task>
```

Pattern 4: Exclusive Choice

```
<task>
    <inputs>
        <edge>
                <edge-name> Edge1 < /edge-name>
                <select-cond>
                    <cond>
                        <param>ParamName< /param>
                        <comparator>==< /comparator>
                        <value>TestValue< /value>
                    < /cond>
                < /select-cond>
        < /edge>
        <edge>
                <edge-name> Edge2 < /edge-name>
                <select-cond>
                    <cond>
                        <param>ParamName< /param>
                        <comparator>!=< /comparator>
                        <value>TestValue< /value>
                    < /cond>
                < /select-cond>
        < /edge>
        <accept-set>
                <set><edge-name> Edge1 < /edge-name>< /set>
                <set><edge-name> Edge2 < /edge-name>< /set>
        < /accept-set>
    < /inputs>
< /task>
```

Pattern 5: Simple Merge

```
<task>
    <inputs>
        <edge><edge-name> Edge1 < /edge-name>< /edge>
        <edge><edge-name> Edge2 < /edge-name>< /edge>
        <accept-set>
                <set><edge-name> Edge1 < /edge-name>< /set>
```

```
            <set><edge-name> Edge2 < /edge-name>< /set>
        < /accept-set>
    < /inputs>
< /task>
```

Pattern 6: Multi Choice

```
<task>
    <outputs>
        <edge>
            <edge-name> Edge1 < /edge-name>
            <select-cond>
                <cond>
                    <param>ParamName< /param>
                    <comparator>==< /comparator>
                    <value>TestValue< /value>
                < /cond>
            < /select-cond>
        < /edge>
        <edge>
            <edge-name> Edge2 < /edge-name>
            <select-cond>
                <cond>
                    <param>ParamName< /param>
                    <comparator>!=< /comparator>
                    <value>TestValue< /value>
                < /cond>
            < /select-cond>
        < /edge>
        <accept-set>
            <set><edge-name> Edge1 < /edge-name>< /set>
            <set><edge-name> Edge2 < /edge-name>< /set>
        < /accept-set>
    < /outputs>
< /task>
```

Pattern 7: Synchronizing Merge

The decision on when to merge is decided by a timeout. If within a certain amount of time of the first input arriving, additional inputs are not detected, the synchronization moves ahead.

```
<task>
    <inputs>
        <edge>
            <edge-name> Edge1 < /edge-name>
            <select-cond>
                <cond>
                    <param>ParamName< /param>
                    <comparator>==< /comparator>
                    <value>TestValue< /value>
                < /cond>
            < /select-cond>
        < /edge>
        <edge>
            <edge-name> Edge2 < /edge-name>
            <select-cond>
```

```
                    <cond>
                        <param>ParamName< /param>
                        <comparator>!=< /comparator>
                        <value>TestValue< /value>
                    < /cond>
                < /select-cond>
        < /edge>
        <accept-set>
                <set><edge-name> Edge1 < /edge-name>< /set>
                <set><edge-name> Edge2 < /edge-name>< /set>
                <set>
                    <edge-name> Edge1 < /edge-name>
                    <edge-name> Edge2 < /edge-name>
                < /set>
        < /accept-set>
    < /inputs>
< /task>
```

Pattern 8: Multi Merge

Since CiAN does not support multiple cases, the multi merge pattern simplifies to the simple merge pattern.

```
<task>
    <inputs>
        <edge><edge-name> Edge1 < /edge-name>< /edge>
        <edge><edge-name> Edge2 < /edge-name>< /edge>
        <accept-set>
                <set><edge-name> Edge1 < /edge-name>< /set>
                <set><edge-name> Edge2 < /edge-name>< /set>
        < /accept-set>
    < /inputs>
< /task>
```

Pattern 9: Discriminator

The CiAN discriminator is automatically reset for every case that is run.

```
<task>
    <inputs>
        <edge>
                <edge-name> Edge1 < /edge-name>
                <select-cond>
                    <cond>
                        <param>ParamName< /param>
                        <comparator>==< /comparator>
                        <value>TestValue< /value>
                    < /cond>
                < /select-cond>
        < /edge>
        <edge>
                <edge-name> Edge2 < /edge-name>
                <select-cond>
                    <cond>
                        <param>ParamName< /param>
                        <comparator>!=< /comparator>
                        <value>TestValue< /value>
```

```
                    < /cond>
                < /select-cond>
        < /edge>
        <accept-set>
                <set><edge-name> Edge1 < /edge-name>< /set>
                <set><edge-name> Edge2 < /edge-name>< /set>
        < /accept-set>
    < /inputs>
< /task>
```

Appendix E

Qualitative Code Comparison for Cases

This section shows the code for the three cases outlined in Chapter 4 using BPEL, YAWL, and CiAN. For brevity, parts of the code not crucial to the illustration have not been included.

E.1 Case 1: Writing a Paper

E.1.1 CiAN Code for Case 1

```
<collaboration>
    <task>
        <task-name>WriteIntro< /task-name>
        <outputs>
                <edge>
                    <edge-name>IntroToBG< /edge-name>
                    <var>IntroText< /var>
                    <partner>writeBG< /partner>
                < /edge>
                <edge>
                    <edge-name>IntroToBody< /edge-name>
                    <var>IntroText< /var>
                    <partner>writeBody< /partner>
                < /edge>
                <edge>
                    <edge-name>IntroToImpl< /edge-name>
                    <var>IntroText< /var>
                    <partner>writeImpl< /partner>
                < /edge>
                <accept-set>
                    <set>
                        <edge-name>IntroToBG< /edge-name>
                        <edge-name>IntroToBody< /edge-name>
                        <edge-name>IntroToImpl< /edge-name>
                    < /set>
                < /accept-set≫ /outputs>
    < /task>
```

```
<task>
    <task-name>WriteBG< /task-name>
    <inputs>
            <edge>
                <edge-name>IntroToBG< /edge-name>
                <var>IntroText< /var>
                <partner>writeIntro< /partner>
            < /edge>
            <accept-set>
                <set>
                    <edge-name>IntroToBG< /edge-name>
                < /set>
            < /accept-set> /inputs>
    <outputs>
            <edge>
                <edge-name>BGToEval< /edge-name>
                <var>IntroBGText< /var>
                <partner>writeEval< /partner>
            < /edge>
            <accept-set>
                <set>
                    <edge-name>BGToEval< /edge-name>
                < /set>
            < /accept-set> /outputs>
< /task>

<task>
    <task-name>WriteBody< /task-name>
    <inputs>
            <edge>
                <edge-name>IntroToBody< /edge-name>
                <var>IntroText< /var>
                <partner>writeIntro< /partner>
            < /edge>
            <accept-set>
                <set>
                    <edge-name>IntroToBody< /edge-name>
                < /set>
            < /accept-set> /inputs>
    <outputs>
            <edge>
                <edge-name>BodyToEval< /edge-name>
                <var>IntroBodyText< /var>
                <partner>writeEval< /partner>
            < /edge>
            <accept-set>
                <set>
                    <edge-name>BodyToEval< /edge-name>
                < /set>
            < /accept-set> /outputs>
< /task>

<task>
    <task-name>WriteImpl< /task-name>
    <inputs>
            <edge>
                <edge-name>IntroToImpl< /edge-name>
                <var>IntroText< /var>
                <partner>writeIntro< /partner>
            < /edge>
            <accept-set>
                <set>
                    <edge-name>IntroToImpl< /edge-name>
                < /set>
            < /accept-set> /inputs>
    <outputs>
            <edge>
                <edge-name>ImplToEval< /edge-name>
                <var>IntroImplText< /var>
                <partner>writeEval< /partner>
```

```
                < /edge>
                <accept-set>
                    <set>
                        <edge-name>ImplToEval< /edge-name>
                    < /set>
                < /accept-set> /outputs>
    < /task>

    <task>
        <task-name>WriteEval< /task-name>
        <inputs>
                <edge>
                    <edge-name>BGToEval< /edge-name>
                    <var>IntroBGText< /var>
                    <partner>writeBG< /partner>
                < /edge>
                <edge>
                    <edge-name>BodyToEval< /edge-name>
                    <var>IntroBodyText< /var>
                    <partner>writeBody< /partner>
                < /edge>
                <edge>
                    <edge-name>ImplToEval< /edge-name>
                    <var>IntroImplText< /var>
                    <partner>writeImpl< /partner>
                < /edge>
                <accept-set>
                    <set>
                        <edge-name>BGToEval< /edge-name>
                        <edge-name>BodyToEval< /edge-name>
                        <edge-name>ImplToEval< /edge-name>
                    < /set>
                < /accept-set> /inputs>
        <outputs>
                <edge>
                    <edge-name>EvalToConc< /edge-name>
                    <var>PaperText< /var>
                    <partner>writeConc< /partner>
                < /edge>
                <accept-set>
                    <set>
                        <edge-name>EvalToConc< /edge-name>
                    < /set>
                < /accept-set> /outputs>
    < /task>

    <task>
        <task-name>WriteConc< /task-name>
        <inputs>
                <edge>
                    <edge-name>EvalToConc< /edge-name>
                    <var>PaperText< /var>
                    <partner>writeEval< /partner>
                < /edge>
                <accept-set>
                    <set>
                        <edge-name>EvalToConc< /edge-name>
                    < /set>
                < /accept-set> /inputs>
    < /task>
< /collaboration>
```

E.1.2 BPEL Code for Case 1

```
<process>
    <name>WritePaper< /name>
    <sequence>
        <invoke>writeIntro< /invoke>
        <flow>
                <invoke>writeBG< /invoke>
                <invoke>writeBody< /invoke>
                <invoke>writeImpl< /invoke>
        < /flow>
        <invoke>writeEval< /invoke>
        <invoke>writeConc< /invoke>
    < /sequence>
< /process>
```

E.1.3 YAWL Code for Case 1

```
<processControlElements>
    <task>
        <id>writeIntro< /id>
        <flowsInto><nextElementRef><id>writeBG< /id>< /nextElementRef>< /flowsInto>
        <flowsInto><nextElementRef><id>writeBody< /id>< /nextElementRef>< /flowsInto>
        <flowsInto><nextElementRef><id>writeImpl< /id>< /nextElementRef>< /flowsInto>
        <join code>and< /join code>
        <split code>and< /split code>
    < /task>

    <task>
        <id>writeBG< /id>
        <flowsInto><nextElementRef><id>writeEval< /id>< /nextElementRef>< /flowsInto>
        <join code>and< /join code>
        <split code>and< /split code>
    < /task>

    <task>
        <id>writeBody< /id>
        <flowsInto><nextElementRef><id>writeEval< /id>< /nextElementRef>< /flowsInto>
        <join code>and< /join code>
        <split code>and< /split code>
    < /task>

    <task>
        <id>writeImpl< /id>
        <flowsInto><nextElementRef><id>writeEval< /id>< /nextElementRef>< /flowsInto>
        <join code>and< /join code>
        <split code>and< /split code>
    < /task>

    <task>
        <id>writeEval< /id>
        <flowsInto><nextElementRef><id>writeConc< /id>< /nextElementRef>< /flowsInto>
        <join code>and< /join code>
        <split code>and< /split code>
    < /task>

    <task>
        <id>writeConc< /id>
        <join code>and< /join code>
        <split code>and< /split code>
    < /task>
```

```
< /processControlElements>
```

E.2 Case 2: Checking Out From an Online Store

E.2.1 CiAN Code for Case 2

```
<collaboration>
    <task>
        <task-name>PlaceOrder< /task-name>
        <outputs>
                <edge>
                    <edge-name>OrderToShip< /edge-name>
                    <var>OrderInfo< /var>
                    <partner>CalcShipCost< /partner>
                < /edge>
                <edge>
                    <edge-name>OrderToTax< /edge-name>
                    <var>OrderInfo< /var>
                    <partner>CalcTax< /partner>
                < /edge>
                <accept-set>
                    <set>
                        <edge-name>OrderToShip< /edge-name>
                        <edge-name>OrderToTax< /edge-name>
                    < /set>
                < /accept-set> /outputs>
    < /task>

    <task>
        <task-name>CalcShipCost< /task-name>
        <inputs>
                <edge>
                    <edge-name>OrderToShip< /edge-name>
                    <var>OrderInfo< /var>
                    <partner>PlaceOrder< /partner>
                < /edge>
                <accept-set>
                    <set>
                        <edge-name>OrderToShip< /edge-name>
                    < /set>
                < /accept-set> /inputs>
        <outputs>
                <edge>
                    <edge-name>ShipToPrint< /edge-name>
                    <var>ShipCost< /var>
                    <partner>PrintBill< /partner>
                < /edge>
                <accept-set>
                    <set>
                        <edge-name>ShipToPrint< /edge-name>
                    < /set>
                < /accept-set> /outputs>
    < /task>

    <task>
        <task-name>CalcTax< /task-name>
        <inputs>
                <edge>
                    <edge-name>OrderToTax< /edge-name>
                    <var>OrderInfo< /var>
                    <partner>PlaceOrder< /partner>
```

```
            < /edge>
            <accept-set>
                <set>
                    <edge-name>OrderToTax< /edge-name>
                < /set>
            < /accept-set≫ /inputs>
    <outputs>
            <edge>
                <edge-name>TaxToPrint< /edge-name>
                <var>TaxAmount< /var>
                <partner>PrintBill< /partner>
            < /edge>
            <accept-set>
                <set>
                    <edge-name>TaxToPrint< /edge-name>
                < /set>
            < /accept-set≫ /outputs>
< /task>

<task>
    <task-name>PrintBill< /task-name>
    <inputs>
            <edge>
                <edge-name>ShipToPrint< /edge-name>
                <var>ShipCost< /var>
                <partner>CalcShipCost< /partner>
            < /edge>
            <edge>
                <edge-name>TaxToPrint< /edge-name>
                <var>TaxAmount< /var>
                <partner>CalcTax< /partner>
            < /edge>
            <accept-set>
                <set>
                    <edge-name>ShipToPrint< /edge-name>
                    <edge-name>TaxToPrint< /edge-name>
                < /set>
            < /accept-set≫ /inputs>
    <outputs>
            <edge>
                <edge-name>PrintToCoupon< /edge-name>
                <var>OrderTotal< /var>
                <partner>PrintCoupon< /partner>
                <select-cond>
                    <cond>
                        <param>var:OrderTotal< /param>
                        <comparator>≥< /comparator>
                        <value>100< /value>
                    < /cond>
                < /select-cond>
            < /edge>
            <edge>
                <edge-name>PrintToDone< /edge-name>
                <var>OrderTotal< /var>
                <partner>Done< /partner>
                <select-cond>
                    <cond>
                        <param>var:OrderTotal< /param>
                        <comparator><< /comparator>
                        <value>100< /value>
                    < /cond>
                < /select-cond>
            < /edge>
            <accept-set>
                <set>
                    <edge-name>PrintToCoupon< /edge-name>
                < /set>
                <set>
                    <edge-name>PrintToDone< /edge-name>
                < /set>
            < /accept-set≫ /outputs>
```

```
    < /task>

    <task>
        <task-name>PrintCoupon< /task-name>
        <inputs>
                <edge>
                    <edge-name>PrintToCoupon< /edge-name>
                    <var>OrderTotal< /var>
                    <partner>PrintBill< /partner>
                < /edge>
                <accept-set>
                    <set>
                        <edge-name>PrintToCoupon< /edge-name>
                    < /set>
                < /accept-set≫ /inputs>
        <outputs>
                <edge>
                    <edge-name>CouponToDone< /edge-name>
                    <var>CouponValue< /var>
                    <partner>Done< /partner>
                < /edge>
                <accept-set>
                    <set>
                        <edge-name>CouponToDone< /edge-name>
                    < /set>
                < /accept-set≫ /outputs>
    < /task>

    <task>
        <task-name>Done< /task-name>
        <inputs>
                <edge>
                    <edge-name>CouponToDone< /edge-name>
                    <var>CouponValue< /var>
                    <partner>PrintCoupon< /partner>
                < /edge>
                <edge>
                    <edge-name>PrintToDone< /edge-name>
                    <var>OrderTotal< /var>
                    <partner>PrintBill< /partner>
                < /edge>
                <accept-set>
                    <set>
                        <edge-name>CouponToDone< /edge-name>
                    < /set>
                    <set>
                        <edge-name>PrintToDone< /edge-name>
                    < /set>
                < /accept-set≫ /inputs>
    < /task>
< /collaboration>
```

E.2.2 BPEL Code for Case 2

```
<process>
    <name>CheckOutFromStore< /name>
    <sequence>
        <invoke>PlaceOrder< /invoke>
        <flow>
                <invoke>CalcShipCost< /invoke>
                <invoke>CalcTax< /invoke>
        < /flow>
        <invoke>PrintBill< /invoke>
```

```
        <switch>
                <case condition="total ≥ 100">
                    <invoke>IncludeCoupon< /invoke>
                < /case>
                <otherwise>
                    <invoke>Done< /invoke>
                < /otherwise>
        < /switch>
    < /sequence>
< /process>
```

E.2.3 YAWL Code for Case 2

```
<processControlElements>
    <task>
        <id>PlaceOrder< /id>
        <flowsInto><nextElementRef><id>CalcShipCost< /id>< /nextElementRef>< /flowsInto>
        <flowsInto><nextElementRef><id>CalcTax< /id>< /nextElementRef>< /flowsInto>
        <join code>and< /join code>
        <split code>and< /split code>
    < /task>

    <task>
        <id>CalcShipCost< /id>
        <flowsInto><nextElementRef><id>PrintBill< /id>< /nextElementRef>< /flowsInto>
        <join code>and< /join code>
        <split code>and< /split code>
    < /task>

    <task>
        <id>CalcTax< /id>
        <flowsInto><nextElementRef><id>PrintBill< /id>< /nextElementRef>< /flowsInto>
        <join code>and< /join code>
        <split code>and< /split code>
    < /task>

    <task>
        <id>printBill< /id>
        <flowsInto><nextElementRef><id>CouponCondition< /id>< /nextElementRef>< /flowsInto>
        <flowsInto><nextElementRef><id>Done< /id>< /nextElementRef>< /flowsInto>
        <join code>and< /join code>
        <split code>and< /split code>
    < /task>

    <condition>
        <name>CouponCondition< /name>
        <flowsInto><nextElementRef><id>PrintBill< /id>< /nextElementRef>< /flowsInto>
    < /condition>

    <task>
        <id>printBill< /id>
        <flowsInto><nextElementRef><id>Done< /id>< /nextElementRef>< /flowsInto>
        <join code>and< /join code>
        <split code>and< /split code>
    < /task>

    <task>
        <id>Done< /id>
        <join code>or< /join code>
        <split code>and< /split code>
    < /task>
< /processControlElements>
```

E.3 Case 3: Soil Sample Analysis

E.3.1 CiAN Code for Case 3

```
<collaboration>
    <task>
        <task-name>TakeSoilSample< /task-name>
        <earliest-start>1:00PM< /earliest-start>
        <deadline>3:00PM< /deadline>
        <duration>1:00hr< /duration>
        <outputs>
                <edge>
                    <edge-name>SampleToBasicAnalysis< /edge-name>
                    <var>SampleData< /var>
                    <partner>DoBasicAnalysis< /partner>
                < /edge>
                <accept-set>
                    <set>
                        <edge-name>SampleToBasicAnalysis< /edge-name>
                    < /set>
                < /accept-set> /outputs>
    < /task>

    <task>
        <task-name>DoBasicAnalyis< /task-name>
        <inputs>
                <edge>
                    <edge-name>SampleToBasicAnalyis< /edge-name>
                    <var>SampleData< /var>
                    <partner>TakeSoilSample< /partner>
                < /edge>
                <accept-set>
                    <set>
                        <edge-name>SampleToBasicAnalyis< /edge-name>
                    < /set>
                < /accept-set> /inputs>
        <outputs>
                <edge>
                    <edge-name>BasicAnalysisToSecondAnalysis< /edge-name>
                    <var>SampleData< /var>
                    <partner>SecondAnalysis< /partner>
                    <select-cond>
                        <cond>
                            <param>var:SampleData< /param>
                            <comparator>>< /comparator>
                            <value>50< /value>
                        < /cond>
                    < /select-cond>
                < /edge>
                <edge>
                    <edge-name>BasicAnalysisToPrepReport< /edge-name>
                    <var>SampleData< /var>
                    <partner>PrepareReport< /partner>
                    <select-cond>
                        <cond>
                            <param>var:SampleData< /param>
                            <comparator><< /comparator>
                            <value>50< /value>
                        < /cond>
                    < /select-cond>
                < /edge>
                <accept-set>
                    <set>
                        <edge-name>BasicAnalysisToPrepReport< /edge-name>
                    < /set>
                    <set>
                        <edge-name>CouponToDone< /edge-name>
```

```
                < /set>
            < /accept-set≫ /outputs>
< /task>

<task>
    <task-name>SecondAnalysis< /task-name>
    <inputs>
            <edge>
                <edge-name>BasicAnalysisToSecondAnalysis< /edge-name>
                <var>SampleData< /var>
                <partner>DoBasicAnalysis< /partner>
            < /edge>
            <accept-set>
                <set>
                    <edge-name>BasicAnalysisToSecondAnalysis< /edge-name>
                < /set>
            < /accept-set≫ /inputs>
    <outputs>
            <edge>
                <edge-name>SecondAnalysisToConfirmAnalysis< /edge-name>
                <var>SampleData< /var>
                <partner>ConfirmAnalysis< /partner>
                <select-cond>
                    <cond>
                        <param>var:SampleData< /param>
                        <comparator>>< /comparator>
                        <value>50< /value>
                    < /cond>
                    <cond>
                        <param>var:SampleData< /param>
                        <comparator><< /comparator>
                        <value>100< /value>
                    < /cond>
                < /select-cond>
            < /edge>
            <edge>
                <edge-name>SecondAnalysisToCall911< /edge-name>
                <var>SampleData< /var>
                <partner>Call911< /partner>
                <select-cond>
                    <cond>
                        <param>var:SampleData< /param>
                        <comparator>>< /comparator>
                        <value>100< /value>
                    < /cond>
                < /select-cond>
            < /edge>
            <accept-set>
                <set>
                    <edge-name>SecondAnalysisToConfirmAnalysis< /edge-name>
                < /set>
                <set>
                    <edge-name>SecondAnalysisToCall911< /edge-name>
                < /set>
            < /accept-set≫ /outputs>
< /task>

<task>
    <task-name>ConfirmAnalysis< /task-name>
    <inputs>
            <edge>
                <edge-name>SecondAnalysisToConfirmAnalysis< /edge-name>
                <var>SampleData< /var>
                <partner>SecondAnalysis< /partner>
            < /edge>
            <accept-set>
                <set>
                    <edge-name>SecondAnalysisToConfirmAnalysis< /edge-name>
                < /set>
            < /accept-set≫ /inputs>
    <outputs>
```

```
            <edge>
                <edge-name>ConfirmAnalysisToPrepRept< /edge-name>
                <var>SampleData< /var>
                <partner>PrepareReport< /partner>
                <select-cond>
                    <cond>
                        <param>var:SampleData< /param>
                        <comparator><< /comparator>
                        <value>75< /value>
                    < /cond>
                < /select-cond>
            < /edge>
            <edge>
                <edge-name>ConfirmAnalysisToCall911< /edge-name>
                <var>SampleData< /var>
                <partner>Call911< /partner>
                <select-cond>
                    <cond>
                        <param>var:SampleData< /param>
                        <comparator>>< /comparator>
                        <value>75< /value>
                    < /cond>
                < /select-cond>
            < /edge>
            <accept-set>
                <set>
                    <edge-name>ConfirmAnalysisToPrepareReport< /edge-name>
                < /set>
                <set>
                    <edge-name>ConfirmAnalysisToCall911< /edge-name>
                < /set>
            < /accept-set≫ /outputs>
< /task>

<task>
    <task-name>Call911< /task-name>
    <inputs>
            <edge>
                <edge-name>SecondAnalysisToCall911< /edge-name>
                <var>SampleData< /var>
                <partner>SecondAnalysis< /partner>
            < /edge>
            <edge>
                <edge-name>ConfirmAnalysisToCall911< /edge-name>
                <var>SampleData< /var>
                <partner>ConfirmAnalysis< /partner>
            < /edge>
            <accept-set>
                <set>
                    <edge-name>SecondAnalysisToCall911< /edge-name>
                < /set>
                <set>
                    <edge-name>ConfirmAnalysisToCall911< /edge-name>
                < /set>
            < /accept-set≫ /inputs>
    <outputs>
            <edge>
                <edge-name>Call911ToPrepareReport< /edge-name>
                <var>911CallID< /var>
                <partner>Stop< /partner>
            < /edge>
            <accept-set>
                <set>
                    <edge-name>Call911ToPrepareReport< /edge-name>
                < /set>
            < /accept-set≫ /outputs>
< /task>

<task>
    <task-name>PrepareReport< /task-name>
    <inputs>
```

```
            <edge>
                <edge-name>BasicAnalysisToPrepReport< /edge-name>
                <var>SampleData< /var>
                <partner>DoBasicAnalysis< /partner>
            < /edge>
            <edge>
                <edge-name>Call911ToPrepReport< /edge-name>
                <var>SampleData< /var>
                <partner>Call911< /partner>
            < /edge>
                <edge-name>ConfirmAnalysisToPrepReport< /edge-name>
                <var>SampleData< /var>
                <partner>ConfirmAnalysis< /partner>
            < /edge>
            <accept-set>
                <set>
                    <edge-name>BasicAnalysisToPrepReport< /edge-name>
                < /set>
                <set>
                    <edge-name>Call911ToPrepReport< /edge-name>
                < /set>
                <set>
                    <edge-name>ConfirmAnalysisToPrepReport< /edge-name>
                < /set>
            < /accept-set> /inputs>
    <outputs>
            <edge>
                <edge-name>PrepareReportToGetSupervisorSignature< /edge-name>
                <var>ToxicityReport< /var>
                <partner>GetSupervisorSignature< /partner>
            < /edge>
            <accept-set>
                <set>
                    <edge-name>PrepareReportToGetSupervisorSignature< /edge-name>
                < /set>
            < /accept-set> /outputs>
< /task>

<task>
    <task-name>GetSupervisorSignature< /task-name>
    <inputs>
            <edge>
                <edge-name>PrepareReportToGetSupervisorSignature< /edge-name>
                <var>ToxicityReport< /var>
                <partner>PrepareReport< /partner>
            < /edge>
            <accept-set>
                <set>
                    <edge-name>PrepareReportToGetSupervisorSignature< /edge-name>
                < /set>
            < /accept-set> /inputs>
    <outputs>
            <edge>
                <edge-name>GetSupervisorSignatureToStop< /edge-name>
                <var>ToxicityReport< /var>
                <partner>Stop< /partner>
            < /edge>
            <accept-set>
                <set>
                    <edge-name>GetSupervisorSignatureToStop< /edge-name>
                < /set>
            < /accept-set> /outputs>
< /task>

<task>
    <task-name>Stop< /task-name>
    <inputs>
            <edge>
                <edge-name>ConfirmAnalysisToStop< /edge-name>
                <var>SampleData< /var>
                <partner>ConfirmAnalysis< /partner>
```

```
                < /edge>
                <edge>
                    <edge-name>Call911ToStop< /edge-name>
                    <var>911CallID< /var>
                    <partner>Call911< /partner>
                < /edge>
                <edge>
                    <edge-name>GetSupervisorSignatureToStop< /edge-name>
                    <var>ToxicityReport< /var>
                    <partner>GetSupervisorSignature< /partner>
                < /edge>
                <accept-set>
                    <set>
                        <edge-name>ConfirmAnalysisToStop< /edge-name>
                    < /set>
                    <set>
                        <edge-name>Call911ToStop< /edge-name>
                    < /set>
                    <set>
                        <edge-name>GetSupervisorSignatureToStop< /edge-name>
                    < /set>
                < /accept-set> /inputs>
    < /task>
< /collaboration>
```

E.3.2 BPEL Code for Case 3

The soil sample analysis case cannot be implemented in BPEL. This is because BPEL lacks the ability to specify spatiotemporal constraints on tasks as well as factor in environmental parameters into the decision mechanism of the workflow execution engine. These parameters could in theory be "faked" as variables in the BPEL execution engine. However, these values would need to be gathered by an external program which would then need to reach into the BPEL engine and modify its state. According to currently available literature, this is not possible.

E.3.3 YAWL Code for Case 3

The soil sample analysis case cannot be implemented in YAWL for much the same reasons that it cannot be in BPEL. The lack of the ability to specify spatiotemporal constraints on tasks as well as there being no means factor in environmental parameters into the decision mechanism of the workflow execution engine prevents this case from being fully implemented in YAWL.

References

[1] NS2 Network Simulator. http://nsnam.isi.edu/nsnam/index.php/User_Information, June 2008.

[2] The Omnet++ Discrete Event Simulation System. http://www.omnetpp.org/, June 2008.

[3] Active Endpoints. ActiveVOS Engine. http://www.activebpel.org/infocenter/ActiveVOS/v50/index.jsp, June 2008.

[4] AJAXWorkspace. Ajax Workspace. http://www.ajaxworkspace.com/, 2008 July.

[5] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services - Concepts, Architectures, and Applications*. Springer, 2004.

[6] G. Alonso, R. Gunthor, M. Kamath, D. Agrawal, A. El Abbadi, and C. Mohan. Exotica/FDMC: A Workflow Management System for Mobile and Disconnected Clients. *Parallel and Distributed Databases*, 4(3), 1996.

[7] M. Alt, S. Gorlatch, A. Hoheisel, and H.-W. Pohl. A Grid Workflow Language Using High-Level Petri Nets. In *Parallel Processing and Applied Mathematics*, volume 3911 of *LNCS*, pages 715–722, 2006.

[8] A. Avdl. 1.1 Billion Cell Phones Sold Worldwide in 2007. http://www.switched.com/2008/01/25/1-1-billion-cell-phones-sold-worldwide-in-2007-says-study/, January 2008.

[9] T. Bauer and P. Dadam. A Distributed Execution Environment for Large-Scale Workflow Management Systems with Subnets and Server Migration. In *Proceedings of the Second IFCIS International Conference on Cooperative Information Systems*, pages 99–108, 1997.

[10] W. Binder, I. Constantinescu, B. Faltings, K. Haller, and C. Türker. A Multiagent System for the Reliable Execution of Automatically Composed Ad-hoc Processes . *Autonomous Agents and Multi-Agent Systems*, 12(2):219–237, March 2006.

[11] G. Cabri, L. Leonardi, and F. Zambonelli. The Impact of the Coordination Model in the Design of Mobile Agent Applications. In *Proceedings of the 22nd International Computer Software and Application Conference*, pages 436–442, 1998.

[12] G. Cabri, L. Leonardi, and F. Zambonelli. MARS: A Programmable Coordination Architecture for Mobile Agents. *Internet Computing*, 4(4):26–35, 2000.

[13] T. Catarci, F. deRosa, M. de Leoni, M. Mecella, M. Angelaccio, S. Dustdar, B. Gonzalvez, G. Iiritano, A. Krek, G. Vetere, and Z. M. Zalis. WORKPAD: 2-Layered Peer-to-Peer for Emergency Management through Adaptive Processes. In *Proceedings of the International Conference on Collaborative Computing: Networking, Applications, and Worksharing*, pages 1–9, November 2006.

[14] G. Chafle, S. Chandra, V. Mann, and M. G. Nanda. Decentralized Orchestration of Composite Web Services. In *Proceedings of the 13th International World Wide Web Conference*, pages 134–143, 2004.

[15] Ian Chakeres. Additional MANET Page. `http://www.ianchak.com/manet/`, July 2008.

[16] C. Chiang, H. Wu, W. Liu, and M. Gerla. Routing in Clustered Multihop, Mobile Wireless Networks. In *IEEE Singapore International Conference on Networks*, pages 197–211, 1997.

[17] F. M. Costa and G. S. Blair. The Role of Meta-Information Management in Reflective Middleware. In *Proceedings of the ECOOP 2000 Workshop on Reflection and Meta-level Architectures*, June 2000.

[18] T. S. Dahl, M. J. Mataríc, and G. S. Sukhatme. Adaptive Spatio-Temporal Organization in Groups of Robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1044–1049, 2002.

[19] S. Dustdar. Caramba - A Process-Aware Collaboration System Supporting Ad hoc and Collaborative Processes in Virtual Teams. *Distributed and Parallel Databases*, 15(1):45–66, January 2004.

[20] D. Estrin, A. Sayeed, and M. Srivastava. Tutorial - Wireless Sensor Networks. `http://nesl.ee.ucla.edu/tutorials/mobicom02/`, 2002.

[21] B. Naveh et al. Jgrapht. `http://www.jgrapht.org/`, 2008 June.

[22] J. Billington et al. The Petri Net Markup Language: Concepts, Technology, and Tools. In *Proceedings of the International Conference on Applications and Theory of Petri Nets*, number 2679 in LNCS, pages 483–505, 2003.

[23] H.-L. Fang, P. Ross, and D. Corne. A Promising Genetic Algorithm Approach to Job-Shop Scheduling, Rescheduling, and Open-Shop Scheduling Problems. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 375–382, 1993.

[24] C.-L. Fok, G.-C. Roman, and G. Hackmann. A Lightweight Coordination Middleware for Mobile Computing. In *Proceedings of the 6th International Conference on Coordination Models and Languages*, volume 2949 of *Lecture Notes in Computer Science*, pages 135–151. Springer-Verlag, February 2004.

[25] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, pages 653–662, June 2005.

[26] Fujitsu Software Corporation. i-Flow White Paper. `http://www.fujitsu.com/downloads/SG/fapl/workflow/iflow_whitepaper.pdf`, 2002.

[27] A. Garrido, M. A. Salido, F. Barber, and M. A. López. Heuristic methods for solving job-shop scheduling problems. In *Proceedings of the 14th Workshop on New Results in Planning, Scheduling and Design*, 2000.

[28] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[29] B. P. Gerkey and M. J. Matarík. Sold!: Auction Methods for Multirobot Coordination. *IEEE Transactions on Robotics and Automation*, 18(5):758–768, October 2002.

[30] B. P. Gerkey and M. J. Matarík. A Formal Analysis and Taxonomy of Task Allocation in Multi-robot Systems. *International Journal of Robotics Research*, 23(9):939–954, September 2004.

[31] A. Giridhar and P. R. Kumar. Distributed Clock Synchronization Over Wireless Networks: Algorithms and Analysis. In *Proceedings of the 45th IEEE Conference on Decision and Control*, pages 4915–4920, December 2006.

[32] C. Godart, P. Molli, G. Oster, P. Olivier, H. Skaf-Molli, P. Ray, and F. Rabhi. The ToxicFarm Integrated Cooperation Framework for Virtual Teams. *Distributed and Parallel Databases*, 15(1):67–88, January 2004.

[33] D. N. Goodman. Used Phones Drive Third World Wireless Boom. `http://www.msnbc.msn.com/id/15434609/`, October 2006.

[34] Z. J. Haas, M. R. Pearlman, and P. Samar. The Zone Routing Protocol for Ad Hoc Network. `http://www.ietf.org/proceedings/02jul/I-D/draft-ietf-manet-zone-iarp-02.txt`, July 2002.

[35] G. Hackmann, M. Haitjema, C. Gill, and G.-C. Roman. Sliver: A BPEL Workflow Process Execution Engine for Mobile Devices. In *Proceedings of the* 4^{th} *International Conference on Service Oriented Computing*, number 4294 in Lecture Notes in Computer Science, pages 503–508, December 2006.

[36] C. Hahn. A Comprehensive Investigation of Distribution in the Context of Workflow Management. In *Proceedings of the* 8^{th} *International Conference on Parallel and Distributed Systems*, pages 187–192, 2001.

[37] R. Handorean, C. Gill, and G.-C. Roman. Accommodating Transient Connectivity in Ad Hoc and Mobile Settings. In *Proceedings of The* 2^{nd} *International Conference on Pervasive Computing*, number 3001 in LNCS, pages 305–322, 2004.

[38] R. Handorean and G.-C. Roman. Secure Service Provision in Ad Hoc Networks. In *Proceedings of The* 1^{st} *International Conference on Service Oriented Computing*, number 2910 in LNCS, pages 367–383. Springer-Verlag, 2003.

[39] R. Handorean, G.-C. Roman, R. Sen, G. Hackmann, and C. Gill. SPAWN: Service Provision in Ad-hoc Wireless Networks. Technical Report WU-CSE-2005-35, Department of Computer Science, Washington University in St. Louis, August 2005.

[40] R. Handorean, R. Sen, G. Hackmann, and G.-C. Roman. Automated Code Management for Service Oriented Computing in Ad Hoc Networks. Technical Report WU-CSE-2004-17, Department of Computer Science, Washington University in St. Louis, April 2004.

[41] T. Harvey and K. Decker. Planning Ahead to Provide Scheduler Choice. In *Proceedings of the Autonomous Agents Infrastructure Workshop*, May 2001.

[42] J. Haustein and J. Siegel. kSOAP. `http://www.ksoap.org`, 2006.

[43] I. Horrocks. DAML+OIL: A Description Logic for the Semantic Web. *IEEE Bulletin of the Technical Committee on Data Engineering*, 2002.

[44] IETF MANET Working Group. IETF MANET Charter. `http://www.ietf.org/html.charters/manet-charter.html`, July 2008.

[45] IETF MANET Working Group. MANET Status Pages. `http://tools.ietf.org/wg/manet/`, July 2008.

[46] JBoss - A Division of Red Hat. JBoss Enterprise Middleware Documentation. `http://www.jboss.com/docs/index`, June 2008.

[47] D. B. Johnson and D. A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. *Mobile Computing*, 353:153–181, 1996.

[48] B. Karp and H. T. Kung. GPSR: Greedy Perimeter Stateless Routing for Wireless. In *Proceedings of the 6th International Conference on Mobile Computing and Networking*, pages 243–254, 2000.

[49] G. Karumanchi, S. Muralidharan, and R. Prakash. Information Dissemination in Partitionable Mobile Ad Hoc Networks. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pages 4–14, 1999.

[50] J. Kempf and P. St. Pierre. *Service Location Protocol For Enterprise Networks: Implementing and Deploying a Dynamic Service Resource Finder*. John Wiley and Sons, 1999.

[51] L. Li and I. Horrocks. A Software Framework for Matchmaking Based on Semantic Web Technology. In *Proceedings of the 12th International Conference on the World Wide Web*, pages 331–339, 2003.

[52] Q. Li and D. Rus. Communication in Disconnected Ad Hoc Networks Using Message Relay. *Parallel and Distributed Computing*, 63:75–86, January 2003.

[53] N. Liu, M. A. Abdelrahman, and S. Ramaswamy. A Multi-agent Model for Reactive Job Shop Scheduling. In *Proceedings of the 36th Southeastern Symposium on System Theory*, pages 241–245, 2004.

[54] Z. Luo, A. Sheth, K. Kochut, and J. Miller. Exception handling in workflow systems. *Applied Intelligence*, 13(2):125–147, September 2000.

[55] G. Mainland, D. C. Parkes, and M. Welsh. Decentralized, Adaptive Resource Allocation for Sensor Networks. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*, pages 315–328, 2005.

[56] M. Mamei, F. Zambonelli, and L. Leonardi. Tuples On The Air: A Middleware for Context-aware Computing in Dynamic Networks. In *Proceedings of the 2nd International Workshop on Mobile Computing Middleware*, pages 342–347, 2003.

[57] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara. Bringing Semantics to Web Services: The OWL-S Approach. In *Proceedings of the 1st International Workshop on Semantic Web Services and Web Process Composition*, pages 77–80, July 2004.

[58] P. H. Mauguiere, J.-C. Billaut, and J.-L. Bouquard. New Single Machine and Job-Shop Scheduling Problems with Availability Constraints. *Journal of Scheduling*, 8(3):211–231, June 2005.

[59] A. Maurino and S. Modafferi. Partitioning Rules for Orchestrating Mobile Information Systems. *Personal and Ubiquitous Computing*, 9(5):291–300, September 2005.

[60] Merriam Webster Dictionary. Groupware Definition. `http://www.merriam-webster.com/dictionary/groupware`, July 2008.

[61] Microsoft Corporation. Microsoft Office Groove 2007. `http://office.microsoft.com/en-us/groove/default.aspx`, June 2008.

[62] Microsoft Corporation. The BizTalk Server. `http://www.microsoft.com/biztalk/en/us/default.aspx`, June 2008.

[63] A. Montz, D. Mosberger, S. O'Malley, L. Peterson, T. Proebsting, and J. Hartman. Scout: A Communications-Oriented Operating System. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 58–61, May 1995.

[64] R. Muller, U. Greiner, and E. Rahm. AgentWork: A Workflow System Supporting Rule-based Workflow Adaptation. *Data and Knowledge Engineering*, 51(2):223–256, November 2004.

[65] S. Muller-Wilken, F. Wienberg, and W. Lamersdorf. On Integrating Mobile Devices into a Workflow Management Scenario. In *Proceedings of the 11th International Workshop on Database and Expert Systems Applications*, pages 186–190, 2000.

[66] A. L. Murphy, G. P. Picco, and G.-C. Roman. LIME: A Middleware for Physical and Logical Mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 524–533, April 2001.

[67] S. Murthy and J. J. Garcia-Luna-Aceves. An Efficient Routing Protocol for Wireless Networks. *Mobile Networks and Applications*, 1(2):183–197, 1996.

[68] P. Muth, D. Wodtke, J. Weissenfels, A. K. Dittrich, and G. Weikum. From Centralized Workflow Specification to Distributed Workflow Execution. *Journal of Intelligent Information Systems*, 19(2):159–184, 1998.

[69] OASIS Consortium. UDDI Version 3.0.2. `http://uddi.org/pubs/uddi_v3.htm`, October 2004.

[70] OASIS Consortium. BPEL Specification v2.0. `http:\\www.oasis-open.org\committees\download.php\18714\wsbpel-specification-draft-May17.htm`, 2006.

[71] A. Omicini and F. Zambonelli. TuSCoN: A Coordination Model for Mobile Information Agents. In *Proceedings of the 1st International Workshop on Innovative Internet Information Systems*, pages 177–187, 1998.

[72] Open Handset Alliance. Android Project Home Page. http://code.google.com/android/, June 2008.

[73] Open Source. Collaborative Virtual Workspace. http://cvw.sourceforge.net/cvw/info/CVWOverview.php3, July 2008.

[74] Oracle Inc. Oracle 9i Application Server: Oracle Workflow. http://www.oracle.com/technology/products/integration/workflow/workflow_fov.html, June 2008.

[75] OrbiTeam Software. Basic Support for Collaborative Work. http://www.bscw.de/english/product.html, July 2008.

[76] Pallas Athena BV. Flower user manual. http://us.pallas-athena.com/products/bpmflower_product/, 2002.

[77] M. Paolucci, T. Kawmura, T. Payne, and K. Sycara. Semantic Matching of Web Services Capabilities. In *Proceedings of the 1st International Semantic Web Conference*, 2002.

[78] G. Papadopoulos. *Models and Technologies for the Coordination of Internet Agents: A Survey*, volume Coordination of Internet Agents: Models, Technologies, and Applications, chapter 2, pages 25–56. Springer-Verlag, March 2001.

[79] G. A. Papadopoulos and F. Arbab. Coordination Models and Languages. In *The Engineering of Large Systems*, volume 46 of *Advances in Computers*, pages 329–400. Centrum voor Wiskunde en Informatica (CWI), August 1998.

[80] V. Park. and M.S. Corson. Temporally Ordered Routing Algorithm (TORA) IETF Draft. http://tools.ietf.org/id/draft-ietf-manet-tora-spec-04.txt, July 2001.

[81] L. E. Parker. ALLIANCE: An Architecture for Fault Tolerant Multi-Robot Cooperation. *IEEE Transactions on Robotics and Automation*, 14(2):220–240, 1998.

[82] H. Peine and T. Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In *Proceedings of the 1st International Workshop on Mobile Agents*, number 1219 in Lecture Notes in Computer Science, pages 50–61. Springer Verlag, April 1997.

[83] C. Perkins and P. Bhagwat. Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. In *Proceedings of the ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, 1994.

[84] C. Perkins and E. M. Royer. Ad hoc On-Demand Distance Vector Routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, February 1999.

[85] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[86] N. Preguiça, J. Legatheaux Martins, H. J. Domingos, and S. Duarte. Integrating Synchronous and Asynchronous Interactions in Groupware Applications. In *Proceedings of the 11th International Workshop on Groupware*, volume 3706 of *Lecture Notes in Computer Science*, pages 89–104, September 2005.

[87] RDF Core Working Group. Resource Description Framework (RDF) Primer. `http://www.w3.org/TR/rdf-primer/`, February 2004.

[88] G.-C. Roman, R. Handorean, and R. Sen. Tuple Space Coordination Across Space & Time. In *Proceedings of the 8th International Conference on Coordination Models and Languages*, number 4038 in Lecture Notes in Computer Science, pages 266–280, June 2006.

[89] N. Russell, W. M. P. van der Aalst, and A. H. M. ter Hofstede. *Advanced Information Systems Engineering*, chapter Workflow Exception Patterns, pages 288–302. Number 4001 in Lecture Notes in Computer Science. Springer, 2006.

[90] V. Sacramento, M. Endler, H. K. Rubinsztejn, L. dos S. Lima, K. Goncalves, and G. A. Bueno. An Architecture Supporting the Development of Collaborative Applications for Mobile Users. In *Proceedings of the 13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 109–114, 2004.

[91] Salutation Consortium. The Salutation Consortium Homepage. `http://www.salutation.org`, October 2003.

[92] C. Schuler, R. Weber, H. Schuldt, and H.-J. Schek. Scalable Peer-to-Peer Process Management - The OSIRIS Approach. In *Proceedings of the IEEE International Conference on Web Services*, pages 26–34, 2004.

[93] R. Sen, G. Hackmann, M. Haitjema, G.-C. Roman, and C. Gill. Coordinating Workflow Allocation and Execution in Mobile Environments. In *Proceedings of 9th International Conference on Coordination Models and Languages*, number 4467 in LNCS, pages 249–267, June 2007.

[94] R. Sen, R. Handorean, G. Hackmann, and G.-C. Roman. An Architecture Supporting Run-Time Upgrade of Proxy-Based Services in Ad Hoc Networks. In *Proceedings of the Pervasive Computing Conference*, pages 689–696, 2004.

[95] R. Sen, R. Handorean, G.-C. Roman, and C. Gill. *Service-Oriented Software System Engineering: Challenges and Practices*, chapter Service Oriented Computing Imperatives in Ad Hoc Wireless Settings, pages 247–269. Idea Group, 2005.

[96] R. Sen, R. Handorean, G.-C. Roman, and G. Hackmann. Knowledge-Driven Interactions With Services Across Ad Hoc Networks. In *Proceedings of the 2^{nd} International Conference on Service Oriented Computing*, pages 222–231, November 2004.

[97] R. Sen, G.C. Roman, and C. Gill. CiAN: A Workflow Engine for MANETs. In *Proceedings of the 10^{th} International Conference on Coordination Models and Languages*, pages 280–295, June 2008.

[98] R. Simmons. Towards Reliable Autonomous Agents. In *Lessons Learned from Implemented Software Architectures for Physical Agents*, pages 196–203, 1995.

[99] H. Stormer and K. Knorr. PDA- and Agent-based Execution of Workflow Tasks. In *Proceedings of Informatik 2001*, pages 968–973, 2001.

[100] Sun Microsystems. Enterprise JavaBeans Technology. `http://java.sun.com/products/ejb/`, June 2008.

[101] Sun Microsystems. JavaSpaces. `http://www.sun.com/jini/specs/jini1.1html/js-title.html`, June 2008.

[102] K. D. Swenson, S. Pradhan, and M. D. Gilger. WfXML 2.0: XML Based Protocol for Run-Time Integration of Process Engines. `http://www.wfmc.org/standards/docs/WfXML20-200410c.pdf`, October 2004.

[103] S. Thatte. XLANG: Web Services for Business Process Design. `http://www.gotdotnet.com/team/xml\_wsspecs/xlang-c/default.htm`, 2001.

[104] C.-K. Toh. A Novel Distributed Routing Protocol to Support Ad-Hoc Mobile Computing. In *Proceedings of the 15^{th} Annual International Phoenix Conference on Computers and Communications*, pages 480–486, 1996.

[105] A. Vahdat and D. Becker. Epidemic Routing for Partially-Connected Ad Hoc Networks. Technical Report CS-2000-06, Duke University, April 2000.

[106] W. M. P. van der Aalst. Workflow patterns. *Distributed and Parallel Databases*, 14:5–51, 2003.

[107] W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.

[108] W. M. P. van der Aalst, A. H. M. ter Hofstede, and N. Russell. Workflow Patterns Home Page. http://www.workflowpatterns.com, 2007.

[109] D. van Than. Web Services Orchestration. http://www.eurescom.de/message/messagejun2003/Web_Service_Orchestration.asp, June 2003.

[110] J. Waldo. The Jini Architecture for Network-Centric Computing. *Communications of the ACM*, 42(7):76–82, 1999.

[111] Web Ontology Working Group. OWL Web Ontology Language. http://www.w3.org/TR/owl-features/, February 2004.

[112] Web Services Choreography Working Group. WS-CDL v1.0. http://www.w3.org/TR/ws-cdl-10/, November 2005.

[113] Web Services Description Working Group. Web Services Description Language (WSDL) Version 2.0 Primer. http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626/, June 2007.

[114] P. Wyckoff. TSpaces. *IBM System Journal*, 37(3):454–474, 1998.

[115] XML Core Working Group. Extensible Markup Language (XML) 1.0 Fourth Edition. http://www.w3.org/TR/REC-xml/, September 2006.

[116] XML Protocol Working Group. W3c recommendation: Soap version 1.2. http://www.w3.org/TR/soap12-part1/, April 2007.

[117] S. Yang. Job Shop Scheduling With an Adaptive Neural Network and Local Search Hybrid Approach. In *Proceedings of the International Joint Conference on Neural Networks*, pages 2720–2727, July 2006.

[118] R. Zlot, A. Stentz, M. Dias, and S. Thayer. Multi-Robot Exploration Controlled by a Market Economy. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 3016–3023, May 2002.

Vita

Rohan Sen

Date of Birth August 26, 1980

Place of Birth Calcutta, India

Degrees B.S. Magna Cum Laude, Computer Science, May 2003
M.S. Computer Science, December 2005
Ph.D. Computer Science, August 2008

Publications Rohan Sen, Gruia-Catalin Roman, Christopher Gill, and Andrew Frank, CiAN: A Workflow Engine for MANETs, In: *Proceedings of the 9th International Conference on Coordination Models & Languages*, June 2008

Rohan Sen, Gregory Hackmann, Mart Haitjema, Gruia-Catalin Roman, and Christopher Gill, Coordinating Workflow Allocation & Execution in Mobile Environments, In: *Proceedings of the 9th International Conference on Coordination Models & Languages*, pp. 249-267, June 2007

Rohan Sen, Radu Handorean, Gregory Hackmann, and Gruia-Catalin Roman, Knowledge-driven Interactions with Services across Ad Hoc Networks, In: *The International Journal of Collaborative Information Systems*, 16(1), pp. 123-153, March 2007

Radu Handorean, Rohan Sen, Gregory Hackmann, and Gruia-Catalin Roman, Supporting Predictable Service Provision in MANETs via Context-Aware Session Management, In: *Journal of Web Services Research*, 3(3), pp. 1-26, July-September 2006

Gruia-Catalin Roman, Radu Handorean, and Rohan Sen, Tuple Space Coordination Across Space and Time, In: *Proceedings of the 8th International Conference on Coordination Models & Languages*, pp. 266-280, June 2006

Radu Handorean, Rohan Sen, Gregory Hackmann, and Gruia-Catalin Roman, Context-Aware Session Management for Services in Ad Hoc Networks, In: *Proceedings of the 2005 International Conference on Services Computing*, pp. 113-120, July 2005 **(best student paper)**

Rohan Sen, Gregory Hackmann, Gruia-Catalin Roman, and Christopher Gill, Opportunistic Exploitation of Knowledge to Increase Predictability of Agent Interactions in MANETs, In: *Proceedings of Software Eng. for Large-Scale Multi-Agent Systems*, Digital Proceedings, May 2005

Rohan Sen, Radu Handorean, Gruia-Catalin Roman, and Christopher Gill, Service-Oriented Computing Imperatives in Ad Hoc Wireless Settings, (book chapter), In: *Service Oriented Software System Engineering: Challenges and Practices*, pp. 247-269, 2005

Rohan Sen, Radu Handorean, Gruia-Catalin Roman, and Gregory Hackmann, Knowledge-Driven Interactions with Services Across Ad Hoc Networks, In: *Proceedings of the 2nd International Conference on Service Oriented Computing*, pp. 222-231, November 2004

Rohan Sen, Radu Handorean, Gregory Hackmann, and Gruia-Catalin Roman, An Architecture Supporting Run-Time Upgrade of Proxy-Bases Services in Ad Hoc Networks, In: *Proceedings of the 2004 Pervasive Computing Conference*, pp. 689-696, June 2004

August 2008

Collaboration in Mobile Environments, Sen, Ph.D. 2008