CLUSTERING, GROUPING, AND PROCESS OVER NETWORKS

By

YONG WANG

A dissertation submitted in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

DECEMBER 2007

To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation of YONG WANG find it satisfactory and recommend that it be accepted.

Co-Chair

Co-Chair

ACKNOWLEDGEMENT

I would like to begin by thanking Dr. Zhe Dang and Dr. Min Sik Kim, my dissertation advisors. They have been wonderful advisors, providing me with invaluable advice and support. Their enthusiasm for research and their breadth of knowledge always impress me. Their encouragement gives me confidence in conducting the research presented here. I thank them for the time and efforts that have been invested into my research.

I would also like to thank Dr. K. C. Wang for agreeing to be on my committee. I thank him for reading my dissertation and providing me helpful comments. I am fortunate enough to work with him.

Also I would like to thank to the folks in WSU, including Yuanyuan Zhou, Tao Yang, and Linmin Yang, for helping each other and having fun together.

I would like to thank our graduate secretary Mrs. Ruby Young, for all her help at WSU.

Finally, I would like to thank my family. I am forever indebted to my parents for everything that they have given me. Their unconditional support and encouragement give me strength to finish this work. I am very lucky to have such a wonderful family. I dedicate this work to them, and to all the people who love me and whom I love.

CLUSTERING, GROUPING, AND PROCESS OVER NETWORKS

Abstract

by Yong Wang, Ph.D.
Washington State University
December 2007

Chair: Zhe Dang and Min Sik Kim

During the last few years there has been a rapid development in computer networks, especially wireless networks which enable mobile applications. Because of the increasing number of devices involved in network applications, it is necessary to investigate approaches to organize those devices based on their application requirements and make them perform the given tasks. The essential functionalities of a computer network is to establish relationships among network nodes, which are called grouping. An initial form of grouping, called clustering, has been observed by researchers whose research goals are to provide a network architecture that can be used to improve the network performance. In this dissertation, we first propose a number of clustering techniques including SMC, BAC, DCC, and TC, which, in some cases, outperform the existing ones.

A more sophisticated form of grouping, in contrast to clustering, is to build relationships among network nodes based on the nodes' functionalities. To do this, we first propose a specification language, called NetSpec, for studying this more general form of grouping. Using NetSpec, users can describe the desired functionalities of an individual network node, how a subset of nodes, called a group or a bond, are logically connected between each other, and, finally, how such groups evolve. For a network application specified in NetSpec, a compiler is described to translate the specification into a program that will run over a network virtual machine. In the dissertation, we describe the instruction set to support the virtual machine that runs above a physical network. The instruction set is powerful enough to execute the upper layer NetSpec specification while it is simple enough to be efficiently implemented by network protocols running on the under layer physical network. To thread instructions in the instruction set into an execution of the NetSpec specification, we also describe how to implement a non-deterministic scheduler to address fairness, synchronization, group communication control, and concurrency control.

Essentially, NetSpec specifies how groups of network nodes evolve. Looking from the angle of each individual node, such an evolution can be characterize as a thread of atomic transitions, each of which is local; e.g., involving two network nodes. Inspired by this view, in the last part of this dissertation, we propose a form of network processes and study, theoretically, its computability and realization on a physical network.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Dedication

This dissertation is dedicated to my parents

who are always there for me

CHAPTER 1

INTRODUCTION

During the last few years there has been a rapid development in computer networks, especially wireless networks which enable mobile applications. A mobile ad hoc network (MANET) is defined as a collection of mobile platforms or nodes where each node is free to move around arbitrarily [1]. It is a distributed, mobile, wireless, multi-hop network that operates without any existing infrastructure. Sensor networks are composed of hundreds, and potentially thousands of tiny sensor nodes, functioning autonomously, in many cases, and without access to renewable energy resources. Cost constraints and the need for ubiquitous, invisible deployments will result in small sized, resource-constrained sensor nodes. Sensor networks are actually ad hoc networks, in most cases not mobile or with occasional mobility. This can eliminate the nightmare brought by mobility in mobile ad hoc networks. What they share with mobile ad hoc networks is the probabilistic nature of the graph, the problem of connectivity and density control, medium sharing, and scalability.

In addition to the above mentioned difficulties, provided the increasing number of devices involved in network applications, it is necessary to investigate approaches to organize those devices based on their application requirements and make them perform the given tasks. The essential functionality of a computer network is to establish relationships among the network nodes, or grouping. It is natural to investigate how to group entities in the network efficiently according to application requirements. One typical example scenario is the helicopter rescue system. The system automatically assigns a helicopter to pick up a sick people and carry him or her to a hospital. When a helicopter carrying a sick people finds that there is a hospital available, it will send that sick people to that hospital and leave the hospital. When a sick people becomes healthy, he or she will leave the hospital. During this process, several groupings happen in the system. First the sick people is grouped with the helicopter. Then, the sick people leaves the helicopter and grouped with a hospital. At the end, the people becomes healthy and leaves the hospital. This process is depicted in Figure 1.1.

In fact, grouping entities in network is initially observed by researchers studying the clustering problems. For large networks with many devices, it is not necessary and not feasible for each device to maintain

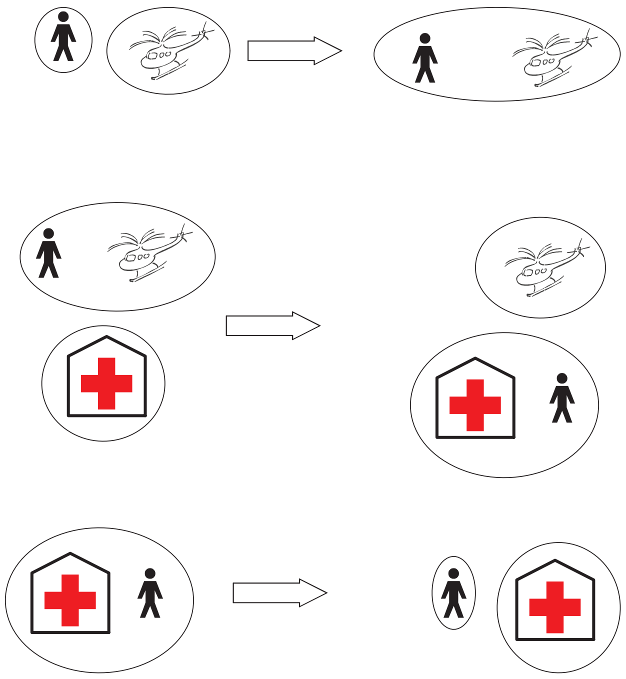


Figure 1.1: Helicopter Rescue System

the whole network's information. To make the operations of a network more efficient and more scalable, it is appealing to select a portion of devices to construct and maintain a control structure. Clustering provides the basis of a group structure. It is defined as partitioning the whole network into groups of devices. In each cluster, there is one clusterhead responsible for controlling the local group of devices, or members. Clusterheads of the network form the backbone of the network. With clustering, local changes are not necessary to be propagated to all other devices in the network. They are only confined to the local group of devices. Efficient routing algorithms [2], data aggregation [3], and network security [4] can be achieved with the aid of clustering. In Chapter 2 we propose some improved clustering approaches for networks, including Size-bounded Multi-hop Clustering (SMC), Bandwidth-adaptive Clustering (BAC), Dual-clusterhead Clustering (DCC), and Typed Clustering (TC) for MANET and WSN.

However, most of the time, the application scope of clustering is limited to providing infrastructure to improve performance of networks, because researchers previously focus on optimizing the performance of communication networks, such as bandwidth, reliability, security and so on. The problem of how to program the applications on the networks is overseen. It is difficult to design, build, and deploy distributed software systems for a network system consisting of large number of devices. Existing approaches to building software system for network applications are not proper to handle this because of the following two challenges.

One challenge is that programmers may deal explicitly with the under layer network utilities using low level programming languages like C++. Thus programmers focus more on under layer details than functionalities of applications.

Device heterogeneity is another challenge brought by large-scale networks. It is not practical to have all devices in the network of the same type. Grimm et al [5] propose that programming distributed applications is increasingly unmanageable because of heterogeneity of devices and system platforms. They also point out that this can lead to duplicated different versions of the same application for different computation devices due to the fact that some existing applications are typically developed for specific devices or system platforms. For example, different applications require different clustering approaches. Though these clustering approaches share a lot of essential similarity, they have to be developed individually. Adopting the idea of JAVA virtual machine [6], a general solution should describe the application functionalities using a specification language, which can be translated into under layer instructions running on a common virtual machine. This can eliminate the redundant effort in developing network applications and let the user focus on the functionalities of applications instead of under layer details.

Our research group proposed Bond Computing System (BCS) [7] targeting at modeling the high-level dynamics of network computing systems. BCS is based on the work of P system, which is an unconventional computing model motivated from natural phenomena of cell evolutions and chemical reactions [8]. It abstracts from the way living cells process chemical compounds in their compartmental structures. Thus, regions defined by a membrane structure contain objects that evolve according to given rules. The objects can be described by symbols or by strings of symbols, in such a way that multisets of objects are placed in regions of the membrane structure. The membranes themselves are organized as a Venn diagram or a tree structure where one membrane may contain other membranes. By using the rules in a nondeterministic, maximally parallel manner, transitions between the system configurations can be obtained. A sequence of transitions shows how the system is evolving. However, P systems assumes maximal parallelism, which is extremely powerful. Considering a network computing model that will eventually be implemented over a network, the cost of implementing the maximal parallelism is unlikely realistic, and is almost impossible in unreliable networks.

The basic building blocks of a BCS are objects, which are logical representations of physical (computing and communicating) entities. In a BCS, objects are grouped in a bond to describe how objects are associated. A configuration is specified by a multiset of bonds, called a collection. The BCS has a number of rules, each of which specifies how a collection evolves to a new one. In BCS, there is no maximal parallelism in a BCS. Rules in a BCS fire asynchronously, while inside a rule, some local parallelism is allowed.

BCS provides a theoretical model for grouping over networks and does not describe how to implement in real network systems. Taking a further step, in Chapter 3, we propose NetSpec, which is a high-level specification language for grouping over network and also a script language of a generalized version of BCS. The challenge is to make NetSpec powerful enough to encode complicated applications, and yet simple enough to efficiently parse into network protocols. The targeted applications for NetSpec are network systems containing computing devices that can move in the system area. The identities of devices are not critical to the applications, while types, or functionality of devices matters. Using NetSpec, users can describe the desired functionalities of an individual network node, how a subset of nodes, called a group or a bond, are logically connected between each other, and, finally, how such groups evolve. For a network application specified in NetSpec, a compiler is described to translate the specification into a program that will run over a network virtual machine, which is described in detail in Chapter 4. The system layout is depicted in Figure 1.2.

From the figure, we can see that the virtual machine separates the translated program from the under layer network services. It supports a set of instructions, which are powerful enough to encode complicated applications, and yet simple enough to efficiently parse into network protocols. As long as the virtual machine supports the instruction set, the program encoded in the instruction set can run on different networks. Compilers, or users can program applications using the provided instruction set. The targeted program is a so-called *instruction sequence program*. To support the instruction set, the network virtual machine deals with synchronization, group communication control, and concurrency control. To execute an instruction, the virtual machine should locate involved entities. To maintain the bond, leader election mechanism is necessary to elect proper entity to maintain the bond information. To ensure the correctness of instruction

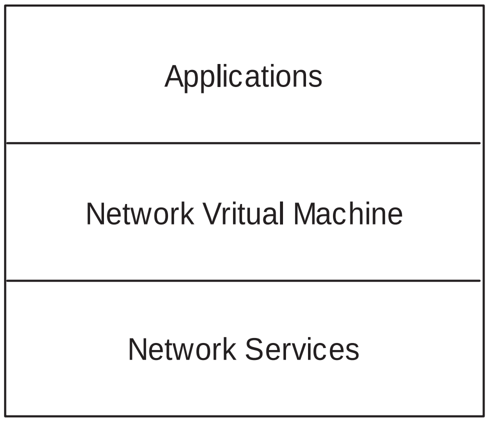


Figure 1.2: System Layout of Proposed Solution

execution, entities should be locked before the operation. Different from programs running on a single processer, the virtual machine provides mechanisms to achieve the sequentialism of instruction executions in network computing systems. The virtual machine also provides concurrency mechanism for interactions among transition executions.

Essentially, NetSpec specifies how groups of network nodes evolve. Looking from the angle of each individual node, such an evolution can be characterize as a thread of atomic transitions, each of which is local; e.g., involving two network nodes. Inspired by this view, in Chapter 5 of this dissertation, we propose a form of network processes, called network process graphs, and study, theoretically, their computability and realization on a physical network.

A network process graph describe a process working on a collection of objects, which are representation of physical entities in a network. Objects are typed but addressless. Objects have their own states (drawn from a finite state set) and change the states as the process goes. The critical part of the graph is the rules, which specify how the process evolves. A rule, for example, describes that, given a source object of type a with state s and a target object of type b with state q, can evolve into, respectively, an object of type a with state s' and an object of type b with state q'. The process executes sequentially; i.e., each time, only one instance of a rule is fired. Additionally, a rule (as the one shown in above) can only be fired when the object of type a with state s is the source specified in the rule and the source currently holds the token (there is only one token throughout the system) and after the firing, the target object of type b with state q'

in the rule receives the token. We demonstrate examples of using such graphs to specify network processes. In particular, we show that the computing power of network process graphs is equivalent to that of VASS (vector addition systems with states). Finally, we describe how to implement the graphs on a physical network.

CHAPTER 2

CLUSTERING FOR NETWORKS

2.1 Overview

During the last few years there has been a rapid development in mobile ad hoc networks (MANET). A MANET is defined as a collection of mobile platforms or nodes where each node is free to move around arbitrarily [1]. We abstract devices in the network as nodes. It is a distributed, mobile, wireless, multi-hop network that operates without any infrastructure.

To make the operations of a MANET more efficient, especially for large networks with many nodes, one approach is to construct and maintain a hierarchical structure. Clustering is such a natural approach to achieve local independent operations and control functions efficiently. Clustering is defined as partitioning the whole MANET into groups of nodes. In each cluster, there is one clusterhead responsible for controlling the local group of nodes, or members. Clusterheads of a MANET form the backbone of the network. With clustering we can achieve efficient routing algorithms [2], data aggregation [9], and network security [4].

Based on the number of hops that a member is from the clusterhead, clustering methods fall in two categories, one-hop [10, 11, 12, 13, 14] and multi-hop [15, 16, 17] approaches. One-hop clustering approaches may form a large number of clusters for large and sparse MANETs. Compared to one-hop clustering approaches, multi-hop approaches can form fewer clusters to allow more control and flexibility [15]. Multi-hop clustering approach can also reduce changes in the cluster structure in case of node movement. In one-hop clustering approaches, if a member moves out of one-hop communication range of the clusterhead, a reclustering process is necessary for the member to find a new cluster to join. In multi-hop clustering approaches, when a member moves out of the one-hop communication range of the clusterhead, if it can still contact the clusterhead with the aid of other forwarding nodes within the same cluster, reclustering process is not necessary, which can reduce the amount of restructuring of clusters.

In this chapter, we first propose a weight-based distributed Size-bounded Multi-hop Clustering (SMC) for MANETs. The main contributions of SMC are as follows. In SMC, clusters are formed and maintained autonomously using only local topology information. In some previous multi-hop clustering approaches like [15, 17], the number of hops that a member can be away from a clusterhead is bounded by a predefined value to approximately control the cluster size. To ensure that clusterheads are not overloaded, SMC bounds each cluster's size by U (a predefined value). To achieve the flexibility inherent in multi-hop clustering and shorten the communication paths between clusterheads and members, instead of bounding the number of hops a member can be away from a clusterhead, SMC uses *RelativeWeight*, which represents the attractiveness of a clusterhead to other nodes and decreases with the number of hops away from the clusterhead, to achieve compactness of clusters. Further, SMC does not require expensive time synchronization mechanisms. The cluster structure evolves with changes in the network topology to adapt to mobility. We compared SMC with other clustering approaches in literature: Lowest-ID [14], WBACA [12], MobDHop [15], and MOBIC [18]. Simulation results show that despite mobility, SMC achieves fewer consistent clusters with short communication paths between clusterheads and members for the given size bound and incurs fewer changes in the face of mobility, which are the desirable properties for clustering especially when scalability is the main concern.

Based on SMC, we propose Bandwidth-adaptive Clustering (BAC), which can adapt to network conditions and reduce the clustering message overhead. BAC forms and maintains multi-hop clusters using only local topology information. To reduce the message overhead, BAC makes members forward the received maintenance messages probabilistically based on network conditions. Lee and Campbell [19] propose that relying only on the buffer occupancy is inaccurate to measure the network condition. Thus, in BAC, members base the forwarding probability on the available bandwidth. The more bandwidth is available, the more probably the maintenance messages will be forwarded. In congested networks, the chance of successful reception of messages is small and the broadcasting of messages can aggravate the congestion. BAC's multi-hop nature can reduce changes when nodes' movement occurs. For multi-hop clustering, when a member moves out of the communication range of the clusterhead, if it can still contact the clusterhead with the aid of other forwarding nodes within the same cluster, reclustering is not necessary. Usually, a multi-hop cluster has more nodes than a one-hop cluster. But if there are too many nodes in a cluster, the clusterhead will be kept busy serving members. To alleviate clusterheads' workload, BAC bounds each cluster's size by U (a predefined value). To further reduce the changes, BAC adopts *Merge* operation to combine neighboring clusters in whole. To form compact clusters, BAC bounds the number of hops between a member and its clusterhead by H (a predefined value).

Then we propose dual-clusterhead clustering (DCC) for wireless sensor network (WSN). Compared to traditional clustering approaches, instead of placing the role of clusterhead on a single node and exhausting the resources on that node, DCC forms clusters with two clusterheads in each cluster using local topology information without requirement of time synchronization mechanism. Liu, Wu, and Pei [20] discuss the two advantages of maintaining multiple active sensor nodes to collect data in a cluster, which is the typical task of a clusterhead. First, it can improve the reliability of data collection. If both of the clusterheads in a cluster are active to collect data, one clusterhead's lost data can be restored in the other active clusterhead. Second, it can help to shorten delays in response to cluster changes. Chiasserini et al. [21] propose that the life-time of cluster-based network is strongly related to clusterheads' failure. In DCC, in the case that one clusterhead fails, the other clusterhead can overtake the failed one. Yang et al. [22] also propose to select two clusterheads in a cluster for robust routing of WSN, while this clustering approach is a centralized one. However, having more than one clusterhead in a cluster brings the problem of coordination and task scheduling. Further, having two clusterheads active in each cluster at the same time will consume more energy. Thus there is a tradeoff between the advantages and disadvantages of having two clusterheads in a cluster. To save energy, in DCC, only one of the two clusterheads in a cluster is active at a time. The active role rotates between the two clusterheads to balance the energy consumption. In large scale WSN, especially for the multi-hop clustering approach, if there are too many nodes in a cluster, the clusterhead will be kept busy serving members in the cluster and become a bottleneck. Ee and Bajcsy [23] propose that a node at the root of a subtree, where a clusterhead is typically located, could be overloaded, which can easily result in scalability issues for large scale WSN. Thus, to alleviate clusterheads' workload, DCC bounds each cluster's size by U (a predefined value). The two clusterheads in each cluster formed by DCC are within the communication range of each other to save energy spent on forwarding messages and increase the reliability of communications.

We investigate two different methods to form clusters with two clusterheads in each cluster for DCC, namely DCC-SH (Single Hop) and DCC-MH (Multiple Hops). In DCC-SH, a member is one hop away from one of the clusterhead and can be at most two hops away from the other clusterhead. In DCC-MH, a member can be multiple hops away from the clusterheads to form fewer denser clusters than DCC-SH at the price that the multi-hop forwarding of messages in DCC-MH consumes more energy than DCC-SH and incurs more messages. Despite the multi-hop nature, DCC-MH forms compact clusters because longer delay in receiving messages from farther clusterheads makes nodes join closer clusters. Simulation results confirm that both DCC-SH and DCC-MH balance the workload of clusterheads, nodes' energy consumption, and cluster size better than those of the popular Lowest-ID and HEED.

In the previous researches, all nodes are considered to be of the same type. None of them takes node type information into account while performing the clustering. However, in reality, nodes represent physical entities. Different physical entities have different types. For example, in a helicopter rescue system, we only want to group a helicopter and several patients together. It is meaningless to group helicopters togethers. Thus, it is useful to take nodes type information into account while clustering nodes. With the type information, we impose more useful and more specific constraints on clustering.

Thus we propose Typed Clustering (TC), which takes nodes' type information into account while clustering them. TC forms and maintains multi-hop clusters using only local topology information. Each node has the type information associated. We describe the constraints based on node types that can be imposed on clustering. Based on our recent work on a specification language for network computing, which is described in details in Chapter 3, we describe how to depict these constraints. To make the description of TC more concrete, we describe how to construct a specific clustering protocol for such an application. TC is based on a multi-hop hop-bounded clustering. TC's multi-hop nature can reduce changes when nodes' movement occurs. For multi-hop clustering, when a member moves out of the communication range of the clusterhead, if it can still contact the clusterhead with the aid of other forwarding nodes within the same cluster, reclustering is not necessary. To form compact clusters, TC bounds the number of hops between a member and its clusterhead by H (a predefined value). To make the description of TC more concrete, we give an example application scenario, in which each type of nodes in a cluster are bounded within a given upper limit.

The rest of this chapter is organized as follows. Section 2.2 introduces the related research work. Sections 2.3, 2.4, 2.5, and 2.6 describe SMC, BAC, DCC, and TC in details. Section 2.7 contains the summary.

2.2 Related Work

With the rapid development of MANET, various clustering methods have been proposed. Based on different criteria, clustering algorithms fall into different categories. Depending on the number of hops an ordinary node is from the clusterhead, clustering methods fall in two categories, one-hop [10, 11, 14, 24, 13] and multi-hop [25, 15, 26, 17] approaches. Different applications have different requirements on the mobility of nodes. Depending on whether nodes in the network are mobile or not, they fall into two further categories, static [27, 3, 26, 13] and mobile [18, 11, 12, 15, 28, 17] approaches.

In the Lowest-ID algorithm [14], a node that has the lowest ID among its neighbors, that have not joined any other cluster, will declare itself the clusterhead. Other nodes will select one of the clusterheads, if it is a direct neighbor, to join and become members. If there is a lowest-ID node with highest mobility, it will cause a lot of changes in clustering. Basagni [10] proposes to use nodes' *weights* instead of IDs or node degrees to select clusterheads. Two one-hop clustering methods that work for static and mobile networks are proposed separately. The approaches select nodes with the highest *weight* in one-hop neighborhood to be clusterheads.

Chatterjee, Das, and Turgut [11] propose another weight-based one-hop clustering, Weighted Clustering Algorithm (WCA), in which, each node has a *combined weight* that is based on the node's degree, transmission power, mobility and battery power. Nodes with the smallest *combined weight* are chosen as clusterheads. Each clusterhead can support δ (a predefined value) nodes in its cluster. Extending this, Dhurandher and Singh [12] propose a one-hop size-bounded clustering algorithm, Weight Based Adaptive Clustering Algorithm (WBACA). Each node calculates its *weight*, based on transmission power, transmission rate, mobility, battery power and degree, to indicate its capability to be a clusterhead. If there is no node with a smaller *weight* in the one-hop neighborhood, the node itself becomes the clusterhead. Otherwise, the node joins the neighboring clusterhead with the smallest *weight*.

Gong, Midkiff and Buehrer [24] also describe a size-bounded one-hop clustering algorithm developed specifically for Ultra Wideband (UWB) networks to minimize the summation of interference generated by all clusters. A node's transmission range in this approach is adaptive to ensure there is another node in contact. Nodes with the minimum interference factor are elected as clusterheads.

Basu, Khan and Little [18] propose another one-hop weight-based clustering approach, MOBIC, that uses the received power levels between two nodes to compute the relative mobility. Extending this, Er and Seah [15] propose a mobility-based d-hop clustering (MobDHop), in which, a clusterhead is the most stable one among all its neighboring nodes. Computing the moving pattern, however, is computationally complex. The distance between two nodes are estimated by measuring the received signal strength, which requires extra hardwares and is only accurate for isotropic networks. Sivavakeesar, Pavlou and Liotta [16] propose a clustering approach that is based on mobility prediction to enable each mobile node to predict the availability of its neighbors and, however, needs location information.

Younis and Fahmy [13] present a one-hop clustering, HEED (Hybrid Energy-Efficient Distributed clustering), which periodically selects clusterheads according to a combination of the node residual energy and a secondary parameter, such as node proximity to neighbors or node degree.

In LEACH (Low-Energy Adaptive Clustering Hierarchy) [3], the main purpose is to distribute energy consumptions throughout clusters in sensor networks. Clusterheads are chosen based on probability. A node chooses the closest clusterhead to join based on received signal strength. Bandyopadhyay and Coyle [27] propose a LEACH-like randomized clustering algorithm for wireless sensor network. A node becomes a clusterhead with probability p and advertises this to nodes within k hops away. A node that receives such advertisements and is not a clusterhead joins the closest cluster. A node that is neither a clusterhead nor has joined any cluster becomes a clusterhead.

Ohta et al. [28] propose an approach that uses cluster division and merging to bound the size of a cluster between a lower and upper bound. When the network is sparse and the degrees of nodes are less than the lower bound, it is not feasible to achieve the lower bound of clusters.

Xu and Gerla [17] propose a *K*-hop clustering algorithm to form the backbone for the routing algorithm. *K* is the predefined maximum number of hops that a member of the cluster can be away from the clusterhead, and allows approximate control of the number and size of clusters.

Amis et al. [25] propose the max-min d-cluster approach, in which a member can be at most d hops away from its clusterhead. All nodes operate asynchronously. This approach runs for $2d$ rounds of information exchange.

Krishnan and Starobinski [26] propose a message-efficient size-bounded clustering method for wireless sensor networks. An initiator distributes the *growth budgets*, which is the number of nodes that can join the cluster, evenly among neighbors and each neighbor continues distributing received *growth budgets* further.

Yang et al. [22] propose to select two clusterheads in a cluster for the robust routing for WSN. They formulate the problem of selecting two clusterheads with the minimum cost in a cluster as a classical transportation problem. The sink node, which maintains the location and energy information of all the nodes in the WSN, runs network flow algorithms to solve the formulated transportation problem. Thus it is a centralized approach instead of a distributed one like DCC.

Manjeshwar and Agrawal [29] classify sensor networks into proactive and reactive networks based on their mode of functioning. They introduce TEEN (Threshold sensitive Energy Efficient sensor Network protocol), which is the first protocol developed for reactive networks. Each node has two thresholds, hard threshold and soft threshold. A node will transmit the data in the current cluster only when the current value of the sensed attribute is greater than the hard threshold and the current value of the sensed attribute differs from the stored value by an amount equal to or greater than the soft threshold. The main drawback is that if the thresholds are not reached, the nodes will never communicate, the user will not get any data from the network at all and will not come to know even if all the nodes die. Thus, this scheme is not well suited for applications where the user needs to get data on a regular basis.

Krishnan and Starobinski propose a message-efficient clustering method for wireless sensor networks [30]. In this approach, the initiator distributes the *growth budgets* evenly among its neighbors, which distribute the received *growth budgets* further. The *growth budget* is the number of nodes that can join the cluster of an initiator node. They assume that only one initiator is active in the same neighborhood at the same time. To ensure this, they provide a probabilistic algorithm to guarantee that initiators will not interfere with each other.

Karmakar and Gupta [31] also propose a size-bounded clustering algorithm to reduce the average node-clusterhead separation for energy-efficient communications. Based on the distance a clusterhead is from a node, clusterheads are grouped into ranks. The rank is defined as the sequence of the closest clusterhead to a node. They present a cost-based greedy heuristic which generates clusters of bounded size with low average rank of the nodes. However, this approach requires the location information of nodes. Durresi and Paruchuri [32] present Adaptive Clustering Protocol (ACP) for WSNs. The network is divided into a uniform hexagonal grid and a node closest to the center of the hexagon is selected as the clusterhead.

However, it is also assumed that each node knows its location.

Li et al. [33] propose an Energy-Efficient Unequal Clustering (EEUC) mechanism that addresses the hot spots problem in multi-hop WSNs. It partitions the nodes into clusters of unequal size. Clusters closer to the base station have smaller sizes than those farther away from the base station to preserve energy for the inter-cluster data forwarding. However, this approach also requires location information. Chiasserini et al. [21] propose a clustering algorithm aiming at maximizing the lifetime of the network by determining optimal cluster size and optimal assignment of nodes to clusterheads. However, they assume that the locations of the clusterheads are known. Moreover, clusterheads are assumed to be chosen a priori and fixed. Venkataraman, Emmanuel, and Thambipillai [34] propose DASCA, a degree and size based clustering for WSN, which restricts the size of each cluster and the number of next hop neighbors a node in a cluster for achieving balance of loads. However, this approach needs the expensive location information of nodes.

Liu, Wu, and Pei [20] propose to group sensor nodes into clusters such that the sensor nodes within each cluster have current strong spatial correlation. However, the dissimilarity measure is application specific, thus it is impossible to use a common dissimilarity measure to accommodate all application scenarios.

The dominating set problem is related to the clustering problem. Rajagopalan provides clustering for ad hoc network based on dominating sets [35]. Depending on how nodes in the backbone are selected, various kinds of dominating sets, such as *k*-dominating set [36, 37], *k*-clustering [38], *k*-connected *k*-dominating set [39], *d*-hop connected and *d*-hop dominating set [40], and connected dominating set [41, 42] are proposed.

2.3 Size-bounded Multi-hop Clustering (SMC)

In SMC, we assume that all nodes have the same communication range. Like the approach in [10], each node is associated with a *Weight* value. Depending on the metric emphasized, the *Weight* can represent the capability, node degree, movement speed, and so on. The bigger a node's *Weight* is, the more capable that node is and more suited to be a clusterhead.

The attractiveness of a clusterhead to another node should decrease as the distance between them increases. In existing multi-hop clustering approaches, like [15, 26], the attractiveness of clusterheads does not decrease as distance increases, which may result in uneven clusters. To achieve the flexibility inherent in multi-hop clustering and shorten the communication paths between clusterheads and members in the result cluster structure, we define $RelativeWeight(y, x)$ to represent the attractiveness of a clusterhead y to another node x and make it decrease inversely proportionally with the number of hops between them. We have tried, in addition to this inversely proportional decrease, quadratic and exponential functions. Both of these make the *RelativeWeight* decrease faster and lead to more clusters, more changes and thus do not adapt to mobility very well.

To deal with mobility, message exchanges are necessary to reflect topology changes in time. Further, because of mobility, each node cannot wait to gather all of the neighboring nodes' information to join the one with the highest *RelativeWeight* or become a clusterhead, since within the waiting interval, the network topology could continue to change.

2.3.1 Proposed Clustering Approach

Initially, all nodes are clusterheads by themselves and periodically broadcast CLUSTER messages containing the address, *Weight* and cluster size. If a clusterhead x is going to join another clusterhead y, we allow for two possible join operations, *Join* and *GroupJoin*.

- If x, which is a clusterhead by itself, finds that $RelativeWeight(y, x)$ is greater than $Weight(x)$ and y's size is less than the upper bound U, it sends a JOIN message to y to become a member of y's cluster.

- If x, whose cluster size is greater than one, finds that $RelativeWeight(y, x)$ is much greater than $Weight(x)$ (exceeding a threshold δ) and the new cluster size does not exceed U, x uses *GroupJoin* to merge with y's cluster by sending a GROUPJOIN message.

The difference between the two kinds of join operations is that if a node uses *Join*, it just requests to become a member of the new cluster by itself. While for *GroupJoin*, which is only used by clusterhead, the clusterhead needs to inform all its members to join the new cluster. Because a large number of small clusters are not desirable, the condition for *Join* operation is looser than that for *GroupJoin* operation.

The clustering approach should be stable and sensitive to topology changes. It should not change the structure drastically in the case of node movement. At the same time, clustering should evolve with changes in the topology of the network. This is a tradeoff between stability and sensitivity to topology changes. The smaller δ is, the more sensitive the clustering structure is to mobility and vice versa.

The primary objective to employ the *GroupJoin* operation is to reduce the number of cluster changes. If a clusterhead joins the new clusterhead without informing all the members in its cluster to join the new clusterhead, a member may have to try several clusterheads before finding a new clusterhead or becoming a clusterhead, which can cause more changes.

If x is a member belonging to clusterhead z's cluster, it can choose to ***join*** clusterhead y if ***RelativeWeight(y,x)*** is much greater than $RelativeWeight(z, x)$ (exceeding a threshold γ) and y's cluster size is less than upper bound U. The smaller γ is, the more sensitive the clustering is to mobility and vice versa.

On receiving the JOIN message from x, if y finds that its cluster size has not reached upper bound U, it will reply with an ACKJOIN message to accept. Then x will become a member of y's cluster and inform z about this by sending a LEAVE message so that z can update its member list. Otherwise, y will send a NOJOIN message to refuse the *Join* request. Upon receiving the GROUPJOIN message from x, if y finds that the new cluster's size will not exceed U, it will send an ACKGROUPJOIN message to x to accept. Otherwise, y will send a NOGROUPJOIN message to refuse the *GroupJoin* request. On receiving the ACKGROUPJOIN message, x will inform all members in its cluster to join y by sending a SWITCH message to them. Because members can choose other clusterhead to join, on receiving the SWITCH message from x, a member will check whether x is still its clusterhead. If this is true, it will join y.

Because of the possibility of packet losses in wireless communications, after sending out the JOIN or GROUPJOIN messages, if a node receives no reply after a waiting period, it will interpret that the message has been lost and will try to join some other cluster or remain in its current cluster.

Each clusterhead broadcasts CLUSTER messages with a predefined frequency. Each member responds with HELLO messages to its clusterhead at the same frequency. Data link disconnection is detected from failure to receive CLUSTER or HELLO messages within a predefined interval.

2.3.2 Dealing with Message Overhead and Consistency

The broadcasting of CLUSTER messages constitutes a large part of the messages. To reduce the number of messages, CLUSTER messages are rebroadcasted only among members in the same cluster. Because of this, there are no chances for members in other clusters to be present in the communication path connecting the clusterhead and members, causing inconsistency in a cluster. Because of the broadcast nature of wireless communications, nodes in the direct vicinity of a cluster can still receive CLUSTER messages and have the chance to join.

During each HELLO interval, each clusterhead broadcasts one CLUSTER message. In the same interval, each member sends one HELLO message and forwards its own clusterhead's CLUSTER message. For a network consisting of n nodes and m clusters, during each HELLO interval, there are n CLUSTER messages and at most $r(n-m)$ HELLO messages, where r is the maximum number of hops that a member is away from the clusterhead. The number of maintenance messages increases as m decreases. Using *RelativeWeight* tends to keep r small, as simulation results confirm, and leads to small message overhead.

2.3.3 Performance Evaluation

We used NS-2 [43] to run our simulations to evaluate SMC. We compare SMC with the widely used Lowest-ID and the weight-based size-bounded one-hop clustering, WBACA, which extends the work of the popular WCA. We also compare SMC with MobDHop and MOBIC that are multi-hop clustering approaches and focus on the stability of clusters. The simulation parameters are listed in Table 2.1 except where pointed out.

Table 2.1: Simulation Parameters for SMC

Parameter	Value
Number of nodes	50
Network size	670 m $\times$ 670 m
Average speed of node movement	20 m/s
Transmission range	200 m
HELLO and CLUSTER messages frequency	Once per second with clock drift
Simulation time	100 seconds

Each node is randomly assigned a *Weight* at the beginning of simulations. For more realism, random waypoint model [44] is used for node mobility. We used AODV [45] as the underlying routing algorithm, which can reduce the number of broadcasts resulting from link breaks and thus is adaptive for high mobility [45]. Each data point reported is the average of 10 runs.

Static Networks

Fig. 2.1 shows the number of clusters formed with different upper bounds of cluster size for SMC, WBACA, which is a size-bounded approach among the four approaches, and the simple lower bound of number of

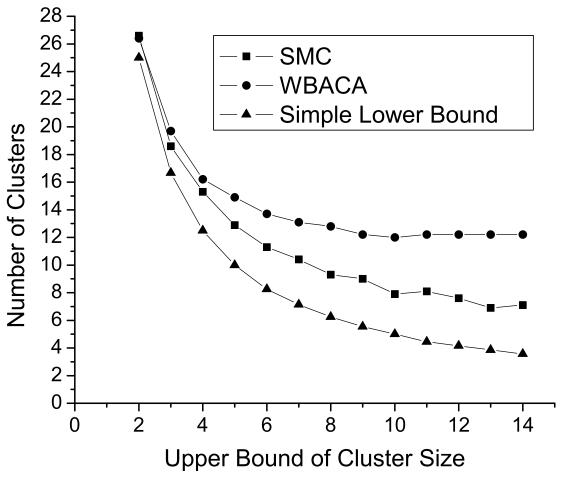


Figure 2.1: Number of Clusters for SMC Performance (Static)

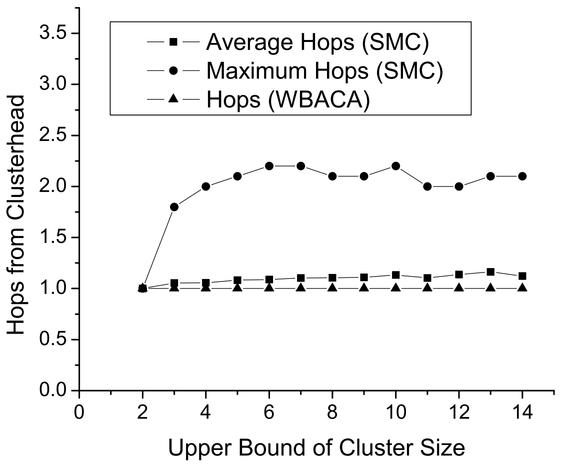


Figure 2.2: Hops from Clusterhead for SMC Performance (Static)

clusters, which is the result of the total number of nodes divided by upper bound of cluster size. For a given upper bound of cluster size, SMC can achieve fewer clusters than WBACA and is closer to the simple lower bound. As the upper bound increases, the number of clusters decreases as expected. Because the *RelativeWeight* of a clusterhead decreases as the number of hops increases, increasing the upper bound beyond a certain limit does not result in a corresponding reduction in the number of clusters. This plateauing effect can also be observed in Fig. 2.1.

Fig. 2.2 shows that the average number of hops between members and clusterheads for SMC is close

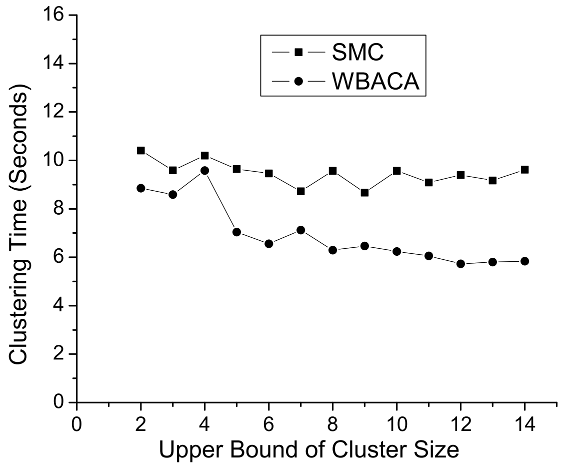


Figure 2.3: Convergence Time for SMC Performance (Static)

to 1 and the maximum number of hops is around 2. This reflects the short communication paths between clusterheads and members, or the compactness of a cluster, due to the fact that using *RelativeWeight* makes nodes join nearby clusterheads.

Fig. 2.3 shows that SMC takes slightly more time to converge than WBACA, as expected, due to multi-hop communication delays and contentions. If there are no cluster changes in 10 seconds, which is 10 message-exchange rounds and long enough for nodes gathering information from nearby nodes, we take it as the convergence of the clustering process. Because members tend to join nearby clusterheads, SMC's convergence time is only slightly more than that of WBACA.

Mobile Networks

Fig. 2.4 shows the number of clusters formed for SMC, MobDHop, Lowest-ID, and MOBIC. The data for MobDHop, Lowest-ID, and MOBIC are from [15] and the simulation scenarios are also adapted to those described in [15]. SMC without upper bound forms fewer clusters than the other three clustering methods that are not size-bounded approaches. Fig.2.5 shows that SMC incurs fewer changes than WBACA for two networks consisting of 50 nodes and 75 nodes separately. Besides the multi-hop nature of clusters, *GroupJoin* further reduces the number of changes.

Fig. 2.6 and Fig. 2.7 show the number of clusters formed and the average number of hops between members and clusterheads, respectively, as time elapses from 6 seconds to 50 seconds. Despite mobility,

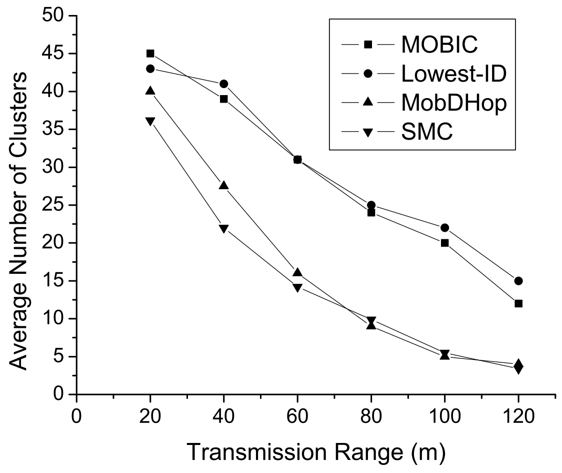


Figure 2.4: Number of Clusters for SMC Performance (Mobile)

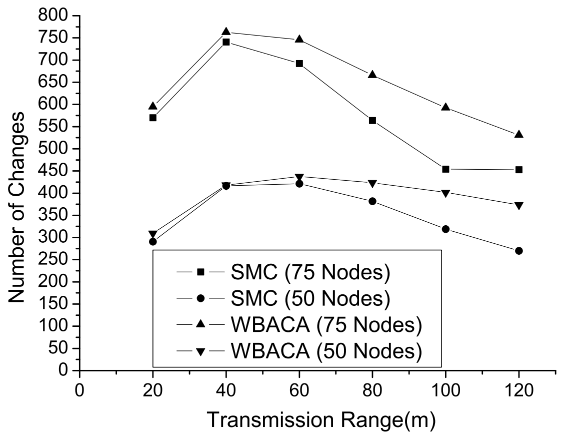


Figure 2.5: Number of Cluster Changes for SMC Performance (Mobile)

SMC forms fewer clusters with short communication paths between clusterheads and members. And the number of clusters remains nearly stable. However, the number of clusters formed by WBACA goes up over time, which shows that SMC is more adaptive to mobility. Fig. 2.8 shows that SMC has about less than two times the messages as that of WBACA for static and mobile networks, even though SMC is a multi-hop clustering.

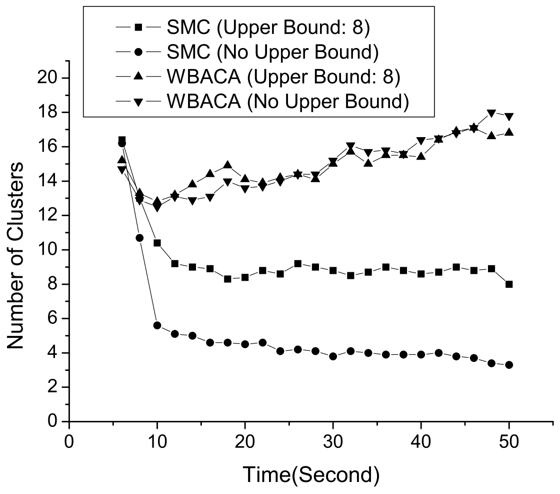


Figure 2.6: Number of Clusters over Time for SMC Performance (Mobile)

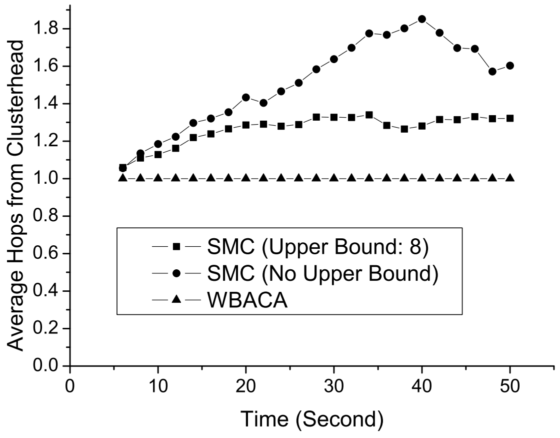


Figure 2.7: Hops from Clusterheads over Time for SMC Performance (Mobile)

2.4 Bandwidth-adaptive Clustering

In this section, we describe a new clustering approach for MANETs, bandwidth-adaptive clustering (BAC). We first describes the basis and a special case of BAC. Then we describes BAC's adaptivity to bandwidth.

2.4.1 Basic Clustering

In this section, we describe BAC's size-bounded and hop-bounded clustering with deterministic forwarding of maintenance messages, which is the basis and a special case of BAC. We assume that all nodes have the

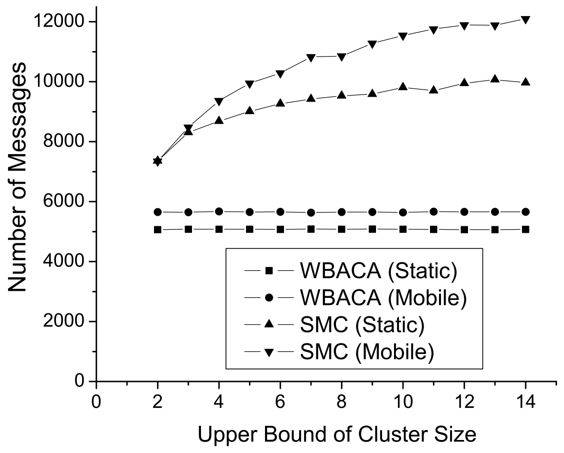


Figure 2.8: Number of Messages

same communication range. Each node has a unique ID and is associated with a *Weight* value. The *Weight* can represent the node's capability, resources, and so on. The greater a node's *Weight* is, the more capable that node is and more suited to be a clusterhead.

Constructions of Clusters

Initially, all nodes are clusterheads and periodically broadcast CLUSTER messages containing the ID, *Weight*, the number of hops that the message has traveled, and the cluster size information.

For a node to join another cluster, BAC allows two possible operations, *Join* and *Merge*. The difference between them is that if a node uses *Join*, it just requests to become a member of the new cluster by itself. *Merge* is only used by a clusterhead with some members in its cluster. The clusterhead needs to inform all its members to join the new cluster. Employing *Merge* operation can reduce members' attempts to find new clusters to join, and the changes.

If x, which is a clusterhead by itself, finds that $Weight(z)$ is greater than $Weight(x)$ and z's cluster size is less than the upper bound U, it sends a JOIN message to z to request to *Join* z's cluster. If clusterhead y, whose cluster size is greater than one, finds that $Weight(z)$ is much greater than $Weight(y)$ (exceeding a threshold δ) and the new cluster size does not exceed U, y uses *Merge* to combine with z's cluster by sending a MERGE message containing its cluster information. The smaller δ is, the more likely clusters are to combine with each other, leading to fewer clusters and more changes. There is a tradeoff between stability and sensitivity to topology changes.

If a member m of clusterhead v's cluster finds that $Weight(z)$ is much greater than $Weight(v)$ (exceeding a threshold γ) and z's cluster size is less than upper bound U, it will request to *Join* clusterhead z's cluster by sending a JOIN message to z.

On receiving the JOIN message from x or m, if z finds that its cluster size has not reached the upper bound U, it will reply with an ACKJOIN message to accept. Otherwise, z will send a NOJOIN message to refuse. If the *Join* request is accepted, m will inform v so that v can update its cluster information.

On receiving the MERGE message from y, if z finds that the new cluster's size will not exceed upper bound U, it will send an ACKMERGE message to y to accept. Otherwise, z will send a NOMERGE message to refuse. On receiving the ACKMERGE message, y will inform members in its cluster to join z's cluster. Members more than H hops away from z will join some other cluster or become a clusterhead.

After sending out the JOIN or MERGE message, if a node cannot receive any reply after a waiting period, it will assume that the message has been lost and try to join some other cluster or remain in its cluster based on its *Weight*.

Maintenance of Clusters

Nodes exchange messages to reflect changes in topology. Each clusterhead broadcasts CLUSTER messages periodically. Each member responds with HELLO messages and forwards its clusterhead's CLUSTER messages that are within H hops away from the clusterheads. Thus, members in other clusters cannot be in the communication paths between the clusterheads and members, causing inconsistency. Because of the broadcast nature of wireless communications, nodes in the direct vicinity of a cluster can still receive CLUSTER messages and have the chance to join.

If a member does not receive a CLUSTER message from its clusterhead after $n_{allowed}$ CLUSTER message intervals, it assumes that its clusterhead cannot be contacted and will either compete as a new clusterhead or join some other cluster based on its *Weight*. If a clusterhead does not receive a HELLO message from a member after $n_{allowed}$ HELLO message intervals, it assumes that the member cannot be contacted and will remove it from the member list.

Because of congestions, a CLUSTER message may have traveled more hops before reaching a member even though the clusterhead does not move away. This will cause a member's incorrect update of the number of hops away from the clusterhead. To distinguish the actual movement of nodes from congestions, if a member detects that a CLUSTER message has traveled more hops than the previously received one, it will check with the neighbor from which it receives the previous CLUSTER message. If that neighbor also becomes more hops away from the clusterhead, the member will update the number of hops away information accordingly. Otherwise, it will not update.

Deterministic Forwarding Message Overhead

For a network consisting of n nodes and m clusters, during each CLUSTER message interval, because a CLUSTER message will be forwarded by members in that cluster, there are n CLUSTER messages. A HELLO message will be forwarded by each node on the path between the member and the clusterhead. So in response to the CLUSTER messages, there are at most $H(n-m)$ HELLO messages.

2.4.2 Bandwidth-adaptive Clustering

Random Forwarding of Messages

The benefits of clustering come at the cost of additionally generated messages to control the clusters. To reduce the message overhead, BAC makes members forward the received CLUSTER messages probabilistically. The forwarding probability, PF, is calculated based on the available bandwidth. The more bandwidth a member has, the more probably that member will forward the received CLUSTER messages. Renesse et al. [46] propose an approach to estimating the usage of bandwidth by calculating the size of sent, received, and sensed packets over a fix period of time. Wan, Eisenman, and Campbell [47] propose CODA (COngestion Detection and Avoidance), which uses a sampling scheme that activates local channel monitoring at the appropriate time to save energy. Because energy is usually not a major concern for MANET, we adopt the approach of [46].

If S is the size of all packets sent, received, and sensed by a node over a period of T, the average used bandwidth over T is:

$$BW_{used} = \frac{S}{T} \tag{2.1}$$

The available bandwidth percentage, P_{avl}, is:

$$P_{avl} = \frac{BW_{max} - BW_{used}}{BW_{max}} \tag{2.2}$$

In (2.2), BW_{max} is the maximum available bandwidth. The forwarding probability, PF, is defined as:

$$PF = \begin{cases} 0 & P_{avl} \leq L_{min} \\ \frac{P_{avl} - L_{min}}{L_{max} - L_{min}} & L_{min} < P_{avl} \leq L_{max} \\ 1 & P_{avl} > L_{max} \end{cases} \tag{2.3}$$

L_{min} and L_{max} are the lower and upper thresholds.

Because of the probabilistic forwarding of maintenance message, for a member at h hops away from the clusterhead, the probability that it cannot received a CLUSTER message after $n_{allowed}$ consecutive CLUSTER message intervals is:

$$P_{loss}(h) = \prod_{j=1}^{n_{allowed}} \left(1 - \prod_{i=1}^{h-1} PF_{ij} \right) \tag{2.4}$$

In (2.4), PF_{ij} is the forwarding probability of node i at the jth time during the consecutive $n_{allowed}$ CLUSTER message intervals. Increasing PF at each node can reduce P_{loss}, reflecting the tradeoff between the number of forwarded CLUSTER messages and the probability that members lose contact with the clusterheads. P_{loss} increases as h increases, which enhances the compactness of clusters.

Reduced Message Overhead

We assume that nodes are uniformly distributed on a 2-dimensional space with density d. All nodes have the same fixed communication range r and can receive every packet within r. We assume that x is the only clusterhead and there is no nearby interfering cluster. In Fig. 2.9, A_{in} denotes the intersection area of the two nodes' communication range. We assume that the forwarding probability, PF, is the same for all nodes. We use $P_{recv}(s)$ to denote the probability that member y can receive CLUSTER messages from clusterhead x at the distance s. We give the analysis results for $P_{recv}(s)$ when y is at most two hops away from x.

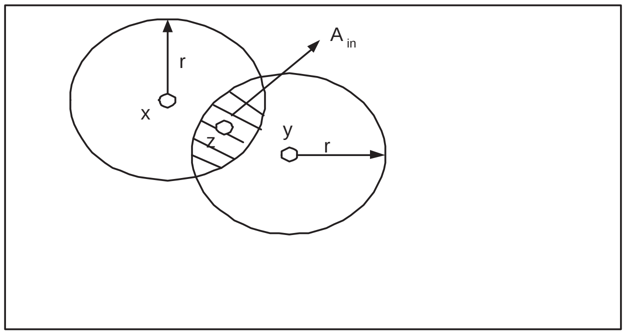


Figure 2.9: Network Scenario for Analysis of P_{recv}

If $s \leq r$, y can receive the CLUSTER messages from x directly. We have:

$$P_{recv}(s) = 1 \tag{2.5}$$

If $r < s \leq 2r$, using the result in [48], the size of A_{in}, $A_{in}(s)$, can be calculated as:

$$A_{in}(s) = 2r^2 \arccos\left(\frac{s}{2r}\right) - \frac{1}{2}s\sqrt{4r^2 - s^2} \tag{2.6}$$

The probability that at least one relay node is located in A_{in} is:

$$P(n_{in} \geq 1 \mid s) = 1 - e^{-dA_{in}(s)} \tag{2.7}$$

Thus, when $r < s \leq 2r$, we have:

$$P_{recv}(s) = PF \times \left(1 - e^{-dA_{in}(s)}\right) \tag{2.8}$$

Using the parameters in Table 2.2, Fig. 2.10 shows $P_{recv}(s)$ with different s and d values given $PF = 0.8$.

2.4.3 Performance Evaluation

We used NS-2 [43] to compare BAC and BAC-D, which is a special case of BAC with deterministic forwarding of maintenance messages, with the weight-based size-bounded one-hop clustering, WBACA, which

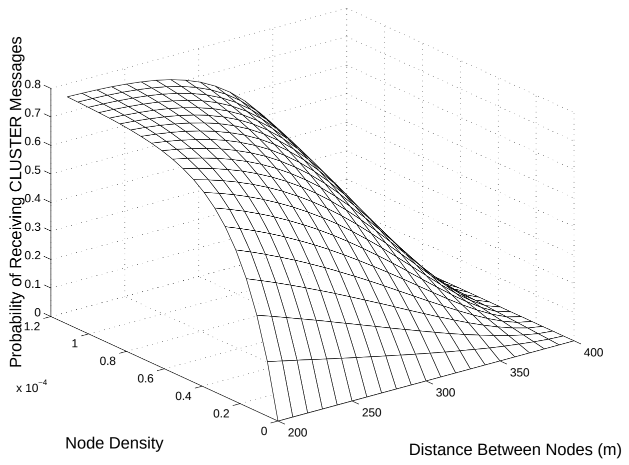


Figure 2.10: Probability of Receiving CLUSTER Messages

Table 2.2: Simulation Parameters for BAC

Parameter	Value
Number of nodes	50
Network size	670 m $\times$ 670 m
Average speed of nodes	20 m/s
Communication range	200 m
Maximum bandwidth	1M bps
(L_{min}, L_{max})	(0.1, 0.9)
$n_{allowed}$	3
CLUSTER message frequency	Once per second with clock drift
Simulation time	100 seconds

extends the popular WCA, and the multi-hop hop-bounded clustering, *K*-hop clustering. The simulation parameters are listed in Table 2.2 unless mentioned otherwise. Each node is randomly assigned a *Weight* at the beginning of the simulations. To increase ties, *Weights* are discretized into 16 levels. We used the random waypoint model [44] with node speed varying between 16m/s and 24m/s, and no pause time. For more realism, each node sends a 200-byte data packet randomly at least every 0.5 second as background data traffics. We used AODV [45] as the underlying routing algorithm, which makes use of advantages from both distance-vector and on-demand. AODV can also reduce the number of broadcasts resulting from broken links and thus is adaptive for high mobility [45]. Each data point reported is the average of 10 runs.

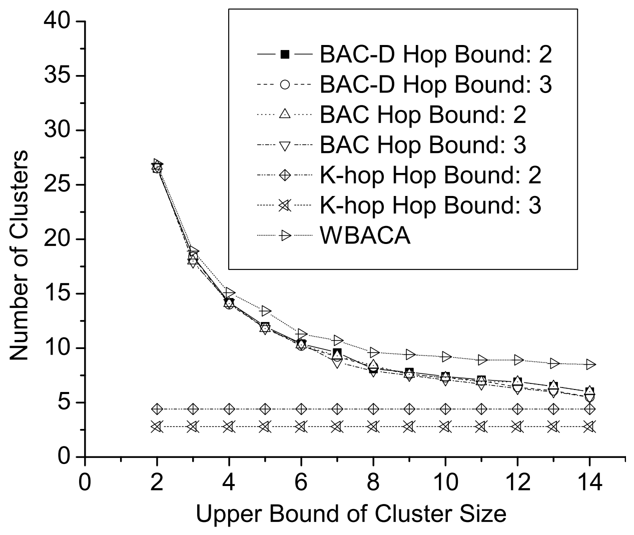


Figure 2.11: Effect of Upper Bound of Cluster Size on the Number of Clusters for BAC Performance (Static)

Static Networks

In Fig. 2.11, we show that for a given cluster size bound, the number of formed clusters for BAC and BAC-D is fewer than WBACA and close to the non-size-bounded *K*-hop clustering. For BAC and BAC-D, because the number of neighbors within a given number of hops is limited, increasing the size bound beyond a certain limit does not result in a corresponding decrease in the number of clusters.

In Fig. 2.12, we show that the average number of hops between members and clusterheads for BAC and BAC-D is close to 1 and fewer than that of *K*-hop clustering. This reflects the compactness of clusters due to the reason that BAC and BAC-D bound the cluster size and the number of hops that a member can be away from a clusterhead.

If there is no cluster change in 10 seconds, which is 10 message-exchange rounds and long enough for nodes exchanging information, we take it as the convergence of the clustering. In Fig. 2.13, we show that BAC and BAC-D take slightly more time to converge than WBACA.

Mobile Networks

In Fig. 2.14, we show that as the communication range increases, the number of formed clusters decreases accordingly. The one-hop clustering, WBACA, forms more clusters than the multi-hop clustering approaches. BAC forms nearly the same number of clusters as BAC-D. In Fig. 2.15, we show that the multi-hop clustering approaches, BAC, BAC-D, and *K*-hop incur fewer changes than one-hop WBACA in the case of

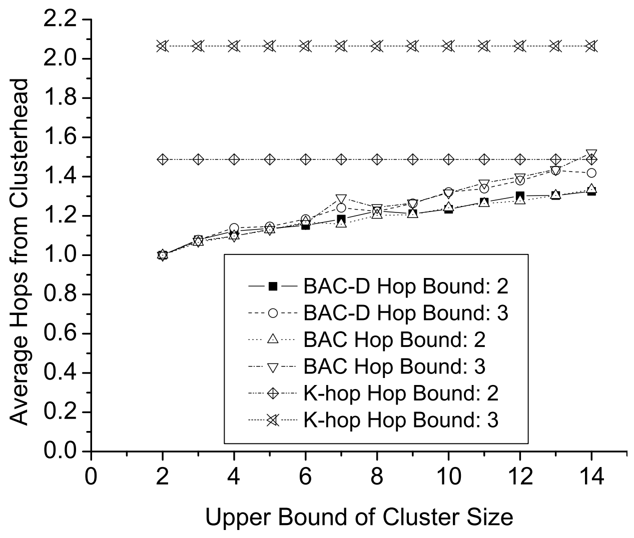


Figure 2.12: Effect of Upper Bound of Cluster Size on Average Hops from Clusterhead for BAC Performance (Static)

nodes' movement. Because of the probabilistic forwarding of maintenance messages, BAC incurs slightly more changes than BAC-D.

To show the effectiveness of PF in reducing message overhead, we experimented with the same PF for all nodes and made it vary from 0.1 to 1 with the cluster size bound being 8 and the hop bound being 3. In Fig. 2.16, we show that as PF decreases, more CLUSTER messages are reduced, while the average cluster size remains nearly the same. This is due to the reason that a member can still receive CLUSTER messages from several forwarding nodes although a single node's PF is reduced.

In Fig. 2.17, we show that the number of clusters formed by BAC and BAC-D is between those of K-hop clustering and WBACA, and remains nearly stable over time. In Fig. 2.18, we show that BAC and BAC-D have short communication paths between clusterheads and members over time.

In Fig. 2.19, we show that BAC and BAC-D incur slightly more messages than WBACA and fewer than K-hop clustering, even though BAC and BAC-D are multi-hop clustering approaches. The reason is that the HELLO messages from members are triggered by CLUSTER messages for BAC and BAC-D. However, WBACA makes nodes send HELLO messages periodically and the more changes of it lead to more messages. Because of the employment of probabilistic forwarding of CLUSTER messages, BAC incurs fewer messages than BAC-D. As the nodes' communication range increases, the network gets denser

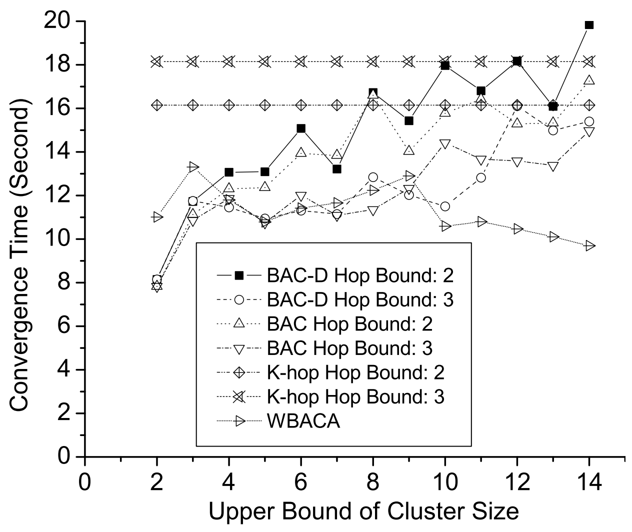


Figure 2.13: Effect of Upper Bound of Cluster Size on Convergence Time for BAC Performance (Static)

and the available bandwidth decreases, which makes BAC drop more messages.

2.5 Dual-clusterhead Clustering

In DCC, we assume all nodes have the same communication range. Each node has a unique ID and is associated with a *Weight* value. Depending on the metrics emphasized, the *Weight* can represent the node's capability, resources, and so on. The bigger a node's *Weight* is, the more capable that node is and more suited to be a clusterhead. Basagni [10] proposes that weight-based approach can choose those nodes that are better suited for the role of clusterheads. We propose two different methods for DCC, namely DCC-SH and DCC-MH. For both of the two methods, the two clusterheads in the same cluster are within the communication range of each other to increase the reliability of the communications between them and save energy spent on intermediate forwarding nodes. In the quasi-one-hop DCC-SH, each member has at least one clusterhead within the communication range. A member can be at most two hops away from the other clusterhead. In the multi-hop DCC-MH, a member can be multiple hops away from the clusterhead. Because the data packets from closer clusterheads arrive earlier than those from farther away clusterheads, nodes prefer to join closer clusterheads.

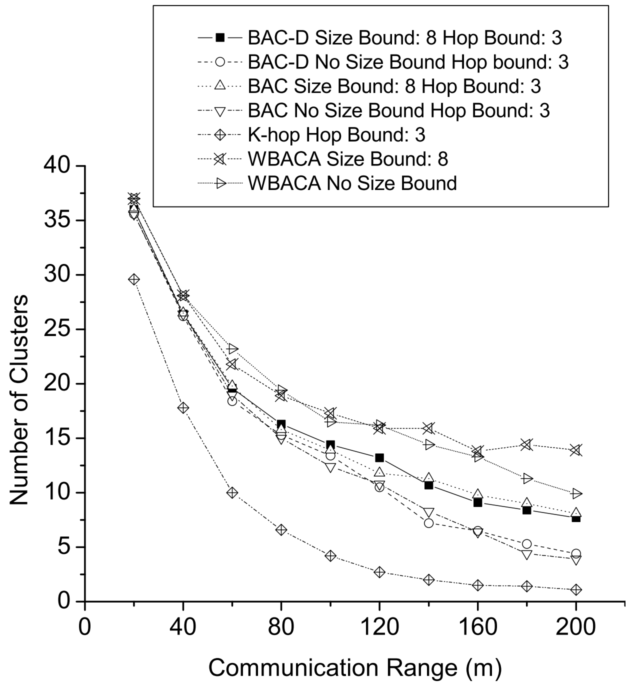


Figure 2.14: Effect of Communication Range on Clusters for BAC Performance (Mobile)

2.5.1 DCC-SH

DCC-SH has two major phases: pairing phase and clustering phase. In the pairing phase, a node with the highest *Weight* in the one-hop neighborhood will try to pair with another node to become clusterheads together. Then, in the clustering phase, the paired clusterheads broadcast CLUSTERHEAD messages to let non-clustered nodes join. Because we do not assume time synchronization, the two phases can happen at the same time in different clusters.

Pairing Phase

Initially, all nodes' are non-clustered. Each node broadcasts HELLO messages containing its ID and *Weight* information. On receiving the HELLO messages from nodes within its communication range, each node creates a list of neighbors. Because of the unreliable nature of wireless communications, each node broadcasts the HELLO message twice to increase the chance of correct receiving of the HELLO messages.

After gathering neighbors' HELLO messages, if a non-clustered node x finds that it has the highest *Weight* in the one-hop neighborhood, it will try to pair with another non-clustered node, y, by sending a PAIR message. In the case that there are more than one highest *Weight* nodes in the neighborhood, nodes'

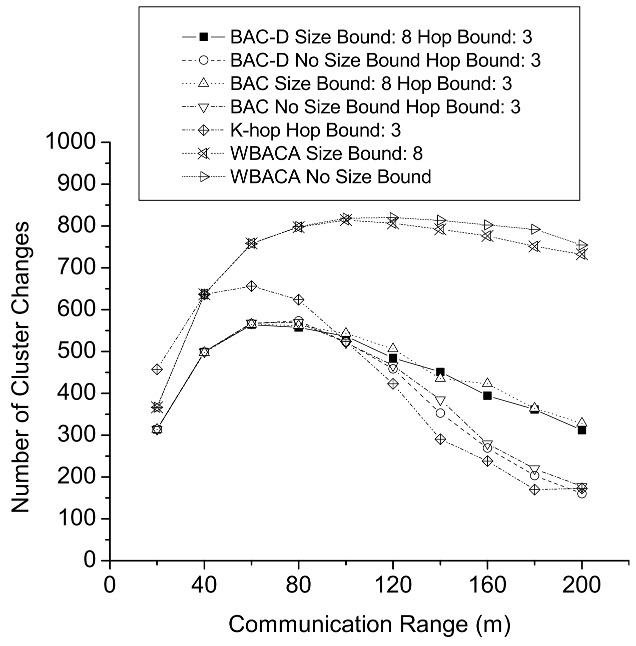


Figure 2.15: Effect of Communication Range on Number of Cluster Changes for BAC Performance (Mobile)

IDs are used to break the ties. To reduce the chance of conflicts, y is randomly chosen. Upon receiving the PAIR message, if y has not been paired with any other node nor has joined some cluster, it will reply with an ACKPAIR message to accept this pair request. Otherwise, y will reply with a NOPAIR message to refuse the pair request. Upon receiving the NOPAIR message, x will try to pair with another neighbor by sending PAIR message to it unless there is no non-clustered neighbor.

Clustering Phase

If x successfully pairs with y, x and y will both become clusterheads. To converge fast and bound the size of each cluster, adopting a similar method as that in [30], we divide the upper bound of cluster size, U, evenly between the two clusterheads. If U is an even number, each clusterhead will accept at most $U/2 - 1$ nodes' join request. If U is an odd number, the clusterhead with smaller ID will accept at most $(U + 1)/2 - 1$ nodes, the other one will accept at most $(U - 1)/2 - 1$ nodes. One assumption in [30] is that each time, only one clusterhead is active to accept nodes' join requests, which can ensure that there is no conflict in the process of attracting member nodes. We do not have this assumption so that multiple CLUSTERHEAD messages can be received by a single node, which chooses the first one to join.

The paired clusterheads x and y broadcast CLUSTERHEAD messages to let neighboring non-clustered

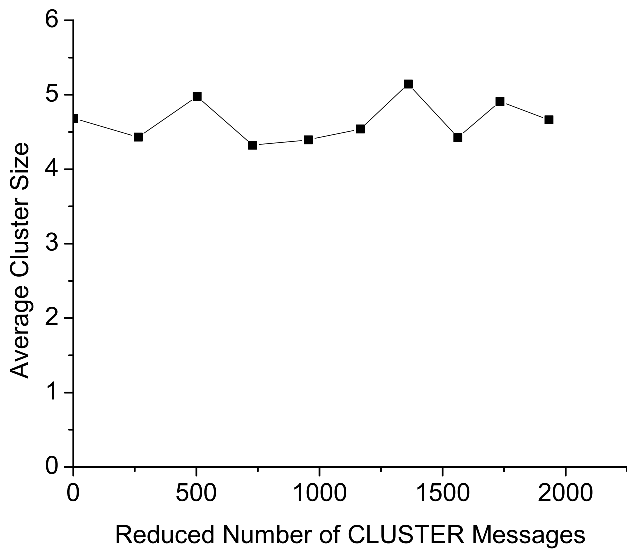


Figure 2.16: Reduced Number of CLUSTER Messages vs Average Cluster Size for BAC Performance (Mobile)

nodes to join. Upon receiving a CLUSTERHEAD message, a non-clustered node m will send a JOIN message to join the cluster without choosing the clusterhead with highest *Weight*. If x or y receives this JOIN message, it will check whether the upper bound of cluster size is reached. If the upper bound is not reached, x or y will send an ACKJOIN message to accept the join request. Otherwise, it will refuse the join request by sending a NOJOIN message.

On receiving the ACKJOIN message, m will broadcast a MEMBER message to let neighbors know that it has become a member node of a cluster so that a non-clustered node can remove m from its one-hop non-clustered neighbors list. If a non-clustered node finds that it becomes the highest *Weight* node in the one-hop neighborhood, it will start the pairing phase.

Upon receiving the NOJOIN message or after a waiting period without any response from the intended clusterhead, the non-clustered node m will try some other clusterhead. If m cannot join any existing cluster and finds that it has the highest *Weight* among all the non-clustered one-hop neighbors, it will try to pair with a non-clustered neighbor to become a clusterhead. Thus, the chance for a non-clustered node to be a member node is higher than to be a clusterhead, which leads to fewer and denser clusters. If a node completes the execution of DCC-SH without selecting to join a cluster or becoming a clusterhead, it becomes a clusterhead by itself.

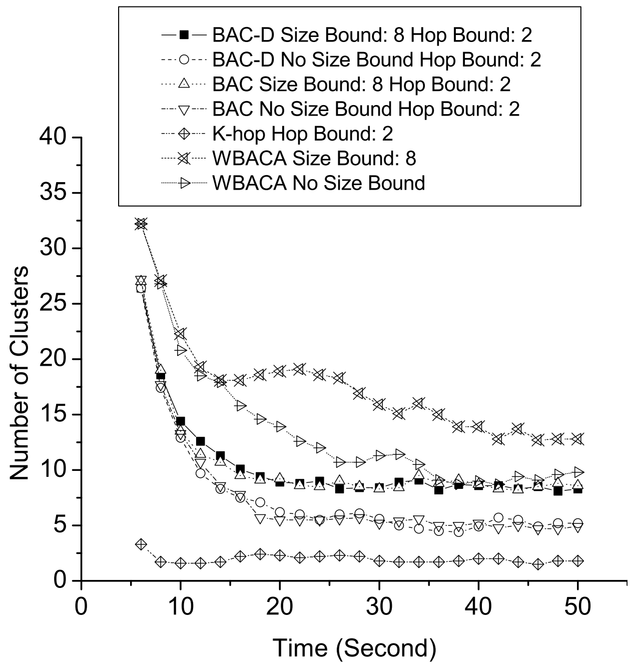


Figure 2.17: Number of Clusters over Time for BAC Performance (Mobile)

An Example of DCC-SH

In Figure 2.20, the numbers in the parentheses are the *Weights* of the nodes. The numbers beside the parentheses are the IDs of nodes. An edge between two nodes represents that the two nodes are in the communication range of each other. We assume U (the upper bound of cluster size) is 8. Node 1 finds that it has the highest *Weight* in the neighborhood. It will try to pair with another node to become clusterheads. If node 1 gets paired with node 2, these two nodes will send CLUSTERHEAD messages to let neighboring nodes join. The upper bound of cluster size is divided evenly between nodes 1 and 2. Each will accept another three nodes' join requests. They accept the join requests of nodes 3, 4, 5, 6, 7, 8. The join requests of nodes 10 and 12 will be refused. Upon receiving the MEMBER message from node 6, node 10 finds that it has the highest *Weight* among the non-clustered one-hop neighbors. It gets paired with node 9 to become clusterheads. After sending out the CLUSTERHEAD messages, they form another cluster with nodes 11, 12, 13.

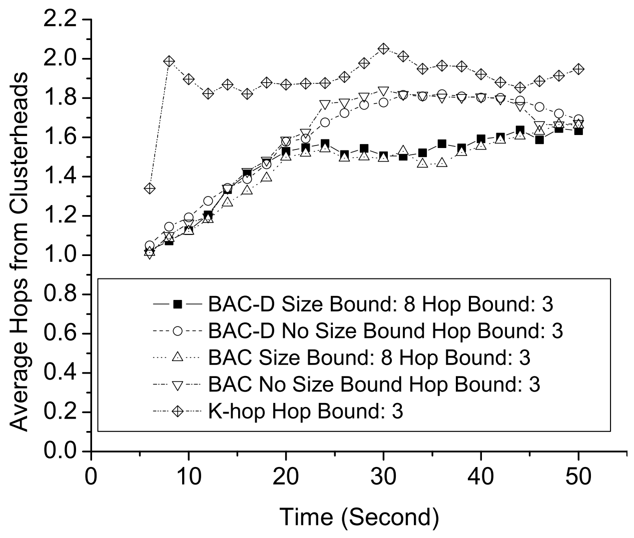


Figure 2.18: Short Communication Paths between Clusterheads and Members over Time for BAC Performance (Mobile)

2.5.2 DCC-MH

DCC-MH has two major phases: clustering phase and core selection phase. In the clustering phase, multi-hop clusters are constructed based on nodes' *Weights*. In the core selection phase, two nodes in the core of each cluster are selected to act as clusterheads. Nodes in the core of a cluster are not always those that have the highest *Weights*. But choosing core nodes as clusterheads can reduce energy consumption on other forwarding nodes. This is a tradeoff between selecting nodes with high *Weight* and reducing energy consumption on forwarding nodes. Similar to DCC-SH, because we do not assume time synchronization, the two phases can happen at the same time in different clusters.

Clustering Phase

Initially, all nodes are non-clustered. To reduce the initial contentions among nodes and save energy, a portion of nodes are selected to be clusterheads. The decision is made by each node choosing a random number between 0 and 1. If the random number is less than a threshold, $CH_{prob} = \frac{\lambda}{Upperbound} \leq 1$, then the node will become a clusterhead. Because a clusterhead can become a member node by combining with another cluster, λ should be big enough to ensure coverage of clustered nodes. There is a tradeoff between λ and the energy consumed for initial clusterheads. The bigger λ is, the more energy will be consumed for initial clusterheads.

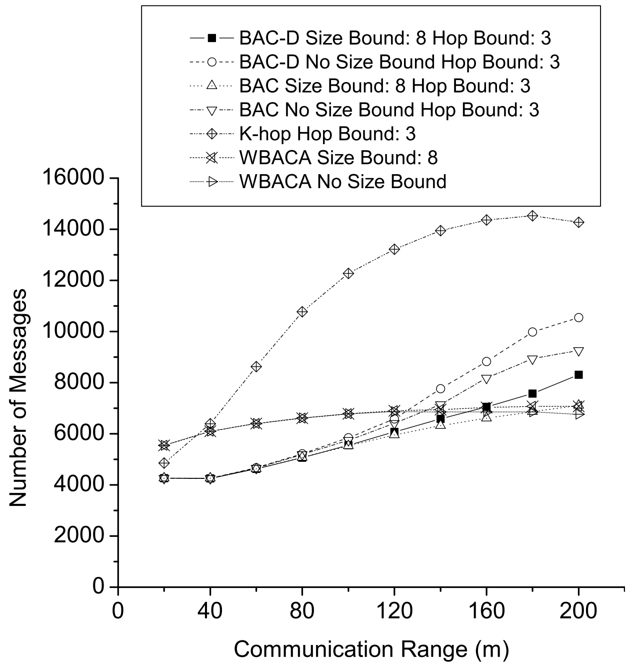


Figure 2.19: Communication Range vs Number of Messages for BAC Performance (Mobile)

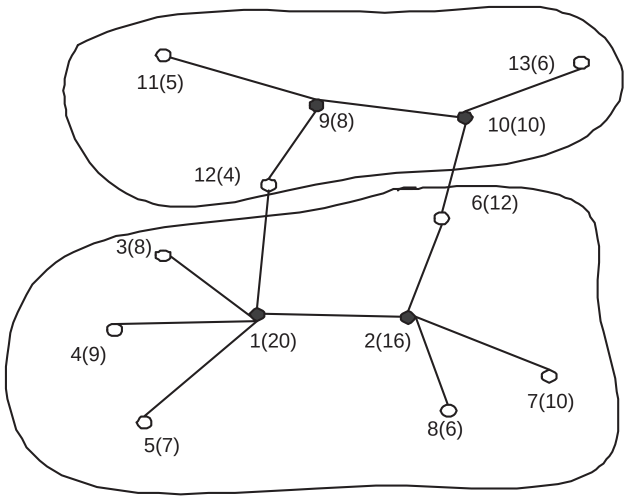


Figure 2.20: Example Scenario of DCC-SH

Each selected clusterhead broadcasts CLUSTERHEAD messages containing its ID, *Weight*, and cluster size information. If a node is going to join another cluster, we allow for two possible operations, *Join* and *GroupJoin*. The difference between them is that *Join* is used by a non-clustered node to request to become a member of a cluster, while *GroupJoin* can only be used by a clusterhead to combine the two clusters in whole. The purpose to adopt *GroupJoin* operation is to produce fewer bigger clusters and reduce the number of changes when combining clusters.

A non-clustered node, x, requests to *Join* a clusterhead, y, by sending a JOIN message to y. The JOIN message contains x's ID and the neighbor's ID, which x receives the CLUSTERHEAD message from and is the parent of x in the tree structure, to let y have the complete information about the tree structure of the formed cluster. On receiving the JOIN message from x, if y finds that its cluster size has not reached the upper bound U, it will reply with an ACKJOIN message to accept and add x to the member list. x will forward CLUSTERHEAD message of y further. Otherwise, y will send a NOJOIN message to refuse the *Join* request. x will try some other clusterhead.

If clusterhead z receives CLUSTERHEAD messages from y and finds that y's *Weight* is bigger than its *Weight*, z will try to combine its cluster in whole with y's cluster by sending a GROUPJOIN message to y. Upon receiving the GROUPJOIN message from z, if y finds that the new cluster's size will not exceed the upper bound U, it will send an ACKGROUPJOIN message to z to accept. Otherwise, y will send a NOGROUPJOIN message to refuse. On receiving the ACKGROUPJOIN message, z will inform all members in its cluster to join y's cluster by sending a SWITCH message to them and forward the CLUSTERHEAD messages of y further. To reduce the energy spent on changing clustering structures, members are not allowed to join some other cluster except for the *GroupJoin* operation. Because of possible packet losses in wireless communications, after sending out the JOIN or GROUPJOIN messages, if a node receives no reply after a waiting period, it will assume that the message has been lost and try to join some other cluster or remain in its current cluster.

Core Selection Phase

When a clusterhead finds no more changes in the cluster, it will begin the core selection process. The nodes in a cluster form a tree structure. By reducing the length of the communication pathes between members and clusterheads, we can reduce the energy used for forwarding messages. Thus it is desirable to place clusterheads at the middle of a tree. The center of a tree is the subgraph induced by the nodes of minimum eccentricity [49]. It consists of either a single node or a pair of adjacent nodes [50]. Because the clusterhead has the complete information about the tree, it uses the algorithm in [50] to compute the center of the tree. First it removes all the nodes of degree one, together with their incident edges. Then it repeats the process until a single node or two adjacent nodes are left, which is the center of the tree.

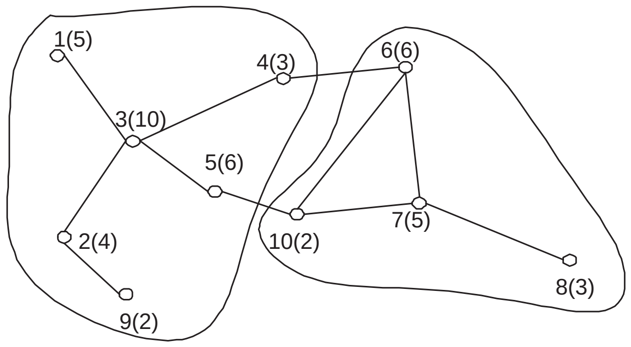


Figure 2.21: Example Scenario of DCC-MH

Because two nodes need to be selected to be the core, if the tree has just one node in the center, a neighbor of the center node will be selected as the other core node. After identifying the two nodes of in the core of the tree, the original clusterhead will inform the newly selected core nodes to become the clusterheads. If the original clusterhead is not in the core, it will give up the role as the clusterhead. The newly selected clusterheads in the core will inform other nodes in the cluster about the new clusterheads information. If a node completes the execution of DCC-MH without joining a cluster or becoming a clusterhead, it becomes a clusterhead by itself.

An Example of DCC-MH

In Figure 2.21, we assume U (the upper bound of size) is 6 and initially all the nodes become clusterheads. Node 3 has the highest *Weight* among its neighbors. Assume nodes 1, 2, 4, 5 receive the CLUSTERHEAD messages from node 3 first. They find that the *Weight* of node 3 is bigger than the *Weights* of themselves and each will send a GROUPJOIN message to node 3. Assume the sequence of the GROUPJOIN messages arrivals at node 3 is 1, 2, 4, 5. Node 3 will accept the *GroupJoin* request of nodes 1, 2, 4, 5 by replying with ACKGROUPJOIN messages. Then nodes 1, 2, 4, 5 will forward the CLUSTERHEAD messages of node 3 further. Nodes 6, 9, 10 will find that the *Weight* of node 3 is bigger than their own *Weights*. They will also request to *GroupJoin* node 3's cluster. If the GROUPJOIN message of node 9 arrives first, nodes 3 will accept this. When node 3 receives the GROUPJOIN message from nodes 6 and 10, it will find that the cluster size has reached the upper bound and send NOGROUPJOIN messages to nodes 6 and 10 to refuse the $GroupJoin$ requests. Because node 6 has the biggest *Weight* among its neighbors, node 6 will remain as a clusterhead. Upon receiving the NOJOIN message, node 10 will try to find some other cluster to join.

At this time, node 6 sends CLUSTERHEAD messages. Then nodes 4, 7, and 10 can also hear the message. Because node 4 has already joined node 3's cluster, node 4 will not join node 6's cluster. Node 10 will *GroupJoin* node 6's cluster. Node 7 is the clusterhead of the cluster formed with node 8. Now node 7 finds that it can combine its cluster with node 6's cluster. It will send a GROUPJOIN message to node 6. Upon receiving such a message, node 6 finds that the cluster size will not exceed the upper bound, so it replies with a ACKGROUPJOIN message to accept. Then, node 7 sends the SWITCH message to let node 8 join node 6's cluster.

When clusterheads 3 and 6 find no more changes in the clusters, they will begin the core selection phase. Clusterhead 3 will identify itself and node 2 as the core. It will inform node 2 to become the new clusterhead and other nodes in its cluster about the new clusterheads information. Clusterhead 6 will identify itself and node 7 as the core. It will inform node 7 to become the new clusterhead and also inform other nodes in its cluster about the new clusterheads information.

2.5.3 Balancing Workload of Clusterheads

After the construction of dual-clusterhead clusters, member nodes send data packets to the clusterheads to process and send to the base station. In DCC, to save energy, each time, only one of the two clusterhead in each cluster is active to collect data packets from members and aggregate them into one data packet to send it to the base station. To balance the workload between the two clusterheads of a cluster, the active role of clusterhead rotates between the two clusterheads at a predefined frequency. If an inactive clusterhead receives data packets from a member node, it will forward them to the active clusterhead and inform the member node about the current active clusterhead so that the member node can send data packets to the active clusterhead directly.

2.5.4 Performance Evaluation

We used NS-2 [43] to implement DCC. We compare DCC-SH and DCC-MH with the widely used Lowest-ID [14] and HEED [13], which extends the work of the popular LEACH [9]. Most of the simulation parameters are similar to those in [13] and are listed in Table 2.3 unless mentioned otherwise. We used a similar radio model as that in [9]. In order to transmit k bits data over distance d, the power spent on the radio is $k(E_{elec} + E_{amp} \times d^n)$ J, where $n = 2$ for $d < d_0$, and $n = 4$ for $d \geq d_0$. E_{amp} varies according to the distance d between the sender and receiver. $E_{amp} = \epsilon_{fs}$ when $d < d_0$, while $E_{amp} = \epsilon_{mp}$ when $d \geq d_0$. To receive a data packet of length k bits, the energy spent is kE_{elec} J. Each node is randomly assigned a *Weight* at the beginning of the simulations. To increase ties, *Weights* are discretized into 20 levels. We use AODV [45] as the underlying routing algorithm for multi-hop communications between nodes for DCC-MH. Each data point reported is the average of 10 runs.

Table 2.3: Simulation Parameters for DCC

Parameter	Value
Number of nodes	100
Network size	100 m $\times$ 100 m
Initial energy	2 $J/battery$
Simulation time	200 seconds
Communication range	30 m
Base station location	(50,175)
E_{elec}	50 nJ/bit
ϵ_{fs}	10 $pJ/bit/m^2$
ϵ_{mp}	0.0013 $pJ/bit/m^4$
Threshold distance (d_0)	75 m

Characteristics of Clusters

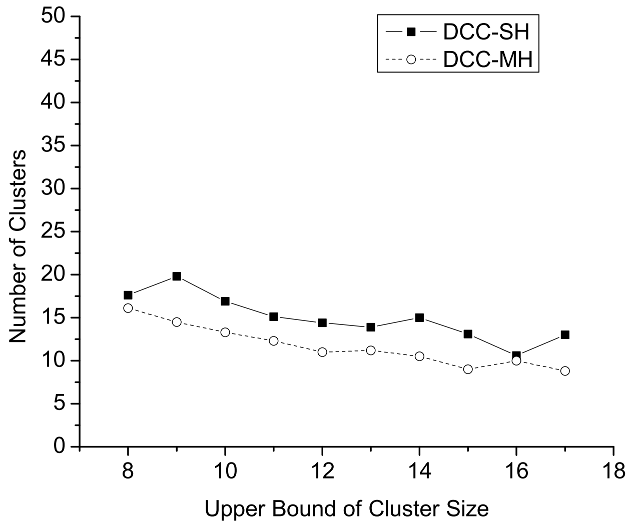


Figure 2.22: Number of Clusters vs Upper Bound for DCC Performance

In Figure 2.22, we show that for a given upper bound of cluster size, the multi-hop DCC-MH forms

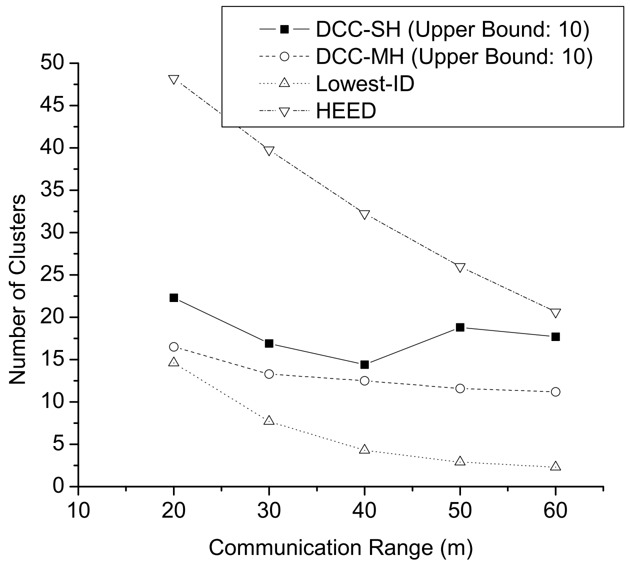


Figure 2.23: Number of Clusters vs Transmission Range for DCC Performance

fewer clusters than DCC-SH. As the size bound increases, the number of clusters decreases as expected. In Figure 2.23, we show that DCC-MH can form fewer clusters than HEED and DCC-SH, and slightly more clusters than the unbounded Lowest-ID approach. The number of formed clusters remains almost stable for DCC with the increase of communication range, while the performance of HEED is affected by the communication range more.

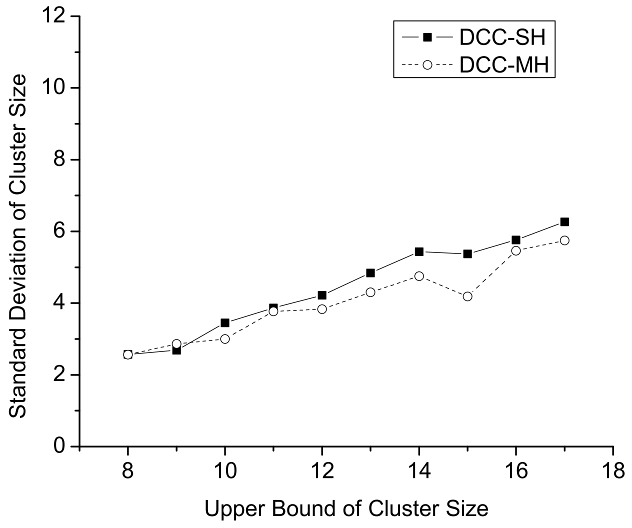


Figure 2.24: Cluster Size Standard Deviation vs Upper Bound for DCC Performance

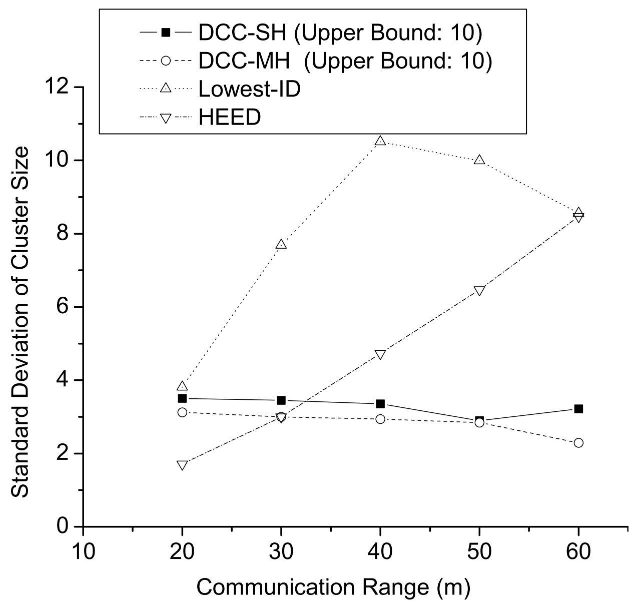


Figure 2.25: Cluster Size Standard Deviation vs Transmission Range for DCC Performance

In Figure 2.24 and Figure 2.25, we show the standard deviation of cluster size with different size bound and communication range separately. DCC controls and balances cluster size better than Lowest-ID and HEED, because the size-bounded nature of DCC confines the variance of clusters' size.

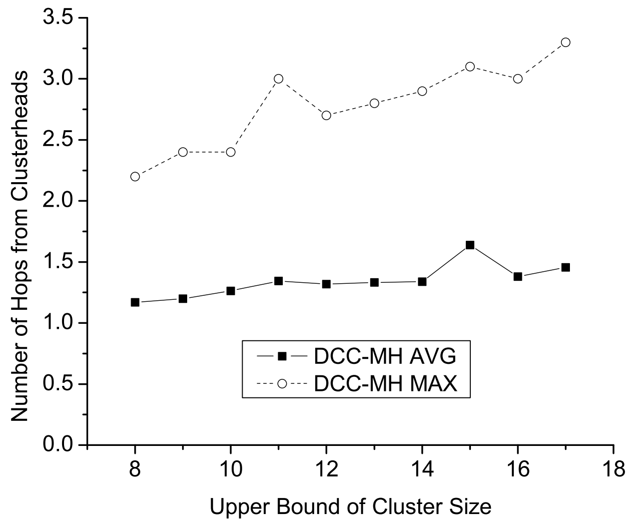


Figure 2.26: Hops from Clusterheads vs Upper Bound for DCC Performance

In Figure 2.26 and Figure 2.27, we show that the average number of hops between members and their

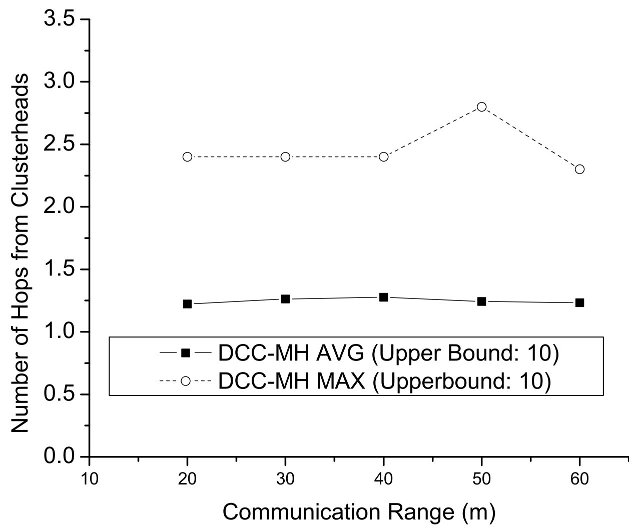


Figure 2.27: Hops from Clusterheads vs Transmission Range for DCC Performance

closer clusterheads for DCC-MH is close to 1 and the maximum number of hops is around 3. This reflects the short communication paths between members and clusterheads, or the compactness of clusters. Besides DCC-MH's size-bounded nature, the longer delay for a node to communicate with a farther away node makes a node prefer joining nearby clusters.

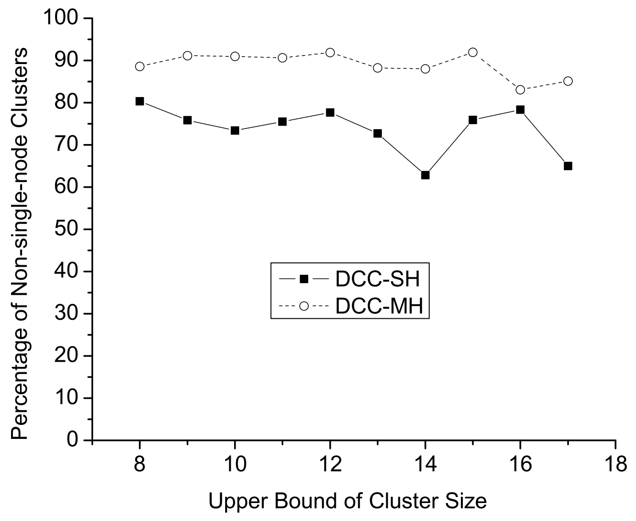


Figure 2.28: Percentage of Non-single Clusters vs Upper Bound for DCC Performance

In Figure 2.28 and Figure 2.29, we show the percentage of non-single-node clusters. Single-node clusters arise when a node is forced to form the cluster by itself because of not joining some cluster or not pairing

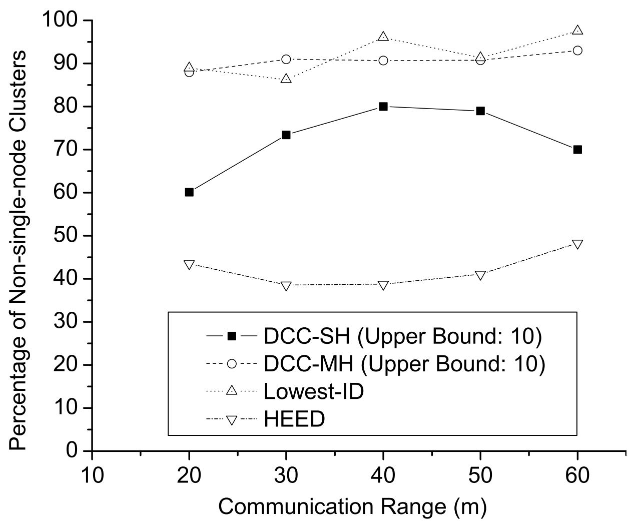


Figure 2.29: Percentage of Non-single Clusters vs Transmission Range for DCC Performance

with some node. DCC-MH has a higher percentage of non-single-node clusters than DCC-SH and close to Lowest-ID, which is not size-bounded. The reason is that a node in DCC-MH can join a clusterhead more than one hop away, while for DCC-SH a non-clustered node cannot get clustered if it cannot pair with any other non-clustered node or join any cluster in the one-hop neighborhood. Both DCC-SH and DCC-MH produce higher percentage of non-single-node clusters than HEED.

Message Overhead

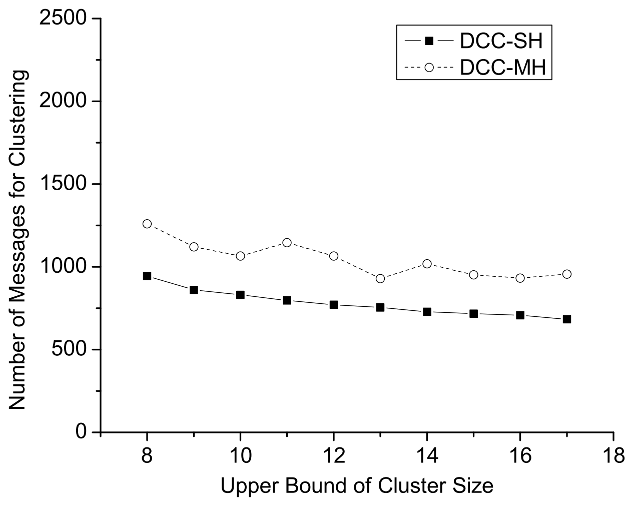


Figure 2.30: Number of Messages vs Upper Bound for DCC Performance

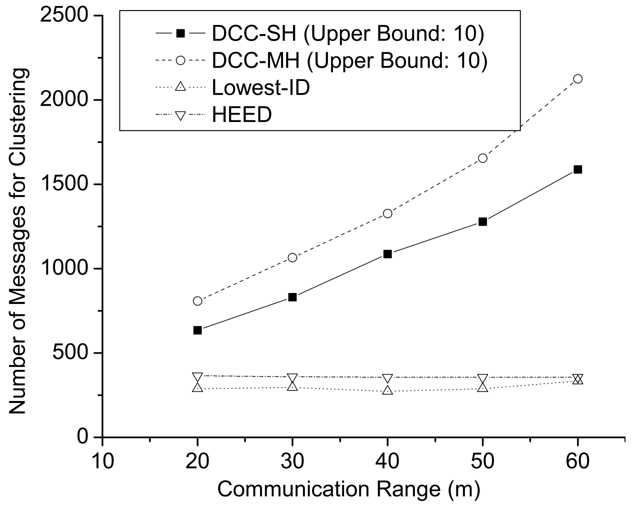


Figure 2.31: Number of Messages vs Transmission Range for DCC Performance

In Figure 2.30 and Figure 2.31, we show the number of messages for clustering with different size bound and communication range separately. The multi-hop DCC-MH incurs more messages than DCC-SH to form fewer and bigger clusters. The number of messages decreases as the upper bound increases because of the decrease of the amount of contentions. Because of the paring and core selection operations, DCC-SH and DCC-MH incur more messages than Lowest-ID and HEED.

Workload of Clusterheads

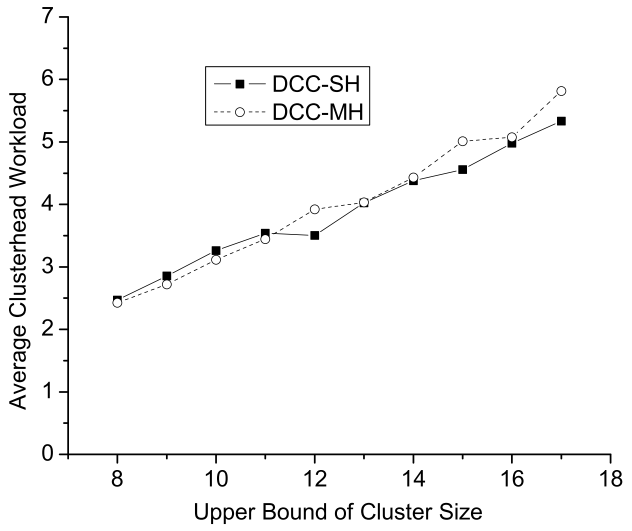


Figure 2.32: Average Clusterhead Workload vs Upper Bound for DCC Performance

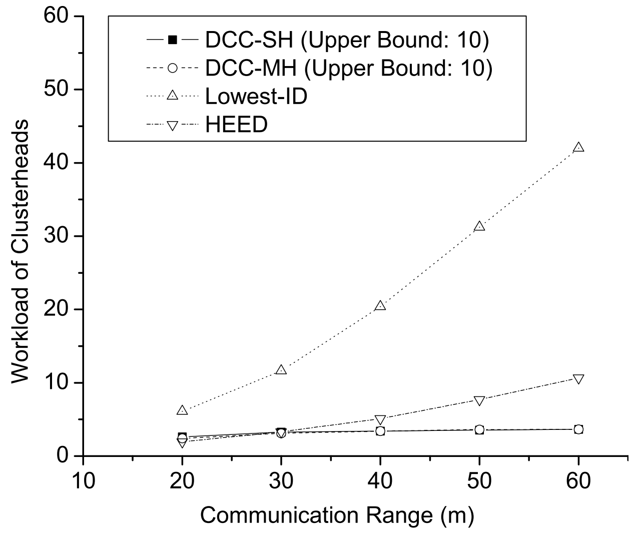


Figure 2.33: Average Clusterhead Workload vs Transmission Range for DCC Performance

In Figure 2.32 and Figure 2.33, we show the average workload of clusterheads with different size bound and communication range separately. After the clustering, each member sends data packets to the clusterheads at a predefined frequency. Clusterheads will aggregate the received data packets and send them to the base station. The workload is defined as the number of data packets a clusterhead receives and processes during each interval. DCC has much lower workload of clusterheads than Lowest-ID and HEED. Clusterheads' workload of Lowest-ID increases quicker than the other three approaches when increasing the communication range, due to the fact that a clusterhead will attract more member nodes for the unbounded Lowest-ID approach when the communication range is increased.

In Figure 2.34 and Figure 2.35, we show the standard deviation of clusterheads' workload with different size bound and communication range separately. The standard deviation of clusterheads' workload for DCC-SH and DCC-MH increases with the upper bound of cluster size. When increasing the communication range of nodes, the standard deviation of clusterheads' workload of Lowest-ID and HEED increases much more than that of DCC, while that of DCC remains nearly stable.

Energy Consumption

In Figure 2.36 and Figure 2.37, we show the average and maximum energy consumption of nodes during the whole simulation interval separately with different communication range. DCC consumes more energy than

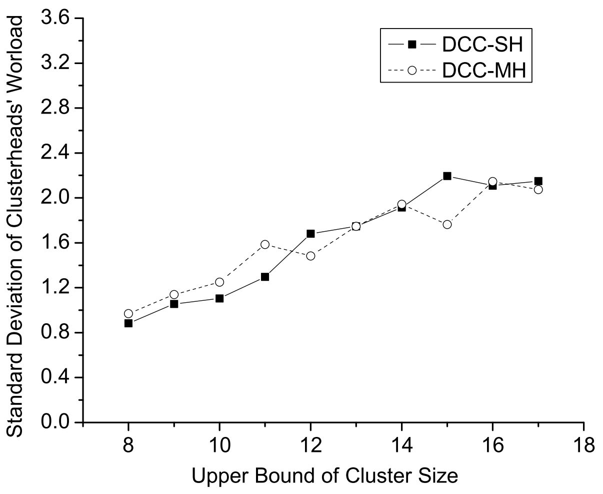


Figure 2.34: Standard Deviation of Clusterhead Workload vs Upper Bound for DCC Performance

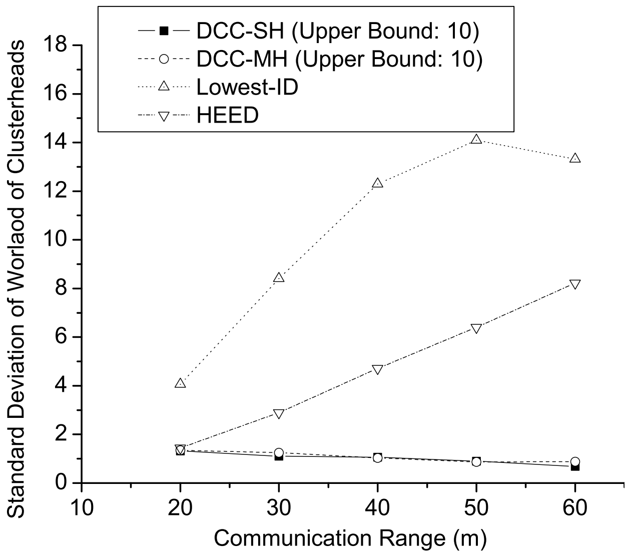


Figure 2.35: Standard Deviation of Clusterhead Workload vs Transmission Range for DCC Performance

Lowest-ID approach because of the extra operations and message overhead for selecting two clusterheads in each cluster. Because HEED has a higher percentage of single-node clusters, the single-node cluster has to communicate to the base station by itself, which leads to more energy consumption. The maximum node energy consumption for both DCC-SH and DCC-MH is less than Lowest-ID and HEED, which shows the effectively balanced energy consumption of DCC. We can infer that DCC will last longer before the first node depletes its energy. Because of the energy consumptions of intermediate forwarding nodes, DCC-MH consumes more energy than DCC-SH.

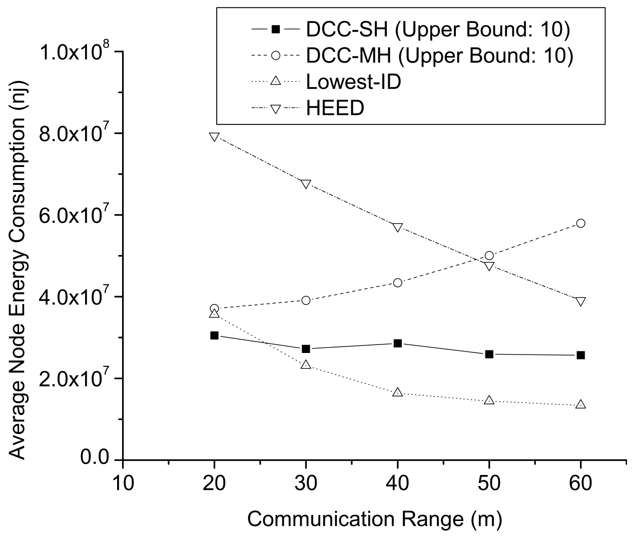


Figure 2.36: Average Node Energy Consumption vs Communication Range for DCC Performance

2.6 Typed Clustering (TC) for Mobile Ad Hoc Networks

2.6.1 Type-related Logic for Clustering

Based on the node type information, we can impose more meaningful and more useful constraints on clustering. In this section we describe the possible constraints that can be imposed on clustering and how to program these constraints using the specification language.

A cluster can be viewed as a collection of typed nodes that satisfy given constraints and are grouped together. For instance, a helicopter carrying a people can be considered a cluster (of two nodes). Later, when the helicopter picks up one more people, the cluster then contains three objects. In our system, clusters do not overlap; i.e., one node cannot appear in more than one cluster at the same time. The cluster information is maintained by the clusterhead in each cluster. Because of nodes moving, the constraints of some clusters may not be satisfied. In this case, the cluster will break up and all nodes become clusterhead by themselves.

To be efficient to describe the clustering process, the specification language should provide the following utilities.

It is desired to get the number of nodes of a designated type. “#” is a prefix unary operator that returns the number of nodes of designated types. For example, for a cluster containing three types of nodes, *A*, *B*, and *C*, `#(A)` will give the number of *A* objects in the cluster.

In some cases, we need to specify the existence of certain nodes that satisfy the given constraints in the

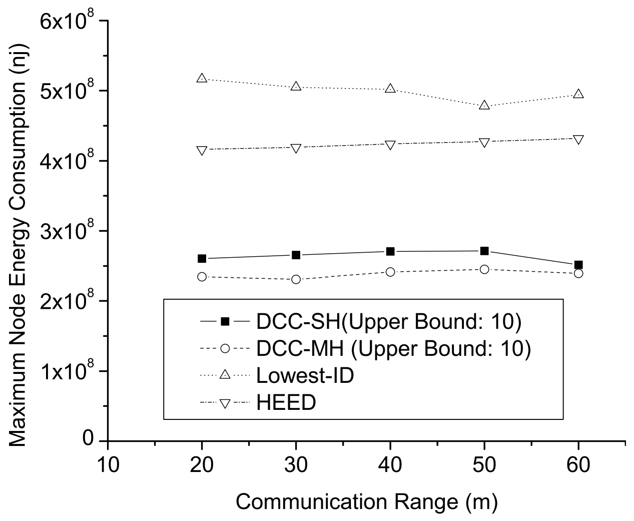


Figure 2.37: Maximum Node Energy Consumption vs Communication Range for DCC Performance

cluster. Adopting the idea of SQL used in database system, we use `Select` statement to check whether there exists such a node that satisfies given requirements in the cluster. The syntax of the `Select` statement is similar to that of the SQL `Select` statement. The `Where` clause is a Boolean expression that determines which node to choose. In some other cases, we need to impose the constraints on all nodes in the cluster. The `SelectEach` statement's syntax is similar to the `Select` statement, except that it is used to check whether all nodes in the given cluster match the constraints in the `Where` clause.

2.6.2 Example Clustering Scenario

To make the description of TC more concrete, we give an example clustering scenario that can be specified using TC. In this example, we have three types of nodes in the network, *A*, *B*, and *C*. For a node n, $Type(n)$ gives the type information of n. We define that priority of a type t node to be a clusterhead as, $Priority(t)$. Nodes of the same type have the same priority. The higher a node's priority is, the more suited the node is to be a clusterhead. For two nodes of the same priority, the node with higher *Weight* value is more suited to be a clusterhead. To balance the cluster size and alleviate the workload of clusterheads, the constraint of each cluster bound the number of each type of nodes within a given range. The maximum number of type t nodes are given as, $Bound(t)$. That is the number of nodes of type t should be less or equal to $Bound(t)$, which can be expressed as $\#(Type(t)) < Bound(t)$.

Construction of Clusters

In this section, we describe TC's multi-hop and hop-bounded clustering process. We assume that all nodes have the same communication range. Each node has a unique ID, a type, and a *Weight* value. The *Weight* can represent the node's capability, resources, and so on. The greater a node's *Weight* is, the more capable that node is and more suited to be a clusterhead.

Initially, all nodes are clusterheads and periodically broadcast CLUSTER messages containing the ID, type, *Weight*, the number of hops that the message has traveled, and the cluster size information.

For a node to join another cluster, TC allows two possible operations, *Join* and *Merge*. The difference between them is that if a node uses *Join*, it just requests to become a member of the new cluster by itself. *Merge* is only used by a clusterhead with some members in its cluster. The clusterhead needs to inform all its members to join the new cluster. Employing *Merge* operation can reduce members' attempts to find new clusters to join, and the changes.

If x, which is a clusterhead by itself, finds that clusterhead z's type is more suitable to be a clusterhead than x, or $Weight(z)$ is greater than $Weight(x)$, and after joining z's cluster the cluster constraint is still satisfied, it will send a JOIN message to z to request to *Join* z's cluster. If clusterhead y, whose cluster size is greater than one, finds that a nearby clusterhead, z, is more suitable to be a clusterhead based on the type information or they are of the same type and $Weight(z)$ is much greater than $Weight(y)$ (exceeding a threshold δ) and after combining the two clusters, the new cluster still satisfies the cluster constraints, y uses *Merge* to combine with z's cluster by sending a MERGE message containing its cluster information. The smaller δ is, the more likely clusters are to combine with each other, leading to fewer clusters and more changes. There is a tradeoff between stability and sensitivity to topology changes.

To reduce the cluster changes happened in the network, we do not allow a member of a cluster to actively leave the current cluster and join another cluster. This can only be performed through the *Merge* operation.

On receiving the JOIN message from x, if z finds that the new cluster's constraints will still be satisfied, it will reply with an ACKJOIN message to accept. Otherwise, z will send a NOJOIN message to refuse.

On receiving the MERGE message from y, if z finds that the new cluster's constraints will still be satisfied, it will send an ACKMERGE message to y to accept. Otherwise, z will send a NOMERGE message to refuse. On receiving the ACKMERGE message, y will inform members in its cluster to join z's cluster.

Members more than H hops away from z will join some other cluster or become a clusterhead.

After sending out the JOIN or MERGE message, if a node cannot receive any reply after a waiting period, it will assume that the message has been lost and try to join some other cluster or remain in its cluster based on its *Weight*.

2.6.3 Clustering For The Example Scenario

In this section, we describe the specific clustering process for the example scenario. The broadcasted CLUSTER messages include information about the number of each type of nodes in the cluster contains. If x, which is a clusterhead by itself, finds that $Priority(z) > Priority(x)$, or $Priority(z) = Priority(x)$ and $Weight(z) > Weight(x)$, and after joining z's cluster, the new cluster's $\#(Type(x))$ is within the bound, $Bound(Type(x))$, it will send a JOIN message to z to request to ***Join*** z's cluster. If clusterhead y, whose cluster size is greater than one, finds that a nearby clusterhead, z, $Priority(z) > Priority(y)$, or $Priority(z) = Priority(y)$ and $Weight(z)$ is much greater than $Weight(y)$ (exceeding a threshold δ) and after combining the two clusters, for each node of type t, the constraint $\#(t) \leq Bound(t)$ is satisfied, y uses ***Merge*** to combine with z's cluster by sending a MERGE message containing its cluster information.

On receiving the JOIN message from x, if z finds that $\#(Type(x)) \leq Bound(Type(x))$ for the new cluster, it will reply with an ACKJOIN message to accept. Otherwise, z will send a NOJOIN message to refuse.

On receiving the MERGE message from y, if z finds that the constraint $\#(t) \leq Bound(t)$ is satisfied, it will send an ACKMERGE message to y to accept. Otherwise, z will send a NOMERGE message to refuse. On receiving the ACKMERGE message, y will inform members in its cluster to join z's cluster. Members more than H hops away from z will join some other cluster or become a clusterhead.

2.6.4 Maintenance of Clusters

Nodes exchange messages to reflect changes in topology. Each clusterhead broadcasts CLUSTER messages periodically. Each member responds with HELLO messages and forwards its clusterhead's CLUSTER messages that are within H hops away from the clusterheads. Thus, members in other clusters cannot be in the communication paths between the clusterheads and members, causing inconsistency. Because of the broadcast nature of wireless communications, nodes in the direct vicinity of a cluster can still receive

CLUSTER messages and have the chance to join. The clusterhead is also responsible to check whether the constraints of clusters are satisfied or not. For the example scenario, the clusterhead needs to check whether the constraint for node type t, $\#(t) \leq Bound(t)$ is satisfied or not. If the constraints are not satisfied because of nodes' moving, the cluster will break up. All nodes become clusterheads by themselves just as the beginning the clustering.

If a member does not receive a CLUSTER message from its clusterhead after $n_{allowed}$ CLUSTER message intervals, it assumes that its clusterhead cannot be contacted and will either compete as a new clusterhead or join some other cluster based on its *Weight*. If a clusterhead does not receive a HELLO message from a member after $n_{allowed}$ HELLO message intervals, it assumes that the member cannot be contacted and will remove it from the member list.

Because of congestions, a CLUSTER message may have traveled more hops before reaching a member even though the clusterhead does not move away. This will cause a member's incorrect update of the number of hops away from the clusterhead. To distinguish the actual movement of nodes from congestions, if a member detects that a CLUSTER message has traveled more hops than the previously received one, it will check with the neighbor from which it receives the previous CLUSTER message. If that neighbor also becomes more hops away from the clusterhead, the member will update the number of hops away information accordingly. Otherwise, it will not update.

2.6.5 Message Overhead

For a network consisting of n nodes and m clusters, during each CLUSTER message interval, because a CLUSTER message will be forwarded by members in that cluster, there are n CLUSTER messages. A HELLO message will be forwarded by each node on the path between the member and the clusterhead. So in response to the CLUSTER messages, there are at most $H(n - m)$ HELLO messages.

2.7 Summary

In this chapter, we describe the various proposed clustering approaches, SMC, BAC, DCC, and TC, for MANET and WSN. All the proposed clustering approaches form and maintain clusters using only local topology information.

By bounding the size of each cluster, SMC alleviates the clusterheads' workload. It achieves fewer clusters for a given upper bound, which is a desirable property for large scale networks. Adopting *RelativeWeight* achieves short communication paths between clusterheads and members. Besides the multi-hop nature of clusters, *GroupJoin* further reduces the number of cluster changes. Simulation results show that the message overhead of SMC is well controlled.

To reduce the message overhead for clustering, BAC makes members forward the maintenance messages probabilistically based on network conditions. BAC's multi-hop nature and *Merge* operation reduce the changes in case of nodes' movement. Different from the *RelativeWeight* of SMC, BAC achieves the compactness of clusters by directly bounding the number of hops between members and clusterheads. BAC achieves better performance on construction and maintenance of clusters, adaptivity to network conditions, and effectiveness in reducing messages with nearly no performance degradation.

By bounding the size of each cluster and rotating the active role between the two clusterheads in each cluster, the workload of clusterheads, nodes' energy consumption, and cluster size are better balanced for DCC than Lowest-ID and HEED. The quasi-one-hop DCC-SH forms more clusters than multi-hop DCC-MH, while the multi-hop forwarding of messages in DCC-MH consumes more energy than DCC-SH and incurs more messages. Despite the multi-hop nature, DCC-MH also achieves compactness of clusters because longer delay in receiving messages from farther away clusterheads makes nodes join closer clusters.

TC takes node type information into account while clustering them to impose more meaningful and more useful constraints on clusters. Based on our recent work on specification language for network computing, we describe how to depict the constraints using the specification language. TC forms and maintains clusters using only local topology information. TC's multi-hop nature and *Merge* operation reduce the changes in case of nodes' movement. TC achieves the compactness of clusters by directly bounding the number of hops between members and clusterheads. To make the description of TC more concrete, we give an example application scenario, in which each type of nodes in a cluster are bounded within a given upper limit. We describe how to construct a specific clustering protocol for such an application.

CHAPTER 3

SPECIFICATION LANGUAGE (NETSPEC) FOR GROUPING OVER NETWORKS

3.1 Overview

With the development in technologies, computers are becoming smaller and more powerful, while wireless communications are becoming faster and more reliable. All these allow users to move around while still be able to have access to computing power and network resources. This has made applications of network systems feasible [51].

It is difficult to design, build, and deploy distributed software systems from network computing systems. Existing approaches to building software system for network computing systems are not proper to handle this, because of the following two challenges as we mentioned in Chapter 1.

One challenge to program for network computing systems is that, programmers may deal explicitly with the under layer network utilities using low level programming languages like C++. Thus programmers focus more on under layer details than functionalities of applications.

Device heterogeneity is another challenge brought by network computing. It is not practical to have all devices in the network system of the same type. Grimm et al [5] propose that programming distributed applications is increasingly unmanageable because of heterogeneity of devices and system platforms. They also point out that this can lead to duplicated different versions of the same application for different computation devices due to the fact that some existing applications are typically developed for specific devices or system platforms.

In distributed computing systems, middlewares like CORBA [52] and Java RMI [6] have been developed to provide a uniform access to resources independent of under layers. Adopting a similar idea, by using high level specification language, programmers can encode all of the necessary functions in one program. A compiler then translates the program into network protocol, which is then deployed on under layer network platforms. This can solve the challenges brought by device heterogeneity in network computing environment and increases the portability and reusability of the application codes. Further, programmers can focus on the functionalities of applications instead of under layer details.

Taking all the above issues into account, we propose NetSpec, a high-level specification language for network computing. The work is based on our group's previous work [7] and to make it more general and more application-oriented. The challenge is to make NetSpec powerful enough to encode complicated applications, and yet simple enough to efficiently parse into network protocols. The targeted applications for NetSpec are network systems containing computing devices that can move in the system area. The identities of devices are not critical to the applications, while types, or functionality of devices matters.

NetSpec is a strongly typed language. Each data type is predefined as part of the programming language. All constants and variables defined for a given program must be described with one of the data types. Integer, Real, Boolean, and String are primitive types. Constants in NetSpec are variables whose values cannot be changed over the execution period of the system.

The basic building blocks of NetSpec are *objects*, which are logical representations of computing and communicating entities in network computing systems. An object is an instance of a *class*. The class construct of NetSpec is similar to the"struct" in C, as it defines an abstract representation of a real world object with attributes. The aggregate of all attributes of an object represents its current state (Note that in BCS [7], objects do not have states). One object's state is different from another by the values of attributes. The attribute values are initialized by physical entities themselves in the network system to reflect the characteristics of them in reality.

Objects are grouped into bonds to show relations among them. A bond [1] is a collection of objects that satisfy a given constraint and are grouped together. For instance, a helicopter carrying a people can be considered a bond (of two objects). Later, when the helicopter picks up one more people, the bond then contains three objects. In our system, bonds do not overlap; i.e., one object cannot appear in more than one bond at the same time. In the network system, it is not practical to adopt a centralized approach to storing bond information. Adopting a distributed method, the bond information is maintained by objects in each bond. Thus, we do not allow the existence of empty bonds for the reason that there is no object to store the empty bond's information.

In NetSpec, a network system has some transitions to operate on bonds and objects. Transitions allow objects to change their states, form new bonds, or join, switch and leave bonds. In our system, transitions are fired non-deterministically. A transition, similar to a bond, has some Boolean expressions that defines

[1]We interchangely use the term bond and group in this dissertation.

its constraints. Once the constraints have been met, the transition can be executed by a run-time mechanism. If two or more transitions are enabled simultaneously, a non-deterministic choice will occur and only one of them will be executed. To ensure the consistency and correctness of the system, operations of transitions are atomic. A transition can be fired only when its constraints are met. The detailed semantics of a NetSpec specification can be found in the next Chapter.

The rest of the this chapter is organized as follows. Section 3.3 describes the illustrated scenarios used throughout the paper to describe how to program such applications using NetSpec. Section 3.4 describes the computation model of NetSpec. Sections 3.5, 3.6, and 3.7 describe the essential parts of NetSpec. Section 3.8 contains the summary. In the appendix, we give the syntax of NetSpec and the complete specifications of the three example applications.

3.2 Related Work

Our model of network computing system is closely related to pervasive computing. Weiser [53] defines pervasive, or ubiquitous computing as the creation of environments saturated with computing and communication capability and gracefully integrated with human users. The basic idea behind network computing is to deploy different kinds of computing devices throughout the designated working or living spaces [5]. In this section, we summarize the current state of pervasive programming approaches, which are also network computing system involving large scale networks. A key issue that programming approaches are trying to address application development complexity too adequately deal with heterogeneous devices, varying degrees of connectivity, and dynamic data sources.

Saif et al. [54] propose that users express the requirements as an abstract high-level goal. Then the system automatically satisfy this goal by assembling, on-the-fly, an implementation that utilizes the resources currently available to the user. This automatic runtime adaptation is the common technique used to convert the device-independent representation to a device-specific representation. They build O2S to offer a general-purpose architectural framework for engineering goal-oriented adaptive systems. Goals are formalized as a language construct and used to guide thee automatic construction of a component-based system. Heuristics are used to choose the a proper implementation of the goal based on the estimate of the most acceptable implementation choice. One of the pitfalls of this approach is its reliance on automatic runtime adaptation of the device-independent representation. It can work when the content is simple or when the device variations are not too great.

Grimm et al. [5] introduce a system architecture for pervasive computing, called *one.world*. This architecture provides an integrated and comprehensive framework for building pervasive applications. They argue for a single application programming interface (API) and a single binary distribution format that can be implemented across the range of devices in a pervasive computing environment. This makes it possible to program once in common API. A single, common binary format enables the automatic distribution and installation of applications.

Kagal et al. [55] propose a policy language designed for pervasive computing applications that is based on deontic concepts and grounded in a semantic language. It consists of a few flexible constructs to allow different kinds of policies to be specified. It allows the security functionality to be modified without changing the implementation of the entities involved. This is similar to NetSpec, while NetSpec focuses on the functionalities of the applications, instead of security policies.

Java's write-once-run-everywhere property allows object code to be moved and dynamically loaded into a process. Based on this, the Jini system aims to provide the minimal set of rules to allow clients and services to find each other and interact [56].

Sivaharan, Blair, and Coulson [57] propose GREEN, a highly configurable and reconfigurable publish-subscribe middleware to support pervasive computing applications. They use an extensible subscription language, FEL to enable the definition of filters for subscription purposes.

Coelho, Anido, and Drummond [58] propose QuickFrame, a development framework, which has a specification language to define application interface and check specific device capabilities. Both QuickFrame and GREEN's specification language is only used to provide a universal method to describe the interface or subscription, instead of specification of system behavior.

Roman et al. [59] propose the Gaia computation environment, which defines a programming environment based on the Model-View-Controller abstraction. Using this abstraction, applications in Gaia are partitioned into four parts: a model to encode the logic of the application, a view to expose the model's state, a controller to map events in the environment of the application as input messages to the model, and a coordinator responsible for storing the bindings of different components in the application's model as well as mechanisms to access and alter these bindings. Gaia uses a high level scripting language, LuaOrb, to capture events and trigger specific actions when certain conditions are met. Thus, compared to NetSpec, LuaOrb's function is limited.

Taking Gaia as a meta-operating system, Ranganathan et al. [60] propose Olympus as a high-level programming model for pervasive computing. It allows developers to program pervasive computing environment without dealing with how common operations are implemented by the under layers. Developers program in terms of virtual entities. The framework takes care of discovering appropriate entities that satisfy developer constraints as well as other constraints in ontologies and user and space policies. The framework also implements some commonly used operations, which are specified as operators in Olympus. Programs developed on the Olympus model consist of two main segments, virtual entities and high-level operators. Olympus takes a more centralized approach to control the entities globally. It does not specify how entities interact with each other distributed.

Carlan et al. [61] propose Aura, which defines a high-level abstraction, *task layer*, to representing user intent. Aura's architecture is focused on adaptation and adjusting performance dynamically.

Chen and Kotz [62] propose the *solar* system framework that allows resourdes to advertise context-sensitive names and for applications to make context-sensitive name queries. It has a specification language that allows composition of context processing operators to calculate the desired context.

3.3 Illustrated Scenarios

To make the description of NetSpec more concrete and more meaningful, we use example scenarios to illustrate how to program them using NetSpec. These examples are used throughout the chapter.

In the helicopter rescue application, the system automatically assigns a helicopter to pick up a sick people and carry him or her to hospitals. When a helicopter carrying a sick people finds that there is a hospital available, it will send that sick people to that hospital and leave the hospital. When a sick people becomes healthy, he or she will leave the hospital.

The highway information system application is to provide an infrastructure to share highway traffic information efficiently. Researchers recently even propose vehicular ad hoc networks (VANETs) [63] to address the communication issues for future vehicular networks. In this highway information system, some special vehicles, information centers, are designated to collect surrounding traffic information and collaborate with each other to exchange the information and disseminate it to vehicles on the road. When a vehicle comes to the communication range of the information center, it will get the traffic information automatically.

The purpose of the pervasive marketing system application is to deploy an infrastructure, based on which customers and venders can trade goods automatically. It has four kinds of objects: customer, bank, vender, and goods. Initially, each customer will get some initial money from a bank. Then the customer will use this amount of money to get some goods from venders. All goods should be got from certain venders.

3.4 Computation Model

Non-deterministic computation. The computation model of NetSpec is based on non-deterministic state machines and assumes no maximal parallelism (Note that P systems [8] assume maximal parallelism). If two or more transitions are enabled simultaneously, a non-deterministic choice will occur and only one of them will be executed. Considering a network computing model that will eventually be implemented over a network, the cost of implementing the maximal parallelism (which requires a global lock) is unlikely realistic, and is almost impossible in real unreliable networks. Hence, in our system, transitions are fired asynchronously.

Sequential And Local Parallel Computations. In a network system, inside a transition, to utilize the power of parallel computation, programmers can specify some local statements to be executed in parallel, provided that the sequence of the statement execution does not affect the final result.

Persistence of Objects. The fundamental building blocks of network systems are objects, which represent physical entities in reality. Thus, in our system, objects cannot evolve into other objects. All objects exist from the beginning to the end.

3.5 Data Type

NetSpec is a strongly typed language. Each data type is predefined as part of the programming language and all constants or variables defined for a given program must be described with one of the data types.

3.5.1 Primitive Data

Integer, Real, Boolean, and String are primitive types. Other complex types can be built based on these. Primitive type is usually used to define attributes of class type, which will be discussed in section 3.5.3.

The primitive type Integer represents the set of integers as defined in mathematics: all positive and negative natural numbers, plus zero. The implementation of the language may somehow limit this set depending on the memory allocated for value, but for the purposes of NetSpec specification, we assume that the Integer type encompasses all mathematical integers.

Similarly, the primitive type Real represents the set of real numbers as defined in mathematics, which can be defined informally as any number that can be given a finite or infinity decimal representation. Again, the implementation of the language may limit this set further to a more finite set of numbers, but this language specification will assume that Real encompasses all real numbers.

String represents a sequence of zero or more (any size, though not infinite) characters, where a character is a symbol that can be given a unique numerical representation.

The primitive type Boolean encompasses a set of two distinct values: “true” and “false”. Every Boolean value is either “true” or “false”, making it the simplest of the primitive types. Often, Boolean values are used to perform logical operations.

```
<primitive_def> ::= <type> <id> ';'
```

The above is the formal syntax of primitive type definition. `<type>` can be Integer, Real, Boolean, or String. `id` is the name of the variable.

```
Boolean healthy;
```

indicates whether a people is healthy or not in the helicopter rescue system.

```
Integer plate_number;
```

indicates the plate number of a vehicle in the highway information system.

3.5.2 Constant

Constants in NetSpec are variables whose values can not be changed over the execution period of the system. A user can define constants with the `const` keyword preceding the definition of Integer, Real, Boolean, and String variables.

```
<const_def> ::= 'const' <type> <id> { '=' <literal> } ? ';'
```

In the constant type definition, `<type>` can be Integer, Real, Boolean, or String. `<literal>` specifies the initial value of the constant. In some cases, the initial value is set by the user. In some other cases, the initial value is set by the under layer system.

For the pervasive marketing application,

```
const Integer initial_money=1000;
```

specifies the initial amount of money each customer can have.

We can also specify the attributes of an object to be constant, which means that after the initialization of the constant by the system, the value remains unchanged.

In the helicopter rescue system, for `Helicopter` class we have

```
const Integer capacity;
```

to specify the maximum number of people a helicopter can hold. Usually, a helicopter's capability is given and remains unchanged during the life cycle of the program. Thus we define it as a constant.

3.5.3 Class and Object

The basic building blocks of a network system specified by NetSpec are *objects*, which are logical representations of physical computing and communicating entities. An object is an instance of a *class*. The aggregate of all attributes of an object represent its current state. One object's state is differentiated from another by the values of attributes. The attribute values are initialized by physical entities themselves in the network system to reflect the characteristics of them in reality. Transitions discussed in section 3.7 can change the values of attributes.

The following is the syntax of the definition of a class.

```
<class_def> ::= 'Class' <id> '{' {<primitive_def> | <const_def>}*'}'
```

As an example, in the following we give the definition of three classes defined in the helicopter rescue system: `People`, `Helicopter`, and `Hospital`.

```
Class People {
    Boolean healthy;
    const Integer age;
}
```

A `People` object represents a physical people in reality. `healthy` attribute indicates the people is healthy or not. We also define `age` attribute to indicate the age of a people.

```
Class Helicopter {
    Integer color;
    const Integer capacity;
}
```

defines an `Helicopter` class. The `color` attribute specifies the color of the helicopter and `capacity` attribute specifies the number of people it can hold. Usually the capacity of a helicopter is given and fixed for a specific helicopter, thus we make it a constant. The initialization of the constant attribute is done by the under layer utilities.

```
Class Hospital {
    Integer ID;
    const Integer capacity;
}
```

defines a `Hospital` class. The `capacity` attribute indicates the number of people it can host. This is also a constant.

From the formal definition of class and the three examples, we can see that the class definition of NetSpec is similar to a “struct” in C, as it defines an abstract representation of a real world object through a series of attributes. A class is a composition type that contains one or more primitive or constant values. Class attributes may only be primitive or constant definitions (i.e. composition of classes within other classes is not possible). We can use the member-of operator, “.”, to access any of the attributes of an object. Class definitions allow us to abstract real world objects, but in NetSpec, we never actually instantiate new objects, nor do we often care about an object’s identity, which is different from the “struct” in C, because of the fundamental difference in the targeted applications.

In the following we define the two classes used in the highway information system, vehicle and information center.

```
Class Vehicle {
    Integer plate_number;
}
```

defines a `Vehicle` class. The `plate_number` attribute indicates the plate number of the vehicle.

```
Class Infocenter {
    Integer capability;
}
```

defines the `Infocenter` class to represent information center. It has the `capability` attribute to indicate the capability of the information center. The larger the `capability` value is, the more capable the information center is.

In the following we describe the definitions of the three classes used in the pervasive marketing system: bank, customer, and vender.

```
Class Bank {
    Integer money;
}
```

defines a `Bank` class. The `money` attribute indicate the amount of money the bank has.

```
Class Customer{
    Integer money;
    Boolean initialized;
}
```

defines a `Customer` class. The `initialized` attribute indicates whether the customer has got some initial amount of money from a bank.

```
Class Vender{
    Integer money;
}
```

defines a `Vender` class. The `money` attribute indicates the amount of money the vender currently has.

3.6 Set and Bond

In NetSpec, operations are performed on objects. Thus, we need to indicate which objects should perform the designated operations. It is inconvenient and sometimes impossible to enumerate every single involved object. In our network computing system, an object is typed but not addressed. Thus it is desirable to organize objects of the same or close properties into sets or bonds and operate on them. Both set and bond are used to specify a group of objects. The difference between them is that in a set, all objects are of the same type and there is no constraint on the relations among the objects. While for a bond, it can contain objects of different data types and there are constraints on the relations among objects in the bond.

3.6.1 Set

Whenever we define a new class type, the language will also provide a set type associated with the class. A set is an aggregation of instantiations of a single class type. The following is a formal definition of a set

```
<set_type> ::= <id> ’Set’
```

When we define a class type of `<id>`, `<id> ’Set’` is the set type that contains the objects of type `<id>`. Every time a new class type is defined, an associative set type is also created for the class. Users do not need to explicitly declare the set type. The order and identity of the objects within a set are not known by the set itself; rather, the set merely serves as a way to group objects of the same class type together.

In the helicopter rescue application, after we define `People` class, `People Set` represents a collection of `People` objects. Similarly, we have `Helicopter Set` and `Hospital Set`. For the high-way information application, we have `Infocenter Set` and `Vehicle Set`. For the market application, we have `Bank Set`, `Customer Set`, `Vender Set`, and `Goods Set`.

3.6.2 Bond

A bond is a collection of objects that satisfy given constraints and are grouped together. Each bond has an argument list to specify the involved objects. We use sets in the argument list to specify the involved objects. Thus, a bond can be viewed as a super set that contains several subsets of objects. The body of a bond’s definition consists of a sequence of `BondSelect` or `BondSelectEach` statements to define relations among objects that are specified. All involved object instances are subject to these constraints. In each of the `BondSelect` or `BondSelectEach` statement, the `<expr>` can only reference the variable in its own statement or those in the header of the bond definition. In the following is the formal definition of bond types.

```
<bond_def> ::= 'Bond' <id> '(' <set_type> <id> {',' <set_type>
<id>}* ')' '{' <bond_select_stmt>* <expr> ';' '}' }
```

In the following, we describe the two bonds used in the helicopter system.

```
Bond People_Helicopter (People Set ps, Helicopter Set hes) {

    BondSelectEach (People p) From ps
    Where p.healthy==false && #(ps)>=1;

    BondSelect (Helicopter h) From hes
    Where #(hes)==1 && #(ps) <=h.capacity;
}
```

The constraint of `People_Helicopter` bond requires that it should have one helicopter and at least one people. The number of `People` objects in the bond should also within the capacity of the helicopter. Objects in the bond are divided into two sets, `ps` and `hes`. `BondSelect` operation non-deterministically chooses a `Helicopter` object to operate. `BondSelectEach` operation specifies that for all `People` objects, the constraints should be satisfied. `BondSelect` and `SelectEach` are explained in more details in Section 3.6.3.

```
Bond Hospital_People (Hospital Set hos, People Set ps) {

    BondSelectEach (People p) From ps
    Where p.healthy==false || p.healthy==true;

    BondSelect (Hospital h) From hos
    Where #(ps)<=h.capacity && #(hos)==1 && #(ps)>=1;
}
```

`Hospital_People` bond specifies that one hospital and at least one people can form the bond. The number of people should be within the capacity of the hospital. Because a hospital can also host people who are just getting well, we have the constraint

```
    (p.healthy==false||p.healthy==true),
```

which is simply true, to allow both healthy people and sick people to stay in the hospital.

From the above syntax definition and examples, we can see that `<set_type> <id>` specifies the set of objects that can be contained in this bond. The `BondSelect`, `BondSelectEach` statements, and the expressions specify the constraints on objects in the bond.

In some applications, we need to check whether an object belongs to a bond or not. Thus each object has a default attribute `Boolean bonded` to indicate whether the object is bonded or not. If an object has not become a member of any bond, its `bonded` attribute's value is false. Otherwise, it is true. The value of `bonded` is set by the system instead of the user. Programmers cannot change it explicitly in the program. Usually the value of `bonded` is set when executing the operation of *Join*, *Leave*, or *Switch*, which are described in Section 3.7.

3.6.3 Basic Operations on Set and Bond

`#` is a prefix unary operator that returns an Integer expression as the number of objects in the set. For example, for `People Set ps`, `#(ps)` will give the number of `People` objects in `ps`.

The bond and set type can specify a group of objects. In some operations, we also need to specify the objects involved.

A `BondSelect` statement arbitrarily chooses an object from the set provided. Since objects are not identifiable or enumerated within sets and bonds, this is the only way to choose and operate on individual objects in a set or bond. The syntax of the `BondSelect` statement is similar to that of the SQL Select statement. The `Where` clause is a Boolean expression that determines which object to choose. If multiple objects in the set meet the constraints provided, one is chosen arbitrarily. Alternatively, the `Where` clause can be omitted to randomly select any one object. Every identifier used in a transition must first be bound by a `BondSelect` statement. The `BondSelectEach` statement's syntax is similar to the `BondSelect` statement, except that it is used to select all objects within the given set that match the constraints in the `Where` clause.

In the following we give the formal syntax of `BondSelect` and `BondSelectEach` statements.

```
<Bond_Select_stmt> ::= <bondselect> ’(’ <type> <id> {’,’ <type>
<id>}* ’)’ ’From’ <location> {’Where’ <expr>}? ’;’

<bondselect> ::= ’BondSelect’ | ’BondSelectEach’
```

3.6.4 More Bond Examples

In this section, we describe the other bonds used in the illustrated examples.

```
Bond Infocenter_Vehicle (Infocenter Set is, Vehicle Set vs) {
    BondSelect From Where #(is)==1 && #(vs)>=1;
}
```

`Infocenter_Vehicle` bond specifies that one information center and at least one vehicle can form the bond.

```
Bond Vender_Goods(Vender Set vs, Goods Set gs) {
    BondSelect From Where #(vs)==1 && #(gs)>=1;
}
```

`Vender_Goods` bond specifies that one vender and at least one goods can form this bond.

```
Bond Customer_Goods(Customer Set cs, Goods Set gs){
    BondSelect From Where #(cs)==1 && #(gs)>=1;
}
```

`Customer_Goods` bond specifies that one customer and at least one goods can form this bond.

3.7 Transitions

Transitions operate on bonds and sets of objects. It allows objects to change their state, form new bonds, or join, switch and leave bonds. In our system, transitions are fired non-deterministically. A transition, similar to a bond, has a Boolean expression that defines its constraint. To ensure the consistency and correctness of the system, the operations of transitions are atomic. The constraints of the transition and the bonds that are involved will only be check before and after the firing of the transition. During the execution of the transition, relations among objects are being adjusted. In this process, some constraints may not be satisfied.

Once the constraints of a transition have been satisfied, it can be executed by the run-time mechanism. If two or more transitions are enabled simultaneously, a non-deterministic choice will occur and only one of them will be executed.

```
<transition_def> ::= 'Transition' <id> '(' <transition_param> <id>
                     {',' <transition_param> <id>}* ')' '{'
                     <select_stmt>* 'if' '(' <expr> ')' '{'
                     <transition_stmt>* '}' '}'
```

The above is the syntax of the definition of a transition. In the following we describe one of the transitions in the helicopter rescue application.

```
Transition T1 (People Set ps, People_Helicopter ph) {
    Select (People p) From ps;
    Select (Helicopter h) From ph;
    if(p.healthy==false && #(ph.ps)<h.capacity)
    {
        p join ph;
    }
}
```

`T1` states that when an unbonded sick people finds that there is a `People_Helicopter` bond and the bond is not full, the sick people will join the `ph` bond and become bonded. The two arguments passed to the transition are `People Set` and `People_Helicopter`, which specify the involved objects. The transition first selects `People` object `p`. If the constraint `(p.healthy==false && p.bonded==false)` is true, `p` will join the `ph` bond. For `ph` bond to accept the join request, it needs to be not full. Thus we will check the constraint `#(ph.ps)<h.capacity`. `#(ph.ps)` gives the number of `People` objects in the `ph` bond.

Thus we can see that in the syntax of the transition definition, `<transition_param>` specifies the involve objects, using sets or bonds. The `Select` statements choose the individual objects to perform the operations. The `if` statements specify the constraints to fire it. The `<expr>` of the `if` statement is a boolean expression that specifies the constraints among the involved objects and bonds. The `<transition_stmt>` describe how to adjust the relations among the involved objects and bonds. It can contain the `new` statement to create new bonds.

A `Select` statement arbitrarily chooses an object from the set provided. The syntax of the `Select` statement is the same as `BondSelect`. Below we give the formal definition of `Select` statements.

```
<select_stmt> ::= <Select> ’(’ <type> <id> {’,’ <type> <id>}* ’)’
’From’ <location> {’Where’ <expr>}? ’;’
```

The `<type> <id>` specifies the chosen objects' handles. `<location>` is the set of objects or bonds. The `<expr>` in a `Select` statement is a boolean expression that defines the constraints on the objects that can be chosen.

The difference between `Select` and `BondSelect` is that `Select` is used to choose an operational objects from a given set or bond. In fact, `BondSelect` defines an $\exists$-quantifier.

For example, if we want to operate on a `People` object with age above 18, we can program as:

```
Select (People p) From ps Where p.age>18
```

If we want to choose a `People` object with the name Smith from the `People Set ps`, we can use the following codes:

```
Select (People p) From ps Where p.name==Smith
```

A new bond can be created with a “new” statement and so long as its constraints are satisfied, the bond remains. Because we do not allow empty bonds, when creating a bond, we should specify the objects that will form the bond.

```
new Bond People_Helicopter ph(p, h);
```

creates a new `People_Helicopter` bond instance `ph` with two objects, `p` and `h`. An example transition in the helicopter application, `T2`, shows the usage of the *new*.

```
Transition T2 (People Set ps, Helicopter Set hes) {

    Select (People p) From ps;
    Select (Helicopter h) From hes;
    if(p.healthy==false)
    {
        new Bond People_Helicopter ph(p, h);
    }
}
```

`T2` states that when an unbonded sick people finds that there is an unbonded helicopter, the sick people and the helicopter can form the bond `People_Helicopter` and become bonded. The two `select` statements choose one `People` object and one `Helicopter` object respectively. Then we test the constraint `(p.healthy==false)`. If this is true, this means that the people is sick. Then we will create a new `People_Helicopter` bond, `ph` with the selected `People` object and `Helicopter` object.

Three basic operations, *Join*, *Switch*, and *Leave*, are fundamental to the adjustment of relations between objects and bonds.

- *Join* operation is for an unbonded object to join a bond and become bonded.
- *Switch* is for a member of a bond to leave the current bond and become a member of another bond.
- *Leave* is used by a member to leave its current bond and become unbonded.

An example transition in the helicopter application, `T3`, shows the usage of the *Switch*.

```
Transition T3 (People_Helicopter ph, Hospital_People hp) {
    Select (People p) From ph;
    Select (Hospital h) From hp;
    if(#(hp.ps)<h.capacity)
    {
        p switch hp;
    }
}
```

T3 states that when a helicopter carrying sick people finds that there is a hospital available, it will send the sick people to that hospital. It selects a People object using Select (People p) From ph. Then we need to test whether the hospital is full by checking the constraint #(hp.ps) $<$ h.capacity. If this is true, because p previously belongs to ph bond, p switches to hp bond. If the helicopter is carrying more than one people and T3 is further executed, more people will be sent to the hospital.

In NetSpec, if the order of the execution of some statements does not affect the result, to fully utilize the computation capability of distributed devices, some statements in a transition can be executed in parallel manner. These statements are specified using doparallel keyword. Otherwise, by default, statements are executed sequentially.

```
doparallel {
    Statement1;
    Statement2;
    Statement3;
    Statement4;
}
```

For example, the above codes specify that Statement1, Statement2, Statement3, and Statement4 can be executed at the same time (i.e., using the interleaving semantics, these statements can be executed in any order.).

3.7.1 Helicopter Rescue Application

In the following we show how to program the transitions for the helicopter rescue application using NetSpec.

```
Transition T1' (People Set ps, Helicopter Set hes) {
    Select (People p) From ps Where p.name=Smith;
    Select (Helicopter h) From hes Where h.color=red;
    if(p.healthy==false)
    {
        new bond People_Helicopter SpRh(p,h);
    }
}
```

`T1′` states that when an unbonded sick people named as Smith finds that there is an unbonded red helicopter, the people and the red helicopter form the bond `SpRh`. The boolean constraint can be placed in the `Select` statement. It can also be placed in the `if` statement after the `Select` statement as the following shows.

```
Transition T1′′ (People Set ps, Helicopter Set hes) {
    Select (People p) From ps;
    Select (Helicopter h) From hes;
    if(p.healthy==false && p.name=Smith && h.color=red)
    {
        new bond People_Helicopter SpRh(p,h);
    }
}
```

The difference between `T1′′` and `T1′` is that the `Helicopter` object chosen by `T1′` is better refined than that chosen by `T1′′`. Thus it is more likely that `T1′` will finish successfully than `T1′′`. It is preferable to place more restricted constraints in the `Select` statements to reduce the chance of transition rollback because of unsatisfied constraints.

```
Transition T4 (People_Helicopter ph, Hospital Set hos){
    Select (Hospital hospital) From hos;
    Select (People p) From ph;
    new Bond Hospital_People hp(p, hospital);
}
```

`T4` states that when a helicopter carrying a sick people finds that there is an unbonded hospital available, it will send the sick people to that hospital. We first use `Select (Hospital hospital) From hos` to select a `Hospital` object. Then we use `Select (People p) From ph` to select a people object from the `People_Helicopter` bond. Then we create the new bond `Hospital_People hp`. Because previously the `People` object `p` belongs to `ph` bond, `p` will switch to new bond hp. If the helicopter is carrying more than one people and `T4` is further executed, more and more people will be sent to the hospital, until there is no people on the helicopter.

```
Transition T5 (Hospital_People hp) {
    Select (People p) From hp;
   if (p.healthy==true)
   {
      p leave;
   }
}
```

`T5` states that if a people in a hospital becomes healthy, the people will leave the hospital and become unbonded. If the hospital has no people, the bond will break up, though this circumstance seldom happens in reality.

3.7.2 Highway Information Application

In this section we show how to program transitions for the highway information application using NetSpec.

```
Transition T1 (Infocenter_Vehicle iv, VehicleSet vs) {
    Select (Vehicle v) From vs;
    v join iv;
}
```

`T1` states that if an unbonded vehicle comes into the communication range of a bond consisting of information center and vehicle, `iv`, it will join the bond.

```
Transition T2 (Infocenter Set is, Vehicle Set vs) {
    Select (Vehicle v) From vs;
    Select (Infocenter i) From is;
    new Infocenter_Vehicle iv (i,v);
}
```

Similarly, `T2` states that if an unbonded vehicle comes into the communication range of an unbonded information center, they will form the new bond, `iv`.

3.7.3 *Pervasive Marketing Application*

In this section we describe how to program the pervasive marketing application using NetSpec.

```
Transition T1(Customer Set cs, Vender_Goods vs){
    Select (Customer c) From cs;
    Select (Goods g) From vs;
    select (Vender v) From vs;
    if(c.money>=g.value)
    {
        c.money-=g.value;
        v.money+=g.value;
        new Bond Customer_Goods cg (c,g);
    }
}
```

`T1` states that when an unbonded customer has the sufficient amount of money, the customer can get some goods from a vender. The customer will transfer the amount of money equal to the value of the goods to the vender.

```
Transition T2(Customer_Goods gs, Vender_Goods vs){
    Select (Customer c) From gs;
    Select (Goods g) From vs;
    select (Vender v) From vs;
    if(c.money>=g.value)
    {
        c.money-=g.value;
        v.money+=g.value;
        g switch vs;
    }
}
```

T2 states that a customer with some goods can still get some goods from a vender.

```
Transition T3 (Bank Set bs, Customer Set cs){
    Select (Customer c) From cs;
    Select (Bank b) From bs;
    if(c.initialized==false)
    {
        b.money-=initial_money;
        c.money=initial_money;
        c.initialized=true;
    }
}
```

T3 is to let the customer get some initial amount of money from the bank. The initial amount of money that a customer can get is defined as a constant value.

3.8 Summary

This chapter introduces NetSpec, a specification language for grouping over networks. It is a script language of BCS, which has well defined semantics. NetSpec is designed to solve the problems brought by network systems, such as low level detail-oriented programming, device heterogeneity, and the poor portability of codes. Using NetSpec, programmers can encode the necessary functions in a NetSpec specification. A compiler then translates the specification into a program running on a network virtual machine which again can be implemented through different network protocols that can be deployed on different under layer network platforms, which will be discussed in detail in Chapter 4. To make the description of NetSpec concrete, we use concrete illustrated examples, such as helicopter rescue system, highway information system, and pervasive marketing system, to show how to program them using NetSpec. The complete syntax definition and the examples specified in NetSpec can be found in the appendices.

CHAPTER 4

VIRTUAL MACHINE OVER NETWORKS AND ITS INSTRUCTION SET

4.1 Overview

In this chapter, we propose a virtual machine over networks. The virtual machine stays in between a NetSpec and the under layer network. That is, after users specify a network application using NetSpec, a compiler can then translate the specification into a targeted program called a *instruction sequence program* running on the virtual machine. The virtual machine separates the program from the under layer network services. As long as the virtual machine supports its own instruction set, the program encoded in the instruction set can run on different networks, which eliminates the redundant effort in developing different versions of the same application for different networks. The instruction set is powerful enough to encode complicated applications, and yet simple enough to efficiently parse into network protocols.

We propose how to construct the compiler that can translate the specification codes written in NetSpec into virtual machine instructions. Then, we describe how to implement a non-deterministic scheduler to ensure the fairness when firing transition code blocks.

To support the instruction set, the network virtual machine has to deal with synchronization, group communication control, and concurrency control. To execute an instruction, the system need locate the involved entities. To maintain a bond, a leader election mechanism is necessary to elect a proper entity to maintain the bond information. To ensure the correctness of an instruction execution, entities should be locked before the execution. Different from programs run on a single processer, the virtual machine provides mechanisms to achieve the sequentialism of instruction executions in network computing systems. The system provides a concurrency mechanism for interactions among transition executions. To evaluate the performance of the proposed solution in real networks, we used NS-2 [43] to simulate an approximated implementation of the helicopter rescue system. The results demonstrate that the proposed solution works well even in unreliable wireless networks.

The rest of the this chapter is organized as follows. Section 4.2 describes the related work. Section 4.3 describes the assumptions of the network virtual machine. Section 4.4 describes the definitions of terminologies used in this chapter. Section 4.5 describes the example scenario used in the chapter to show how the network virtual machine operates. Section 4.6 describes the layout of a typical instruction sequence program and the typical functionality blocks included in a instruction sequence program. Section 4.7 describes the instruction set. Section 4.8 describes how to implement the translator to translate an NetSpec specification into an instruction sequence program. Section 4.9 gives the translated transition code blocks for the example helicopter rescue system. Section 4.10 describes how the transition code blocks are chosen to run in the system. Section 4.11 and Section 4.12 discusses the implementation issues of the instruction set. Section 4.13 describes the simulation of the helicopter rescue system. Section 4.14 contains the summary of this chapter.

4.2 Related Work

When we design and investigate implementation issues of the network virtual machine, we find close relationships with the group and resource management approaches used in existing research of group communications, peer-peer networking, and mobile agent systems.

Group communication is a means for providing communications between multiple entities and multiple entities by organizing them in groups [64]. It provides membership and reliable multicast services. The task of a membership service is to maintain a list of currently active and connected entities in a group. Initially, group communication systems were developed to facilitate the development of fault-tolerant distributed systems. It includes replication using a variant of replication approach [65]. Recently, group communication systems have been exploited for collaborative computing [66].

Cristian [67] first defined membership services for synchronous distributed systems. He and Schmuck [68] then consider timed synchronous systems. Fischer, Lynch, and Paterson [69] show that consensus is impossible in fully asynchronous system even if only one process may fail. Later on, Chandra et al. [70] prove that the primary-partition group membership problem cannot be solved in asynchronous systems with crash failures, even if one allows the removal or killing of non-faulty process that are erroneously suspected to have crashed. Despite the awareness of the original impossibility results and its potential applicability to membership services, Isis [71] and Tansis [72] perform significant work in asynchronous system and make use of membership services.

Peer-peer networking has recently emerged as a new means for building distributed networked applications. It is a completely decentralized network of nodes each of which can act both as a server and as a client. What peer-peer networking shares with our network computing system is that it involves the management of peers to accomplish a common task. Peer-peer system deals with resource locating, which can be used by the network virtual machine system to locate specific entities. Ge et al. [73] classify the currently proposed peer-peer file sharing systems into three different categories. The earliest design uses a central server to coordinate participating nodes and to maintain an index of all available files being shared. When a peer node joins the system, it contacts the central server and sends a list of the local files that are available for other peers to download. To locate a file, a peer sends a query to the central server, which responds with a list of peers that have the desired file. If a peer leaves the system, its list of shared files is removed from the central server. An example of such a system is the Napster network [74].

The other two kinds of approaches distribute the indices of available files among participating nodes. They differ from each other in how they distribute the file indices. In one approach, each peer is responsible for maintaining the indices of only the files it stores. A limited-scope query message is flooded to the network when a peer wants to locate a file in the system. An example of such a system is Gnutella network [75]. The third approach eliminates flooding by systematically distributing the file indices among participating nodes, with queries being routing directly to the node responsible for that subset of the file index. This approach is used by Chord [76], CAN [77], and Pastry [78].

Straber, Baumann, and Hohl [79] model agents as clusters of objects without references to the outside. The agent is the transitive closure over all the objects the main agent object contains a reference to. They can communicate with other agents either locally inside one location or globally with agents on other locations. Mobile Agents furthermore can migrate from one location to another. Mechanisms for the communication between agents and for the migration of agents have to be provided by the Mobile Agent System.

Baumann and Radouniklis [80] propose to group agents in mobile agent environment. Agent groups consist of agents working together on a common task. Each agent works on a subtask. In order to perform their subtasks, agents themselves may dynamically create subgroups of agents. The system has three kinds of group entities, which are group initiator, group member, and group coordinator. It provide means for the group members to communicate, get synchronized, and get terminated.

Satoh [81] presents a framework of reusable mobile agents to manage clusters. The framework enables a mobile agent to be composed of two layered components which are mobile agents. The first is a carrier for the second over particular networks independent of any management tasks and the latter defines management tasks performed at each host, independent of any networks. The frameworks also offers a mechanism for matchmaking the two components. This involves selecting mobile agents according to given criteria. It matchmakes between task agents and navigator agents by comparing the itineraries of the navigator agents. It provides a specification language for the itineraries of mobile agents.

The process over networks emphasizes on high-level programming approaches for network applications. It involves the specification language, which provides a method to describe network applications, and the compiler, which converts the upperlayer specifications into the instruction sequence program that can run on network supporting the instruction set. Mobile agent technology provides a method to migrate executable codes among computing devices in the network. Its essential idea is used by the network virtual machine, which is a part of the framework, as the underlay implementation utilities.

4.3 Assumptions

In the following, we describe the assumptions of the virtual machine over networks.

Broadcasting of status. All the operations are based on message exchanges. Entities in the network broadcast their attribute values, or state, information periodically. Through the reception of nearby entities' state information, an entity can know its surrounding environment's information.

Reliable communications. Entities in the network can interact with each other through the under layer communication network. Cooperation is modeled with implicit message passing rather than with data sharing. To simplify the discussion, we assume that messages can be reliably sent and received. In reality, reliable communications can be achieved by under layer network protocols, such as TCP.

Asynchronous processing. We make no assumptions about the relative speeds of processes or about the delay time in delivering a message.

Global clock. All entities are in a fully distributed system. To use the time-out algorithms to detect the inactivity of entities, we assume that processes have access to a synchronized global clock. Researchers have developed successful clock synchronization protocols for computer networks over the past few decades.

Current clock synchronization protocols for wireless networks can be classified into two categories: probabilistic [82, 83] and deterministic approaches [84].

4.4 Definitions

In this section, we describe the definitions of some terminologies used in this chapter.

Bond Leader. An object which is responsible for maintaining the information of its bond. It also controls the bond operation. In each bond, there is only one node elected as the bond leader.

Instruction Sequence. A sequence of codes that describe the functions in the provided instruction set.

Transition Code Block. A transition code block is defined as an execution of a sequence of instructions, which is the counter part of the transition in NetSpec. A transition code block consists of a sequence of instruction, which may be internal computation, message transmissions, or changes to the membership of bonds that it creates or joins.

Transition Leader. An entity which is responsible for controlling the execution of the transition code block. For each transition code block, at each phase of the execution process, there is only one object as the transition leader.

Instruction Issuer. An instruction execution may involve several objects. An instruction issuer can be an unbonded object or a bond leader, that starts the execution of a single instruction.

4.5 Illustrated Scenarios

To make the description of the virtual machine's operations more concrete and more meaningful, we use an example scenario to illustrate how to use the supported instructions to program them. This example is used throughout the chapter. In the helicopter rescue application, the system automatically assigns a helicopter to pick up a sick people and carry him or her to hospitals. When a helicopter carrying a sick people finds that there is a hospital available, it will send that sick people to that hospital and leave the hospital. When a sick people becomes healthy, he or she will leave the hospital. In Chapter 3 we have given the specification of the helicopter rescue system. Figure 4.1 depicts the relation between the general functionality blocks and the specifications. A specification consists of essentially data type definitions, bond definitions, and transitions. We treat data type definition and bond definition differently.

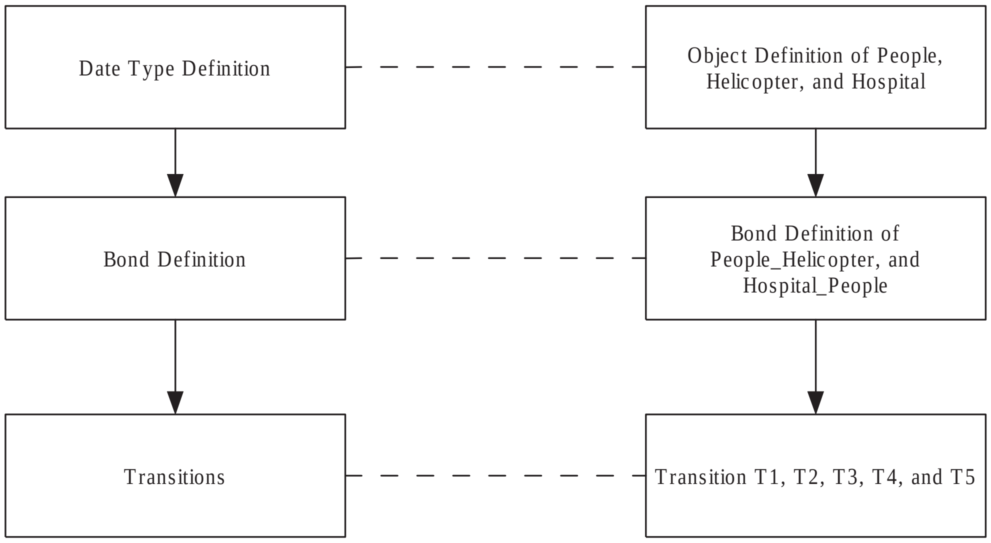

Date Type Definition
Object Definition of People, Helicopter, and Hospital
Bond Definition
Bond Definition of People_Helicopter, and Hospital_People
Transitions
Transition T1, T2, T3, T4, and T5

Figure 4.1: Generalization of NetSpec Specification

4.6 Instruction Sequence Program Layout

In Figure 4.2 we depict the relation between a specification and a typical instruction sequence program. In the left is the specification layout. In the right is the instruction sequence program layout. The *data type definition* part defines all the data types used in the program, which may be classes or primitive data types. The *bond definition* part defines all the bond types used in the instruction sequence program. The bond definition can be used to check the bond integrity. The transitions in NetSpec are translated into transition code blocks in the instruction sequence program. One transition in NetSpec has one corresponding transition code block. A transition code block is independent from other transition code blocks. It is fired individually. An instruction sequence program consists of a sequence of instructions, which will be described in detail in Section 4.7. Section 4.8 describes how to translate each part in NetSpec into instructions supported by the virtual machine. Section 4.10 describes how those instruction programs run on the virtual machine. Further, we discuss the implementation issues of the instruction set in Section 4.11 and Section 4.12.

Figure 4.3 depicts the relation between a transition in NetSpec and a transition code block in instruction sequence program. In the left of Figure 4.3 is the transition in NetSpec. It includes transition header, selection statements, and conditions and operations. The transition header specifies the involved set of objects and bonds. The selection statements will find the involved objects. If the conditions are satisfied,

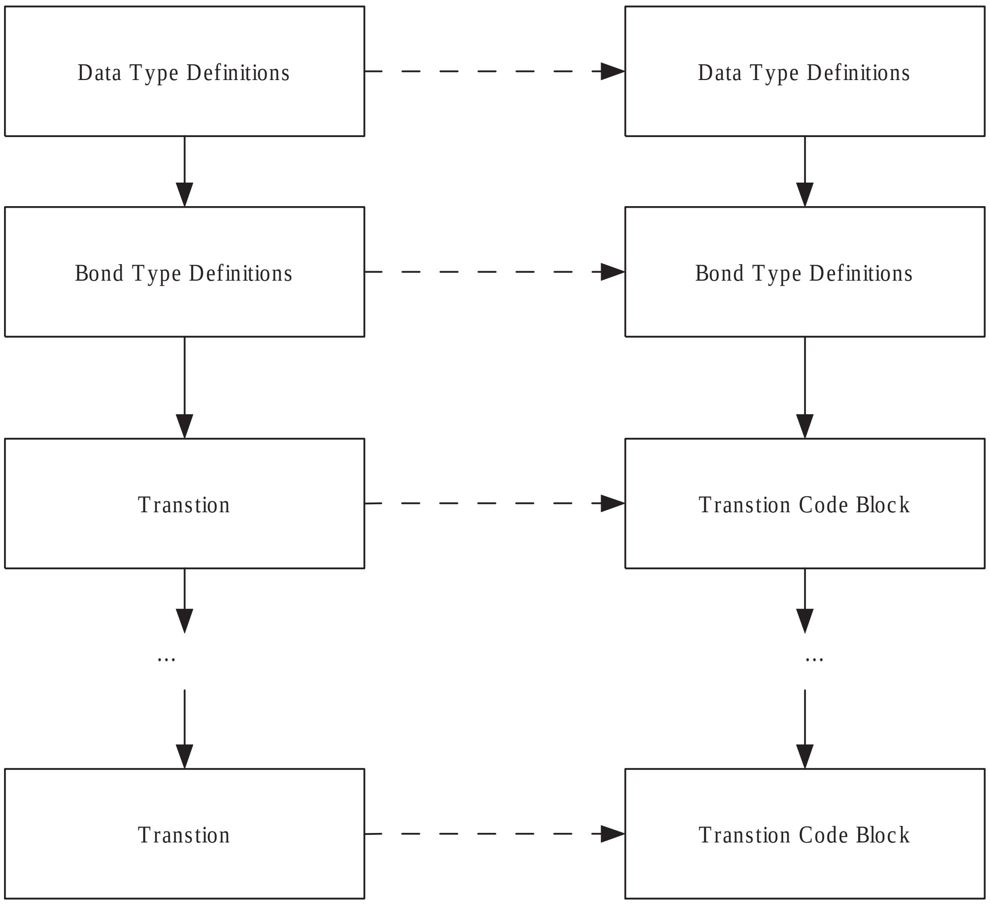


Figure 4.2: Specification Translated Into Instruction Sequence Program

operations on these objects will be performed.

In the right of Figure 4.3 is the transition code block in the instruction sequence program. The start point of the execution of an instruction sequence program is always a starter. It is decided through the execution of $test$ instructions, which check whether the given constraints are satisfied. The $test$ instructions are generated from the header of a transition and the `Selection` statements in the specification. If the constraints are satisfied, instructions following it will be executed. Otherwise, the instruction sequence will not start to execute. The instruction issuer of the $test$ instruction is the first transition leader of the program, or the starter. During each phase of the instruction execution, there is only one transition leader, while the leadership may be transferred among entities in the network. In the *Looking for and Lock Entities* part, the transition leader of the transition code block starts to find and lock the involved entities. Depending on different networks, the under layer identities of entities may be in different formats. Although BCS has the nature of single process, to make the proposed solution more applicable, the virtual machine still provides locking mechanisms to ensure the correctness of operations. Locked entities are only allowed to interact with other locked entities in the same locked group. We need to ensure the sequentialism of the instruction executions. The transition leader is responsible for controlling the transition code block execution. After the completion of the execution of instructions, all the involved locked entities are unlocked for other operations.

4.7 Instruction Set

In this section, we describe the instruction set of the network virtual machine. To support the instruction set, the network virtual machine deals with the underlay network services, which will be described in detail in Section 4.11.

4.7.1 test

$test(entity, [condition])$

$test$ instruction tests whether the entity, which can be an object or a bond, satisfies the given condition. This is the entry point of all instruction sequence programs. Usually, an instruction sequence program has several $test$ instructions. If one of the test instruction's constraint is true, the instruction sequence program will be executed. Otherwise, the execution will be aborted. The $condition$ can be expressed in the following format.

```
[attribute1==value1 op attribute2==value2 op ...]
```

where `op` stands for boolean operation. It can be && standing for logical *and*, or `||` standing for logical *or*. Usually we test object type in the $condition$ part.

4.7.2 findlocknode

$destination = findlocknode(condition)$

The $findlocknode$ instruction is used to find and lock the unbonded destination that satisfies the given conditions. It returns the handle in $destination$, which points to an object that satisfies the given condition in the network. If more than one objects satisfying the condition are found, a random one is chosen. The condition is a boolean expression that determines which object to choose.

To ensure the atomism and correctness of operations,when an entity is found, it will also be locked. Only operations among the locked entities are allowed after they are locked. The locked entities cannot interact with other entities in the system.

The $condition$ can be expressed in the following format.

```
[attribute1==value1 op attribute2==value2 op ...]
```

where `op` stands for boolean operation. It can be `&&` standing for logical and, or `||` standing for logical or.

For example, if we want to find one people above age 18 and named Smith, we can program the condition as:

```
[type==People && age>18 && name==Smith]
```

The `type` attribute is default to each object, which represents the class type of the object.

4.7.3 findlockbond

$destination = findlockbond(condition)$

The $findlockbond$ instruction is used to find and lock the bond that satisfies the given conditions. It returns the handle, $destination$, which points to a bond that satisfies the given condition in the network. If more than one bonds satisfying the condition are found, a random one is chosen. The condition is a boolean expression that determines which object to choose. It can be specified in a similar way as the $findlocknode$ instruction.

4.7.4 findinternalnode

$destination = findinternalnode(bond1, condition)$

The $findinternalnode$ instruction is used to find a member that satisfies the given conditions in the bond, $bond1$. It returns the handle in $destination$, which points to a member that satisfies the given condition. Because the $findinternalnode$ instruction operates on a locked bond, it is not necessary to have redundant lock here. The condition is a boolean expression that determines which node to choose. It can be specified in the same way as $findlocknode$. If more than one members satisfying the condition are found, a random one is chosen. If there is no member satisfying the condition, the destination will be an empty handle.

4.7.5 *numnode*

$num = numnode(objectset)$

$numnode$ instruction is to obtain the number objects of given type in a bond. It is equivalent to the # operator in NetSpec.

4.7.6 *bond*

$handle = bond(ent1, ent2, ent3, ..., condition)$

The $bond$ instruction is used to bond entities together. The entities can be nodes or bonds. If the entities are objects, the objects will form a new bond. If one of the entity is a bond, the other entities will join the bond. The $handle$ points to the result bond. The $condition$ specifies the constraints that the new bond should satisfy. Usually the $condition$ contains the bond type of the newly created bond. The bond constraints can be inferred from the bond definition. In the following we give the definition of $condition$

```
[field1==value1 op field2==value2,...]
```

where `op` can be && and `||`, which are logic “and” and “or”.

4.7.7 *leave*

$leave(source)$

$leave$ is used by $source$ object to leave its current bond and become unbonded.

4.7.8 *switch*

$switch(source, destination)$

The $source$ should be an objet. The $destination$ should be a bond. $switch$ instruction is for the $source$ object to leave the current bond and become a member of $destination$ bond.

4.7.9 *join*

$switch(source, destination)$

The $source$ should be an objet. The $destination$ should be a bond. $join$ instruction is for the $source$ object to become a member of $destination$ bond.

4.7.10 change

$change(objecthandle, changelist)$

$change$ instruction is used to change the value of an object. The new values of attributes are specified in the $changelist$, which ia a list of assignment operations and can be specified as:

$[attribute1 = value1, attribute2 = value2, ...]$.

4.7.11 unlock

$unlock(entity1, entity2, ...)$

$unlock$ instruction is used to unlock the locked entities, which can be objects or bonds.

4.7.12 checkbondintegrity

$checkbondintegrity(bondhandle)$

$checkbondintegrity$ instruction is used to check whether the given bond satisfies the given constraints in the bond definition.

4.7.13 wait

$wait([type == messagetype, time = waitingtime])$

$wait$ instruction is used to synchronize instruction execution sequences, which will be explained in detail in Section 4.11.5. Before receiving the desired message specified by $messagetype$ or the expiration of the $waitingtime$, the transition code block execution will pause. If the desired message is received before the expiration time, the transition code block will continue to execute. If no message is received before the expiration of the timer, the transition code block will abort the execution.

4.7.14 send

$send([type = messagetype])$

$send$ instruction is used to send out the given type of messages to synchronize execution of transition code block executions.

4.7.15 jump

$jump$ instruction is used to implement the switch structure in a program.

$jump(condition, location)$

For $jump$, if the condition is true, the execution of the transition code block will start from the $location$, which is a label. If no condition is specified, the transition code block will unconditionally continue the execution from the $location$.

4.7.16 nondeterministicjump

$nondeterministicjump(L1, L2, ...)$

$nondeterministicjump$ instruction nondeterministically starts the next instruction from one of the location specified in the arguments as `L1`, `L2`, ...

4.8 Code Translation

In this section, we discuss how to construct the compiler that can translate a specification written in NetSpec into instruction sequences programmed using the given instruction set. The sequence of the following subsection is in the order of the block sequence in the right part of Figure 4.3.

To make the description of the code translator concrete, we describe a typical example specification in NetSpec, which captures its key features. The essential functions of NetSpec include data type definition, bond definition, and transitions as depicted in the left of Figure 4.3. In the following, we give the example specification.

```
Bond bondType1 (obj1Type Set os1, obj2Type Set os2, obj3Type Set
os3) {
    BondSelect From Where #(os1)==1 && #(os2)>=1 && #(os3)>=1
}

Transition Template (obj1Type Set obj1Set, obj2Type Set obj2Set,
obj3Type Set obj3Set, bondType1 bond1)

{

Select obj1 From obj1Set Where obj1.attr1==value1;
Select obj2 From obj2Set Where obj2.attr2=value2;

Select obj3 From obj3Set Where obj3.attr3=value3;

obj1 join bond1;

obj2 join bond1;

obj3 join bond1;

}
```

Figure 4.4 shows the relation between the program written in NetSpec and the translated instruction sequence program. It also shows the inserted instructions, such as lock, unlock, and bond integrity check. Those instructions are inserted to ensure the atomism and correctness of the instruction executions.

4.8.1 Decide The Transition Starter

The transition starter is the first instruction issuer, which starts the execution of one of the transition code block in the instruction sequence program. It is decided from the involved objects or bond leaders. The entity selection part of each transition in NetSpec includes a sequence of `Select` statements to identify the involved entities, which can be free objects or bond leaders. Given

```
Select obj1 From obj1Set where obj1.attr1==value1;
```

the translator will generate codes

```
test(this, [type==obj1Type && attr1=value1]);

obj1=findlocknode([type==obj1Type && attr1=value1]);
```

When executing this, if the `test` instruction is successful, the object that executes this instruction will start the instruction sequence program. This also means the `findlocknode` instruction will return the handle to itself, which will be ignored. Other `findlocknode` instructions will be executed to identify and lock the involved objects.

4.8.2 Find and Lock Involved Objects

After deciding the starter of each transition code block, we should identify the involved objects and lock them to ensure correctness. This can be inferred from two parts, the transition header and the `Select` statements.

The transition header contains the parameter list of a transition in NetSpec. Besides the object sets, it can specify the involved bonds. When the compiler encounters the bonds types, it will use the $findlockbond$ instruction to locate the involved bonds. The object sets are used to indicate the involved object types, which can be inferred from following `Select` operations. Thus the translator will only deal with the bond parameters in the header.

For the transition header as

```
Transition Template (obj1Set, obj2Set, obj3Set, bond1)
```

the compiler will encode as

```
bond1=findlockbond([type=bondType1])
```

where `bondType1` is inferred through the definition of `bond1`, which is also a part of the specification.

In NetSpec, `Select` statements are used to identify the involved objects. In the instruction set, $findlocknode$ instruction is used to perform the essential functions of `Select` statements. The constraints of `Select` statements, which is defined in the `'Where' <expr>`, can be specified correspondingly in the constraint parts of $findlocknode$ instruction. `<expr>` is essentially a boolean expression.

For

```
Select obj1 From os1 Where obj1.attr1==value1,
```

we can code using instruction

```
obj1=findlocknode([type==obj1Type && attr1==value1])
```

If the `Select` statement is to choose an object from a bond, we should first use `findlockbond` instruction to find and lock the bond. Then we should use the `findinternalnode` instruction to choose the object in the bond. Before we use `findinternode` instruction to reference internal objects in a bond, we lock the bond to ensure correctness.

4.8.3 Unlock Entities

After the execution of transition code block, we should unlock the involved objects. This is done by inserting the $unlock$ instructions.

In the Figure 4.4 we have codes

```
unlock(obj1, obj2, obj3, bond1)
```

4.8.4 Execution of Bond Operation Instructions

Operations on bond involve the creation of bonds and bond membership adjustment. When there is no object in a bond, the bond does not exist. Thus we do not have the explicit instructions to delete bonds.

The $bond$ instruction is used to create new bond to bond entities together, which can be used to implement the entity grouping functions.

```
new Bond bond2Type bond2(obj1, obj2, obj3)
```

can be encoded as

```
bond(obj1, obj2, obj3, [type==bondType2])
```

The constraints on bonds can be inferred from the bond definition.

`join`, `leave`, and `switch` instructions are used to adjust the bond memberships. They perform the same functionalities as those in NetSpec. They can implement directly their counter parts in NetSpec.

For example,

```
p join bond1
```

can be implemented directly as

```
join(p, bond1)
```

where `p` and `bond1` are handles in the instruction sequence program.

To utilize the nature of network systems, we have local parallelism in NetSpec, which is specified with the keyword `doparallel`. For example, the following code specifies that `Statement1` and `Statement2` can be executed in parallel, which also means that the sequence of the execution does not affect the final result.

```
doparallel{
```

```
Statement1;

Statement2;

}
```

Thus, the translation of the parallel statements involves the non-deterministic jump and a permutation of all possible execution sequences of the translated codes to be executed in parallel. The above code can be translated into,

```
nondeterministicjump(L1, L2);

L1: TransStatement1;

TransStatement2;

jump(end);

L2: TransStatement2;

TrasStatement1;

jump(end);

end:
```

There are two possible combinations of the two statements' translated codes, `TransStatement1` and `TransStatement2`. We label them with label `L1` and `L2`. The `nondeterministicjump` instruction makes a non-deterministic jump to one of the label.

4.8.5 Check Bond Integrity

When a transition code block's execution is finished, all involved bonds will be checked for integrity by inserting $checkbondintegrity$ instructions. The transition leader knows the bond leader of each bond. It will inform the leader of each bond to execute this instruction. The constraints of a bond can be inferred from its definition.

For example,

```
checkbondintegrity(bond2)
```

will check the integrity of `bond2`.

To implement this, each bond leader needs to maintain a data structure, called an object table, which contains detailed attributes information of each objects in its bond. By reading the constraints given in the specification, which are a sequence of `BondSelect` and `BondSelectEach` statements, and checking the object table, the bond leader can check the constraints sequentially.

The translator can infer that `bond2` is of type `bond2Type`. Looking at the definition, the constraint is

```
#(os1)==1 && #(os2)>=1 && #(os3)>=1
```

Then the bond leader can check whether the above boolean expression is satisfied or not by looking at the object table. The # operator can also be implemented by checking the number of objects of each type in the object table.

4.9 Translated Transition Code Blocks For Helicopter Rescue System

```
program T1 {

test(this, [type==People]);

test(this, [type==People_Helicopter]);

b=findlockbond([type==People_Helicopter]]);

p=findlocknode([type==People]);
```

```
h=findinternalnode([type==Helicopter], b);

jump([NOT(p.healthy==false && numnode(b.ps)<h.capacity)], end);

join (p, b);

end: checkbondintegrity(b);

unlock(p,b);

 };
```

Figure 4.5 depicts the relation between the Transition T1 in NetSpec and its translated code.

`T1` states that when an unbonded sick people finds that there is a `People_Helicopter` bond, the sick people will join the bond. First the sick people will start the execution with code:

```
test(this, [type==People]);
```

where $this$ is a special entity handle, which points to the entity that issues the instruction, the instruction issuer.

Then we find a `People_Helicopter` bond using

```
b=findlockbond([type==People_Helicopter]);
```

We use the $jump$ instruction to test the `if` statement's condition. `checkbondintegrity` should be performed before the `unlock` instruction, because after unlocking the entities, they can be involved in other operations.

```
program T2

{

test(this, [type==People]);
test(this, [type=Helicopter]);

h=findlocknode([type==Helicopter]);

p=findlocknode([type==People]);

jump([NOT(p.heathy==false)], end);

ph=bond(p, h, [type==People_Helicopter]);

end: checkbondintegrity(ph);

unlock(p,h);

}
```

Figure 4.6 depicts the relation between the Transition T2 in NetSpec and its translated code.

`T2` specifies that when an unbonded sick people finds that there is an unbonded helicopter, the sick people and the helicopter can form the bond `People_Helicopter` and become bonded.

```
program T3

{

test(this, [type==People_Helicopter]);

test(this, [type==Hospital_People]);

ph=findlockbond([type==People_Helicopter]);
hp=findlockbond(type==Hospital_People);

p=findinternalnode(ph, [type==People]);

h=findinternalnode(hp, [type==Hospital]);

jump([NOT(numnode(hp.ps)<h.capacity)], end);

switch(p, hp);

end: checkbondintegrity(hp);

checkbondintegrity(ph);

unlock(hp, ph);

}
```

Figure 4.7 depicts the relation between the Transition T3 in NetSpec and its translated code.

`T3` specifies that when a helicopter carrying sick people finds that there is a hospital available, it will send the sick people to that hospital. It selects a `People` object using `findinternalnode(ph, [type=People])`. `p` switches to `hp` bond. If the helicopter is carrying more than one people and `T3` is further executed, more people will be sent to the hospital.

```
program T4

{

test(this, [type==People_Helicopter]);

test(this, [type==Hospital]);
ph=findlockbond([type==People_Helicopter]);

hospital=findlocknode([type==Hospital]);

p=findinternalnode(ph, [type==People]);

hp=bond(p, hospital, [type==People_Hospital]);

checkbondintegrity(ph);

checkbondintegrity(hp);

unlock(hospital, ph);

}
```

Figure 4.8 depicts the relation between the Transition T4 in NetSpec and its translated code.

`T4` specifies that when a helicopter carrying sick people finds that there is an unbonded hospital available, it will send the sick people to that hospital. It selects a `People` object using `findinternalnode(p, [type=People])`. After `p` leaves `People_Helicopter` bond, it is bonded together with the hospital.

```
program T5

{

test(this, [type==Hospital_People]);

hp=findlockbond(type==Hospital_People);

p=findinternalnode(hp, [type==People]);
jump([(p.healthy==true)],end);

leave(p);

end: checkbondintegrity(hp);

unlock(hp);

}
```

Figure 4.9 depicts the relation between the Transition T5 in NetSpec and its translated code.

Program T5 states that if a people in a hospital becomes healthy, the people will leave the hospital and become unbonded.

4.10 Execution of Transition Code Blocks

As shown in Figure 4.4, the translated instruction sequence program consists of transition code blocks, which have no dependency on each other and are the execution function units. In this section we describe how to execute the transition code blocks.

4.10.1 Computation Model

The computation model of the network virtual machine is the same as that of NetSpec. The transition scheduler's task is to choose the ready-to-execute transition code block and make sure that at each moment only one transition code block is executing in the system. This is based on non-deterministic state machines and assumes no maximal parallelism. If two or more transition code blocks are enabled simultaneously, a non-deterministic choice will occur and only one of them will be executed. This takes into consideration that the network virtual machine will eventually be implemented over a network, the cost of implementing the maximal parallelism (which requires a global lock) is unlikely realistic, and is almost impossible in real unreliable networks. Hence, transition code blocks are fired asynchronously. To implement the non-deterministic scheduling of transition code blocks' execution, there is only one scheduler for one application in the system.

4.10.2 Deciding Starter of Each Transition Code Block Execution

Each object in the system has a copy of the compiled instruction sequence program code. The first section of each transition code block consists of a sequence `test` instructions. Through the execution of these `test` instructions, each node can decide whether it can start the execution of the instruction sequence program, or whether it can become the starter. If one of the `test` instruction's constraints are satisfied, the object will become the starter and try to start the execution of the instruction sequence program. All the objects periodically check whether it can become the starter and start to execute one of the transition code block.

4.10.3 Scheduling Transition Code Block Execution

If the transition code block's execution can be started, before this, the transition code block starter will contact the scheduler in the system. A transition scheduler is responsible for choosing the right transition code blocks to fire. In realism, more than one transition code block execution fire request can not arrive at the scheduler at the exact same time. There is a order of the request arrival. When there is no transition code block is executing in the system, the scheduler will accept the current fire request. All the following requests will be refused if there is one transition code block executing in the system.

When designing the transition code block scheduler, we should also address the fairness issue. In large systems, if one transition code block's starter's fire request can always arrive at the scheduler earlier than others, its fire request has more chance to be accepted by the scheduler, which may cause unfairness. Adopting the mechanism of deferring before the request as that used in the CSMA [85], before a starter of a transition code block issues the transition fire request, it will defer an amount of time.

$$t_{defer} = \alpha \times \frac{succrate}{rtt} \tag{4.1}$$

In (4.1), t_{defer} is the amount of defer time. α is a factor. rtt is the round trip time for the entity to send and receive a probing data packet from the scheduler. rtt is measured periodically. The more frequently rtt is measured, the more accurate rtt is. The shorter the rtt is, the more the defer time is. To further ensure the fairness, we also take into account the past successful requests. The more request is successful, the longer the defer time is.

If S is the number of successful requests by an entity over a period of T, $succrate$ can be calculated as:

$$succrate = \frac{S}{T} \tag{4.2}$$

The smaller T is, the more accurate the $succrate$ is.

Because of the clock drift of local clocks, to make the measurement of rtt more accurate, it is necessary to synchronize local clocks on each entity. This also justifies the global clock in the system.

4.10.4 Execution of Instructions Inside A Transition Code Block

The body of each transition code block consists of a sequence of instructions. Each instruction has a instruction issuer controlling the execution of the instruction.

To obtain globally correct behavior from applications, it is necessary to synchronize the order in which actions are taken. Thus some instructions should be executed in a sequential manner, which means, an instruction cannot start until the proceeding instruction is completed. This is important to ensure the correctness and consistency of the execution of the program in a completely asynchronous system, although is more complex compared to achieving the sequentialism on a single processor by using the program counter (PC).

There are two different approaches to achieving the sequential execution of instructions. One approach is to use a central server for each transition code block execution. Each transition code block execution has a transition leader, which triggers the execution of the instructions. Each instruction has an issuer to start the execution of the instruction. The transition leader serves as the central server for the central server approach. It synchronizes the execution of instructions by collecting the execution results from involved objects and instructing the next instruction issuer to start. Each instruction issuer is responsible for informing the transition leader of the completion of the instruction execution.

The other approach is to distribute the control functionalities among involved entities in the execution. To take advantage of the fact that each entity in the network computing system can have a complete copy of the network computing program and avoid the bottleneck created by the central server, we adopt the distributed approach. For the distributed approach, in each transition code block's execution, there is a token passed among the involved instruction issuers. The transition leader is responsible for creating the token. The layout of the token is illustrated in Figure 4.10. The token contains the instructions and each instruction's issuer, which can be obtained after executing the $findlocknode$, $findlockbond$, and $findinternalnode$ instructions. In the token, the context of the current execution of the transition code block, such as instruction execution results, is also recorded for the next instruction issuer to check whether it can start the execution of the instruction and obtain the desired execution results of the instructions preceding it. When an instruction issuer gets the token, it will check whether the execution constraints are satisfied. The constraints can be checked by looking at the context information recorded in the token. If they are satisfied, it will start the instruction execution. After the execution of the instruction, the instruction issuer needs to pass the token and informs the following instruction's issuer to start the execution.

```
instruction1;
instruction2;
instruction3;
```

For example, in the above codes, we assume the issuers of each instruction is `i1, i2, i3`. After `i1` determines the completion of the execution of `instruction1`, it will pass the token to inform `i2`. In a similar manner, `i2` can inform `i3`, until the completion of the execution of the instruction sequence. After the execution of the transition code block, the transition leader reports to the scheduler about this so that the scheduler can continue to fire executable transition code blocks.

4.10.5 Local Parallelism And Non-deterministic Jump

In NetSpec, if the order of the execution of some statements does not affect the result, to fully utilize the computation capability of distributed devices, some statements in a transition can be executed in parallel manner. These statements are specified using `doparallel` keyword. Otherwise, by default, statements are executed sequentially.

In the translated instruction sequence program, we use $nondeterministic$ instruction and the combination of all possible execution sequences to implement this. The non-deterministic selection of executable transition code block is made in a pseudo-random manner. The instruction issuer of the $nondeterministic$

instruction chooses the next instruction randomly according to some criteria, such as time, location, or ID.

4.11 Implementation Issues of the Network Virtual Machine

In this section, we discuss the implementation issues of the network virtual machine. We first discuss initialization of objects. To execute an instruction, we have to locate involved entities. To maintain the bond, leader election mechanism is necessary to elect proper entity to maintain the bond information. To ensure the correctness of instruction execution, entities should be locked before the operation. For some applications, the network virtual machine provides concurrency mechanism for interactions among execution of transition code blocks, which are executions of instruction sequence programs in the system.

4.11.1 Initialization of Object Attributes

Since objects represent the physical entities in reality, the attribute values are initialized by the entities. The aggregate of all objets attribute values constitute the state of the system. One state is differentiated from the other by different object attribute values. A fundamental question tightly related to communication is how the entities are identified. There is a need for globally unique entity Ids, though entity identity is not crucial for the upperlayer instruction codes. Identifier schemes that provide for migration transparency are well-understood today, such as those deployed in IPv6 [86] and Mobile IPv4 [87].

4.11.2 Locating and Locking Entities

To implement the $findlocknode$ and $findlockbond$ instructions, we need to provide an efficient method to locate and lock entities that satisfy the given constraints.

To locate the entities, there are two approaches. One is a completely distributed approach. Each entity is responsible for finding out the desired entities through cooperations among them by broadcasting, or multicasting to a limited scope the lookup request to other entities in the network. This approach is close to multicast DNS name resolution [88]. The other approach is a hierarchical approach, which deploys some servers in the network to maintain entities information, which is similar to the naming and discovery approach, such as Jini [56], Salutation [89], and Service Location Protocol [90]. All entities need to register their properties with one of the servers. An entity can submit the lookup request to the server, which returns the handle of the desired entity by looking up its local lookup table, or collaborating with other servers in the system. To make the network virtual machine scalable and avoid the bottleneck of servers, we adopt the distributed approach.

To ensure the correctness of instruction operations, a transition leader should lock the discovered entities and unlock them at the end of the transition code block execution. Only operations among the entities in the same locked group can interact with each other.

For bond, the transition leader will lock with bond leaders first. Then bond leaders will then lock the local members in bonds. For unbonded objects, they will be locked individually.

When a transition leader, p, wants to find an entity that satisfies the given constraints, it will broadcast the *LookUp* message. Upon receiving the *LookUp* message from p, if an object, q, finds that it satisfies the constraints and is not locked, it will reply with *ACKLookUp* to p and become locked. If q is locked, the request will be placed in a waiting queue so that when q is unlocked, the reqeusts on the waiting queue of that item are processed first in first out (FIFO) order. p may receive several objects' *ACKLookUp* messages. It will randomly choose one and send *Accept* message to that object. After accept one object's *ACKLookUp*, for all the other received *ACKLookUp*, p will reply with *Refuse* message. If q receives the *Refuse* message, q will become unlocked and can accept new lookup requests.

If the desired object cannot be obtained after several trials or within a given period of time, the execution of the instruction sequence will be aborted. The transition leader has the complete information about the previously discovered and locked entities. Because there has not been any operation on the locked objects, unlocking them will abort the execution.

At the end of the execution of the transition code block, all the locked entities will be unlocked by the $unlock$ instruction.

4.11.3 Leader Election

In each bond, an entity is elected through distributed leader election algorithm as the bond leader to maintain the information of that bond control the bond operations. Transition leaders are usually elected among involved bond leaders. Existing clustering approaches, such as [25, 10, 11, 15, 14, 30, 17], can be adopted to elect suitable clusterheads as leaders to meet different application requirements. Related work of clustering is described in Section 2.2

For the $bond$ instruction, if the two involved entities are both bonds, one of the bond leader is elected as the operation leader to issue the bond operation. Usually, leaders have the complete information of the group through message exchanges. Thus they should be more capable than other members.

4.11.4 Maintenance of Bonds

Although we are talking about reliable network communications. While in reality, we may encounter unreliable communications. Entities exchange messages to reflect changes in topology. Each bond leader broadcasts BOND messages periodically. Each member responds with HELLO messages. The bond leader also needs to check whether the bond constraints ar satisfied or not. If the constraints are not satisfied, the bond will break up. If a member does not receive a BOND message from its leader after a given period, it assumes that its leader cannot be contacted. If a bond leader does not receive a HELLO message from a member after a given period of time, it assumes that the member cannot be contacted and will remove it from the member list.

4.11.5 Synchronization among Transition Code Block Executions

During the execution of a transition code block, say s_1, it may need the input of another transition code block execution, say s_2. To synchronize the execution of the two transition code blocks, we use the instructions, $wait$ and $send$. The scenario is depicted in Figure 4.11. During the execution of s_1, it waits for a message of type `msg1` with instruction,

```
wait([type==msg1, time=100s]);
```

s_2 at some time sends out the message of type `msg1` by executing

```
send([type=msg1]);
```

Then s_1 can continue to execute. To prevent an endless waiting, if the desired messages are not received for a given period of time, specified by the waiting time parameter, s_1 will stop execution. The sent message can flood the whole network area to ensure the receipt of it. Or to reduce the consumption of network bandwidth, it can be broadcasted to a limited scope.

The Object Management Group (OMG) event services specification [91] defines the Event Service in terms of suppliers and consumers, which also provides the synchronization mechanism for distributed entities. Suppliers are objects that produce event data and provide them via the event service, consumers process the event data provides by the event service. The event channels decouple the suppliers and consumers. For out completely distributed system, it is not feasible to deploy the event channels.

4.12 Instruction Implementation

Based on the issues we address in Section 4.11, we describe the implementations of each individual instruction in this section.

4.12.1 *test*

$test$ instruction tests whether the entity, which can be an object or a bond, satisfies the given condition. This is the entry point of all instruction sequence programs. Thus each object can actively execute this instruction to check whether it satisfy the give constraints. Usually, an instruction sequence program has several test instructions. If one of the test instruction's constraint is true, the transition code block will be executed.

4.12.2 *findlocknode, findlockbond*

The $findlocknode$ instruction is used to find the unbonded destination that satisfies the given conditions. The $findlockbond$ instruction is used to find the bond that satisfy the given conditions. The instruction issuer of the $findlocknode$ or $findlockbond$ instruction use the mechanism described in Section 4.11.2 to find and lock the designated objects.

4.12.3 *findinternalnode*

The $findinternalnode$ instruction is used to find a member that satisfies the given conditions in the bond, $bond1$. Each bond leader maintains the complete information of its bond. The bond leader can locate the member through the use of its object table, which is described in Section 4.8.5.

4.12.4 *numnode*

$numnode$ instruction is to obtain the number objects of given type. It is equivalent to the # operator in NetSpec. This information can also be obtained by looking at the object table. The bond leader can count the number of specified type of objects.

4.12.5 *bond*

The $bond$ instruction is used to bond entities together. The entities can be nodes or bonds. If the entities are objects, the objects will form a new bond. If one of the entity is a bond, the other entities will join the bond.

4.12.6 leave, switch, join

These instructions should be executed after locking the involved objects. Each object maintains information about whether it is a bond leader or it is a bond member. If an object is a bond member, it should maintain its bond leader information. A bond leader should maintain the object table data structure, which keeps record of its member information. To change the membership information involves the update of the object table information on the bond leader side and the membership information on the bond member side.

4.12.7 change

$change$ instruction is used to change the value of an object. By looking at the $changelist$, the object can change the attribute values accordingly.

4.12.8 checkbondintegrity

$checkbondintegrity$ instruction is used to check whether the given bond satisfies the given constraints in the bond definition. The bond leader is the place that performs the integrity check. The check is performed by comparing the information in the object table and the bond definition, which can be performed as described in Section 4.8.5.

4.12.9 wait and send

$wait$ instruction is used to synchronize instruction execution sequences. During the execution of the transition code block, when encountered $wait$ instruction, the instruction issuer just stop at the point and wait for the desired message. If the desired message is received before the expiration time, the transition code block will continue to execute. If no message is received before the expiration of the timer, the transition code block will abort the execution.

$send$ instruction is used to send out the given type of messages to synchronize execution of transition code block executions. This can be implemented using network message communication mechanism.

4.12.10 jump

$jump$ instruction is used to implement the switch structure in a program.

For $jump$, if the condition is true, the current instruction issuer will pass the token to the instruction issuer of the destination instruction. Execution of the transition code block will start from the destination.

4.12.11 nondeterministicjump

$nondeterministicjump$ instruction is used to nondeterministically start the next instruction from one of the location specified in the arguments as `L1`, `L2`, ... The instruction issuer of $nondeterministicjum$ chooses the next instruction in a pseudo random manner based on some criteria, such as time, location, and ID. Then the instruction issuer passes the token to the chosen instruction's issuer.

4.13 Performance Evaluation Through Simulation

We used NS-2 [43] to evaluate the implementation of the helicopter rescue system. NS-2 provides the simulation environment of the wireless networks, which are reliable in nature, though we assume the reliable communications. In the simulation scenario, there are three types of objects in the network: patient, helicopter, and hospital. All nodes are stationary in the network. When a patient is sick, it will broadcast *LookUp* messages to find an available helicopter, which is in correspondence to the resource finding and locking phase of the instruction programs. We define that each helicopter can hold at most four patients. If any helicopter is available, it will reply so that the patient can know the information about the available helicopters in range. Then the patient will join the helicopter. Between the point that the patient issuing the join request and its join request is accepted or refused, the patient will refuse any further operation, which is locked. If a helicopter gets a patient, it will broadcast *LookUp* messages to find a hospital. If there is any hospital available, it will respond. Then the helicopter will transfer the patients on it to the hospital. Periodically, the hospital will determine which patient can be healthy and get the patient out of the hospital. In the simulation, a patient will become unhealthy every 10 seconds, with random clock drift to make it more realistic.

The simulation parameters are listed in Table 4.1 unless mentioned otherwise. We used AODV [45] as the underlying routing algorithm, which makes use of advantages from both distance-vector and on-demand. The final results are the the average of 10 runs. We simulate two network cases, with the communication range of 400 m and 800 m respectively. The 800 m network is an extreme case because the communication range is equal to the network diameter.

The simulation results are listed in Table 4.2.

The lookup failures and join failures are detected through timeout mechanism. After sending out the

Table 4.1: Simulation Parameters for Helicopter Rescue System

Parameter	Value
Number of nodes	50
Number of patients	35
Number of helicopter	9
Number of hospitals	6
Network size	670 m $\times$ 670 m
Communication range	400 m, 800m
Patient sickness frequency	Once per 10 seconds with clock drift
Maximum bandwidth	10M bps
Simulation time	200 seconds

Table 4.2: Simulation Results for Helicopter Rescue System

Communication range	400m	800m
Number of lookups per object	30.736	20.788
Lookup failures per object	0.068	1.109
Percentage of lookup failures per object	0.122	5.628
Number of joins per object	30.456	19.279
Join failures per object	0.061	0.002
Percentage of join failures per object	0.367	3.78e-3

lookup request or the join request, if there is no response, we take it as a failure. For the lookup request, if there is no reply after the given period of time, the patient or the helicopter will send out the lookup request again, until it finds one

Although we have the assumption that the underlay networks should be reliable, in the unreliable case, the number of lookup failures and the number of join failures can be neglectable. For the 400 m network, it is less congested compared to the 800 m network. Thus the number of lookup failures is fewer than the 800 m network. Because of the more successful lookup operations, the 400 m network also has more join requests than the 800 m network. Because data communications are more reliable for 400 m network, patients can get out of hospital more frequently, which makes the 400 m network occur more lookups than the 800 m case.

4.14 Summary

This chapter introduces a virtual machine for grouping over networks. The network virtual machine is designed to solve the problems brought by network systems, such as device heterogeneity, and the poor portability of codes. It separates the upperlayer program from the under layer network services. As long as the virtual machine supports the instruction set, the program encoded using the instruction set can run on different networks, which eliminates the redundant effort in developing different versions of the same application for different networks. Then we propose how to construct the compiler that can translate the specification codes written in NetSpec into under layer instruction sequence programs. We also describe how to implement the non-deterministic scheduler to ensure the fairness when firing transition code blocks. To work on real networks, the network virtual machine deals with synchronization, group communication control, and concurrency control. To make the description of the network virtual machine concrete, we use a concrete illustrated example, the helicopter rescue system, to show how to translate the transitions encoded in NetSpece into instruction sequence programs. To evaluate the performance of the proposed solution in real networks, we used NS-2 to evaluate the performance of an implementation of the helicopter rescue system. The results demonstrate that the proposed solution works well even in unreliable wireless networks.

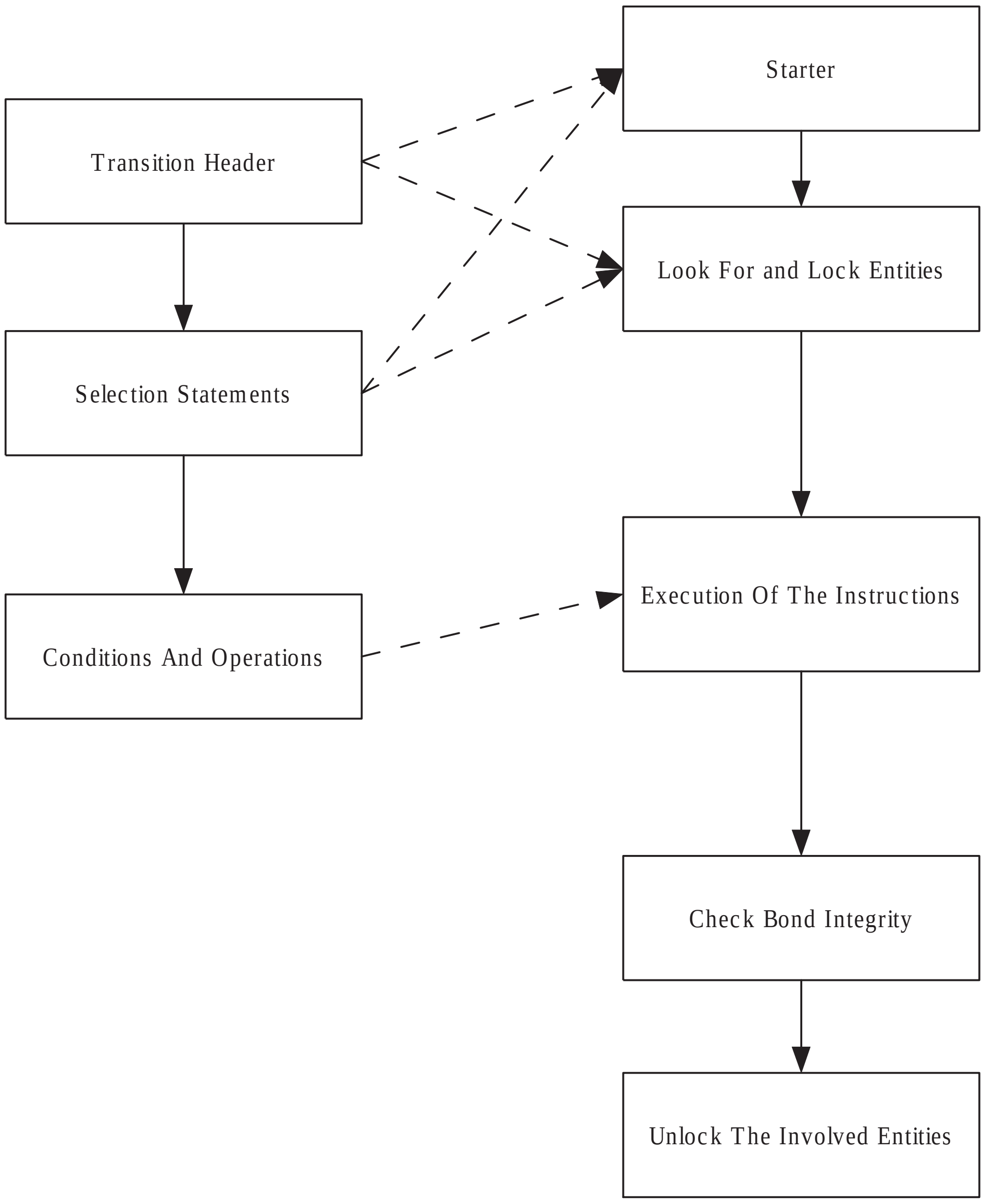
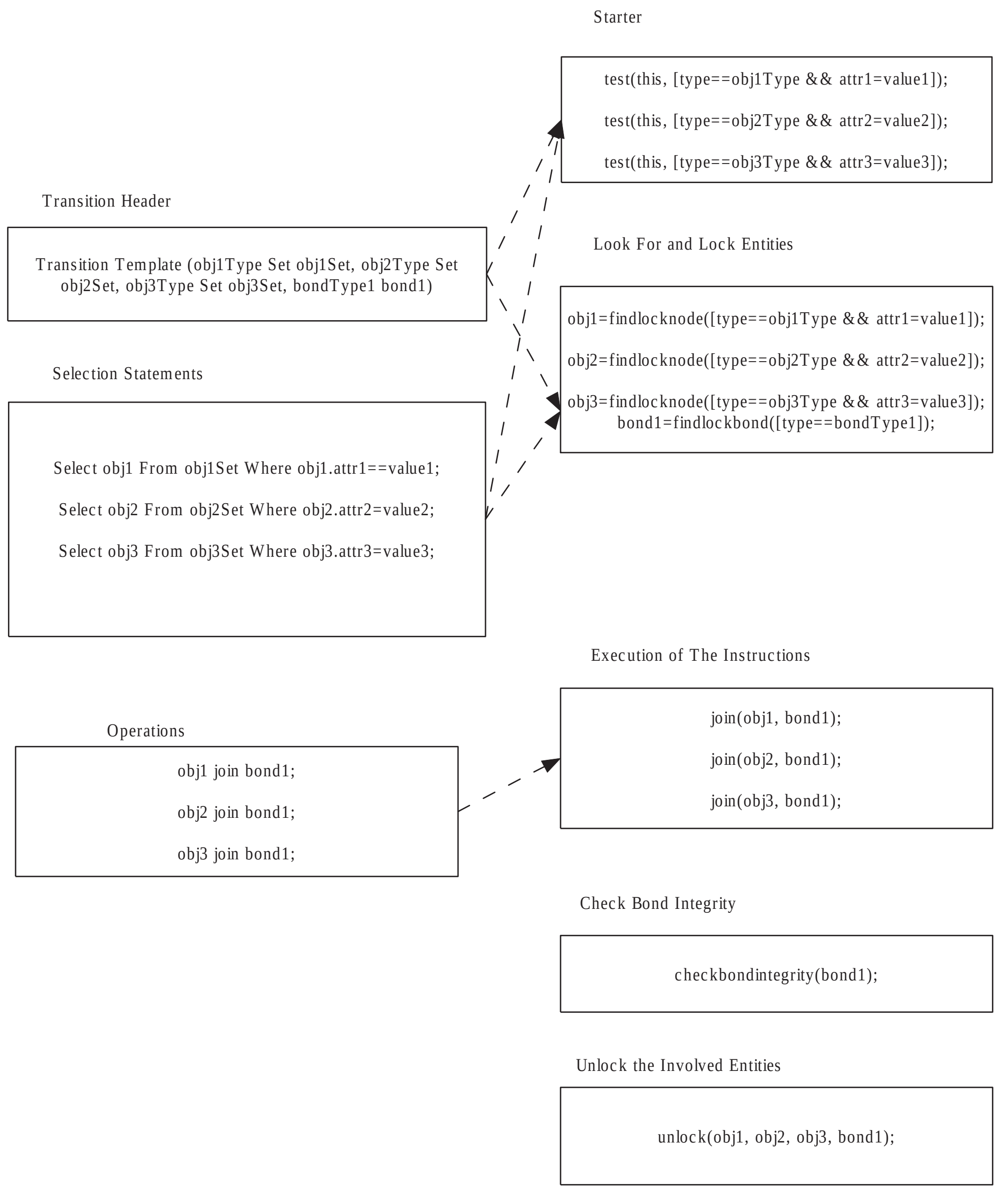

Transition Header
Selection Statements
Conditions And Operations
Starter
Look For and Lock Entities
Execution Of The Instructions
Check Bond Integrity
Unlock The Involved Entities

Figure 4.3: Transition Specification Translation



Starter
test(this, [type==obj1Type && attr1=value1]);
test(this, [type==obj2Type && attr2=value2]);
test(this, [type==obj3Type && attr3=value3]);
Transition Header
Transition Template (obj1Type Set obj1Set, obj2Type Set obj2Set, obj3Type Set obj3Set, bondType1 bond1)
Look For and Lock Entities
obj1=findlocknode([type==obj1Type && attr1=value1]);
obj2=findlocknode([type==obj2Type && attr2=value2]);
obj3=findlocknode([type==obj3Type && attr3=value3]);
bond1=findlockbond([type==bondType1]);
Selection Statements
Select obj1 From obj1Set Where obj1.attr1==value1;
Select obj2 From obj2Set Where obj2.attr2=value2;
Select obj3 From obj3Set Where obj3.attr3=value3;
Execution of The Instructions
join(obj1, bond1);
join(obj2, bond1);
join(obj3, bond1);
Operations
obj1 join bond1;
obj2 join bond1;
obj3 join bond1;
Check Bond Integrity
checkbondintegrity(bond1);
Unlock the Involved Entities
unlock(obj1, obj2, obj3, bond1);

Figure 4.4: Example Transition Specification Translation

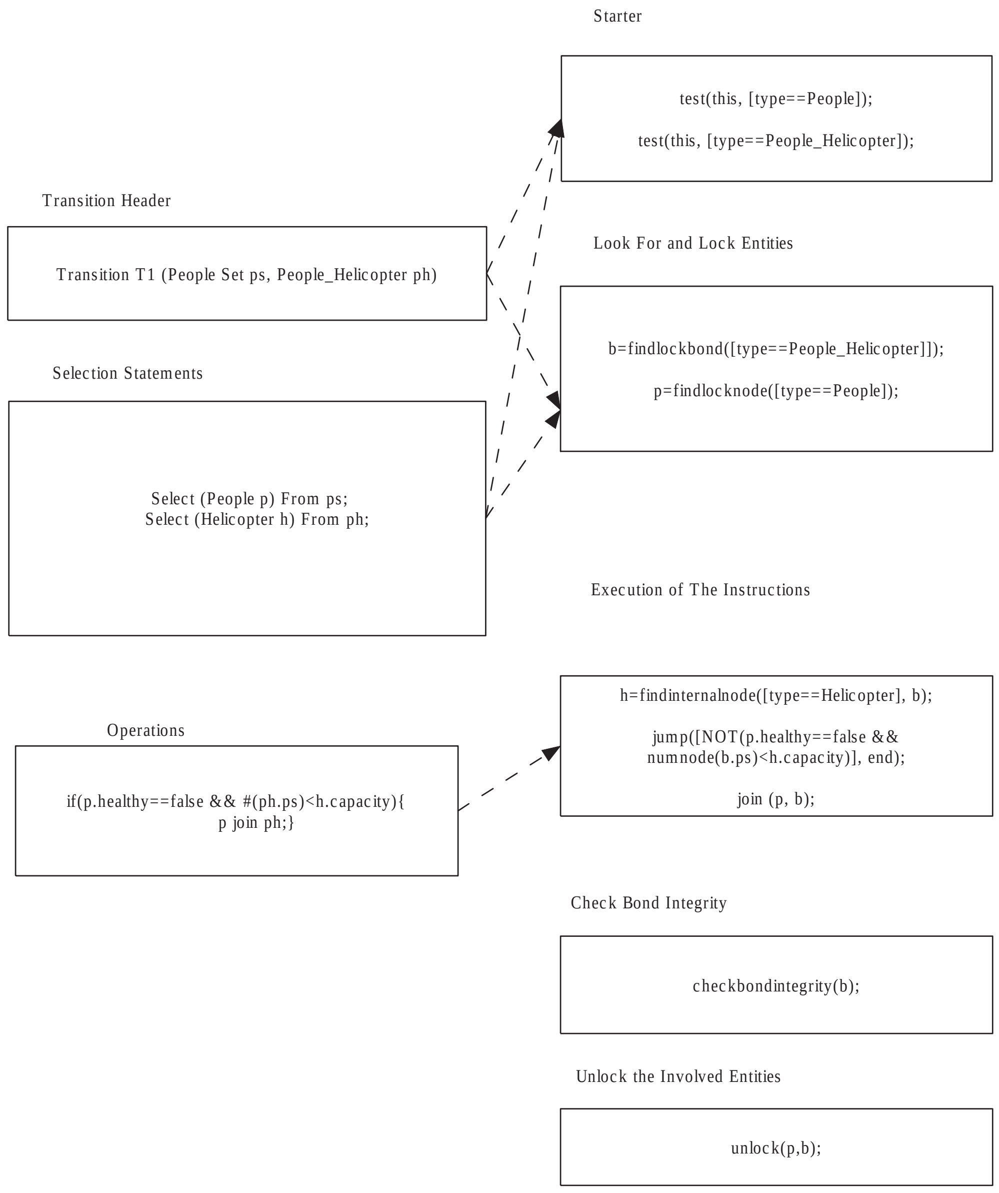

Starter
test(this, [type==People]);
test(this, [type==People_Helicopter]);
Transition Header
Transition T1 (People Set ps, People_Helicopter ph)
Look For and Lock Entities
b=findlockbond([type==People_Helicopter]]);
p=findlocknode([type==People]);
Selection Statements
Select (People p) From ps;
Select (Helicopter h) From ph;
Execution of The Instructions
h=findinternalnode([type==Helicopter], b);
jump([NOT(p.healthy==false && numnode(b.ps)<h.capacity)], end);
join (p, b);
Operations
if(p.healthy==false && #(ph.ps)<h.capacity){
p join ph;}
Check Bond Integrity
checkbondintegrity(b);
Unlock the Involved Entities
unlock(p,b);

Figure 4.5: Transition T1 in NetSpec And Its Translated Code

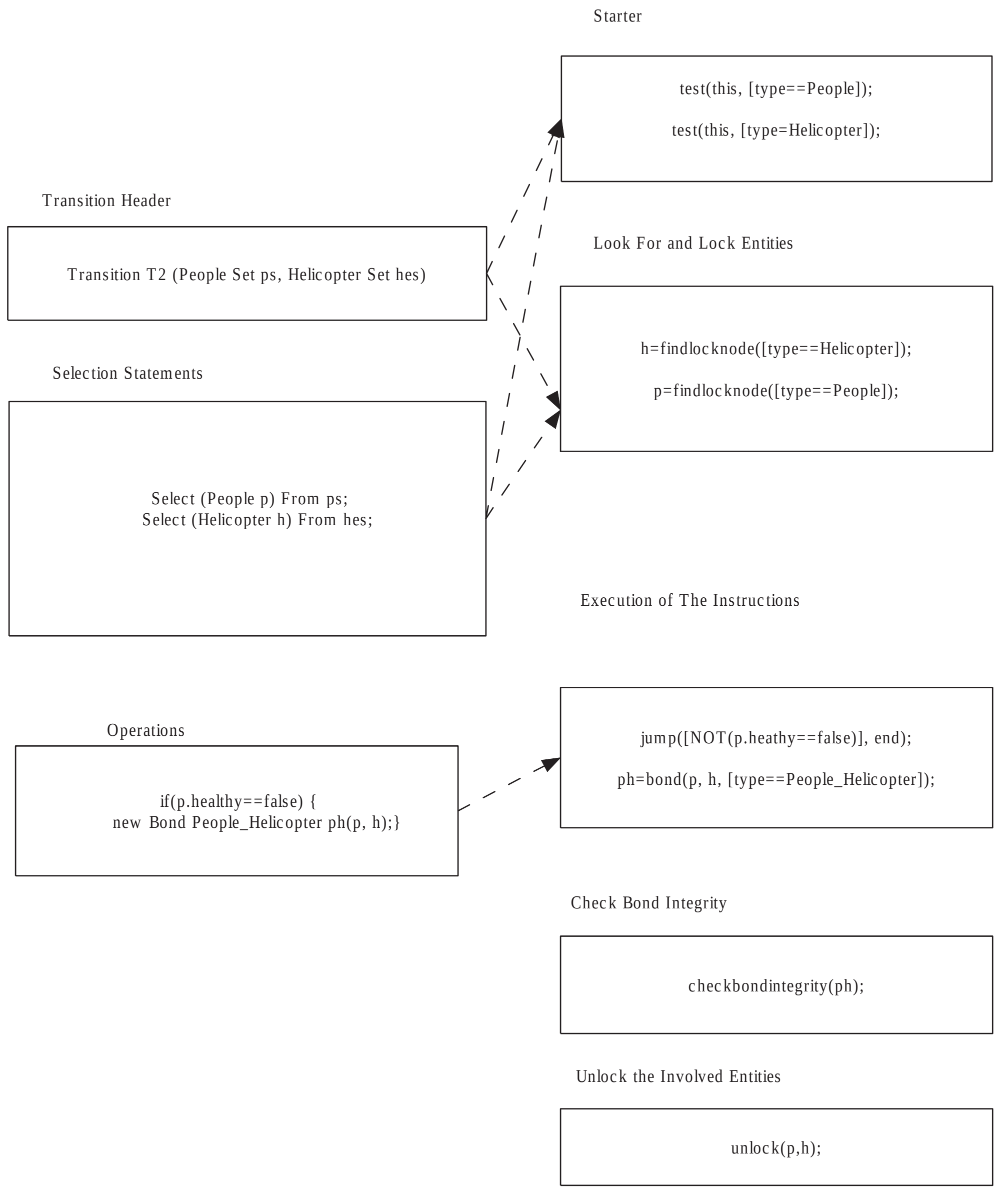

Starter
test(this, [type==People]);
test(this, [type=Helicopter]);
Transition Header
Transition T2 (People Set ps, Helicopter Set hes)
Look For and Lock Entities
h=findlocknode([type==Helicopter]);
p=findlocknode([type==People]);
Selection Statements
Select (People p) From ps;
Select (Helicopter h) From hes;
Execution of The Instructions
Operations
jump([NOT(p.heathy==false)], end);
ph=bond(p, h, [type==People_Helicopter]);
if(p.healthy==false) {
new Bond People_Helicopter ph(p, h);}
Check Bond Integrity
checkbondintegrity(ph);
Unlock the Involved Entities
unlock(p,h);

Figure 4.6: Transition T2 in NetSpec And Its Translated Code

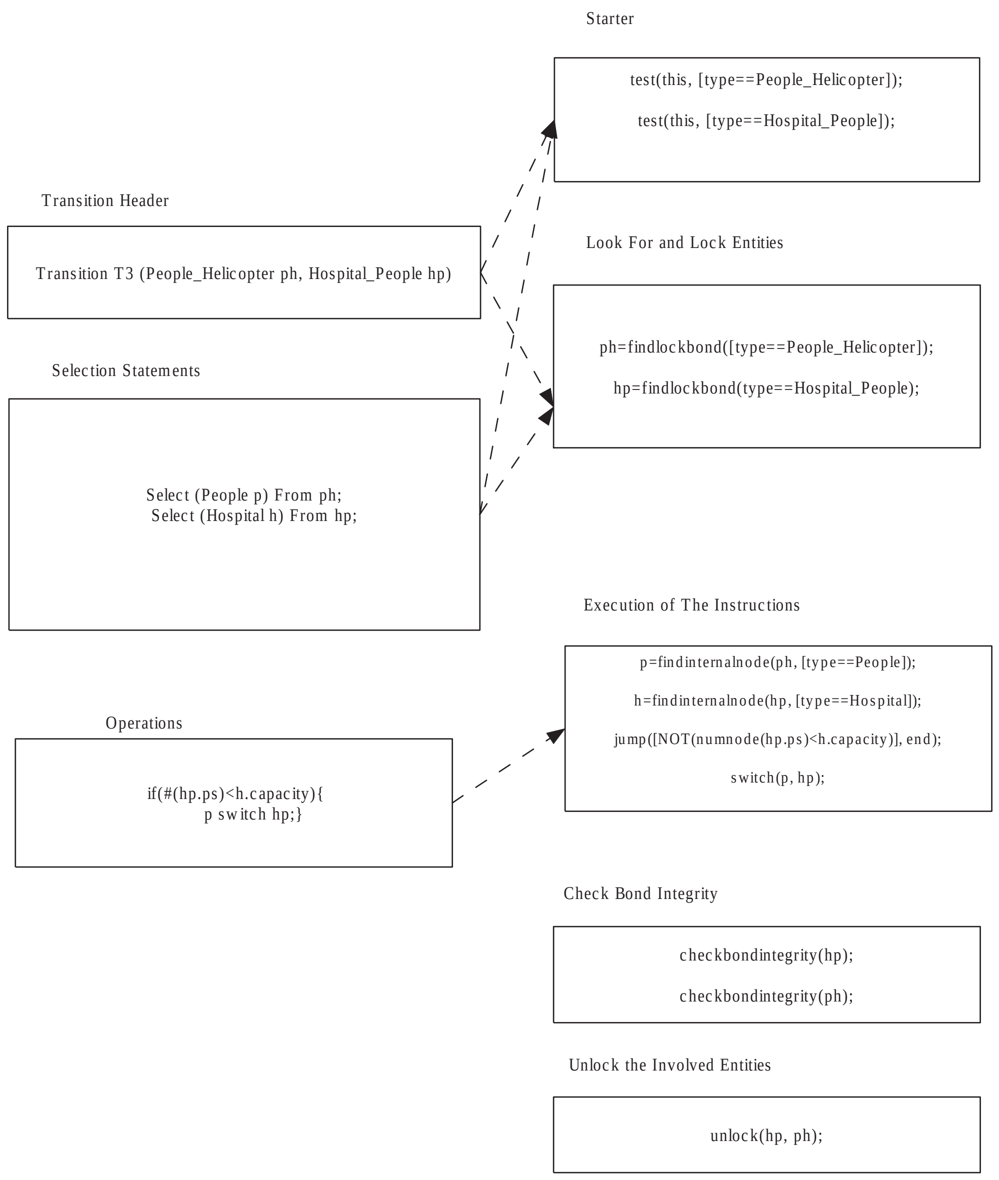

Starter
test(this, [type==People_Helicopter]);
test(this, [type==Hospital_People]);
Transition Header
Transition T3 (People_Helicopter ph, Hospital_People hp)
Look For and Lock Entities
ph=findlockbond([type==People_Helicopter]);
hp=findlockbond(type==Hospital_People);
Selection Statements
Select (People p) From ph;
Select (Hospital h) From hp;
Execution of The Instructions
p=findinternalnode(ph, [type==People]);
h=findinternalnode(hp, [type==Hospital]);
jump([NOT(numnode(hp.ps)<h.capacity)], end);
switch(p, hp);
Operations
if(#(hp.ps)<h.capacity){
p switch hp;}
Check Bond Integrity
checkbondintegrity(hp);
checkbondintegrity(ph);
Unlock the Involved Entities
unlock(hp, ph);

Figure 4.7: Transition T3 in NetSpec And Its Translated Code

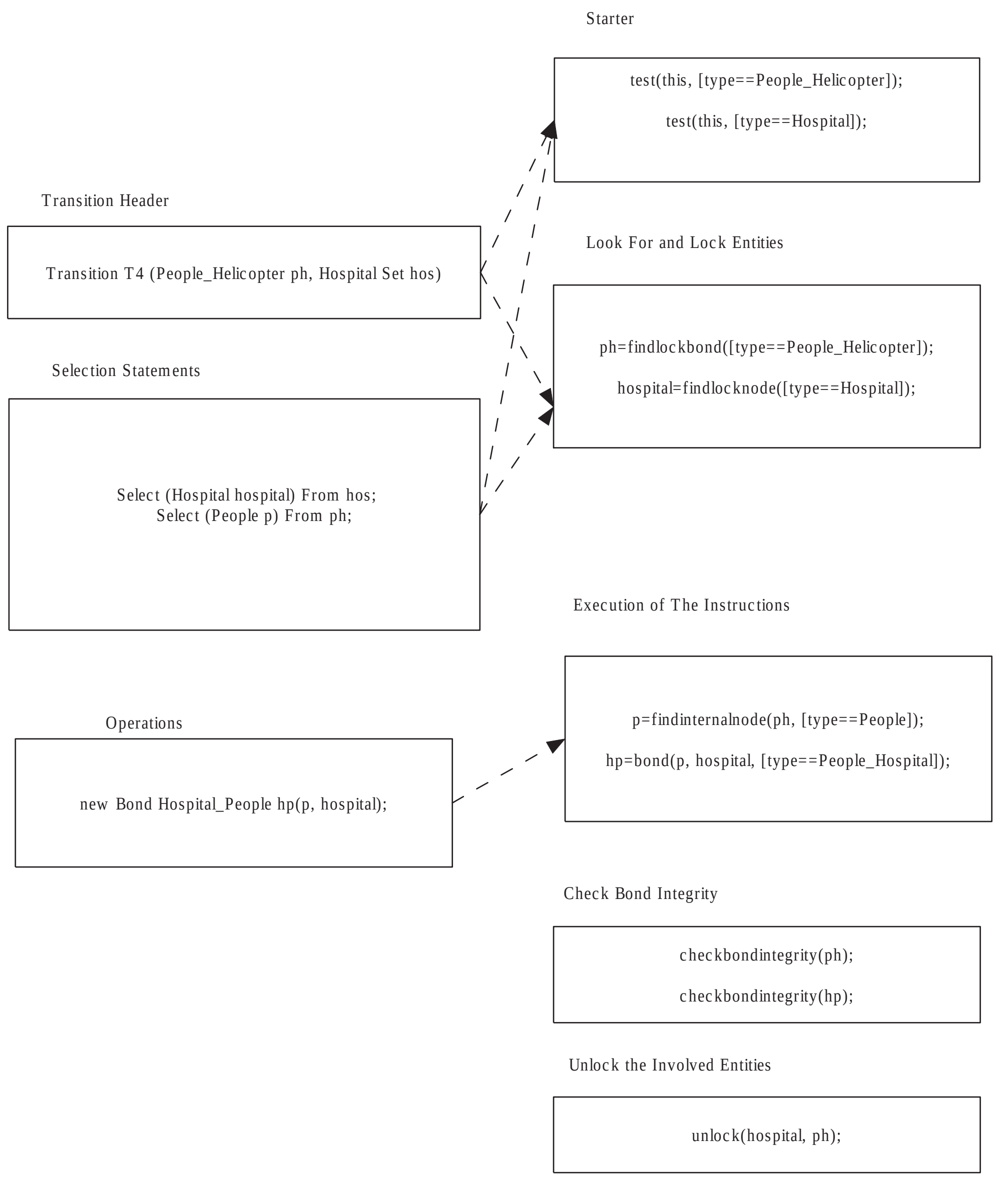

Starter
test(this, [type==People_Helicopter]);
test(this, [type==Hospital]);
Transition Header
Transition T4 (People_Helicopter ph, Hospital Set hos)
Look For and Lock Entities
ph=findlockbond([type==People_Helicopter]);
hospital=findlocknode([type==Hospital]);
Selection Statements
Select (Hospital hospital) From hos;
Select (People p) From ph;
Execution of The Instructions
p=findinternalnode(ph, [type==People]);
hp=bond(p, hospital, [type==People_Hospital]);
Operations
new Bond Hospital_People hp(p, hospital);
Check Bond Integrity
checkbondintegrity(ph);
checkbondintegrity(hp);
Unlock the Involved Entities
unlock(hospital, ph);

Figure 4.8: Transition T4 in NetSpec And Its Translated Code

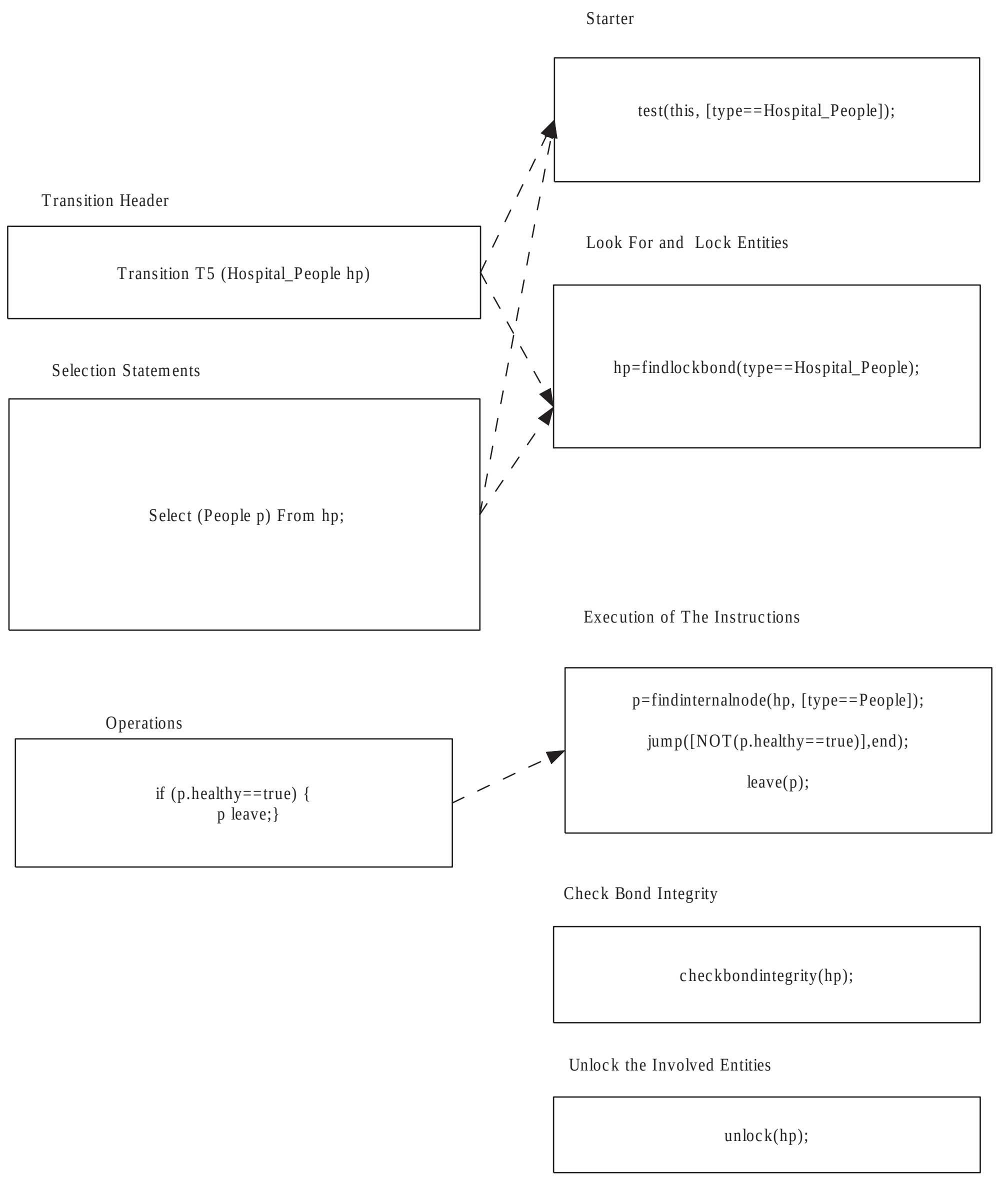

Starter
test(this, [type==Hospital_People]);
Transition Header
Transition T5 (Hospital_People hp)
Look For and Lock Entities
hp=findlockbond(type==Hospital_People);
Selection Statements
Select (People p) From hp;
Execution of The Instructions
p=findinternalnode(hp, [type==People]);
jump([NOT(p.healthy==true)],end);
leave(p);
Operations
if (p.healthy==true) {
p leave;}
Check Bond Integrity
checkbondintegrity(hp);
Unlock the Involved Entities
unlock(hp);

Figure 4.9: Transition T5 in NetSpec And Its Translated Code

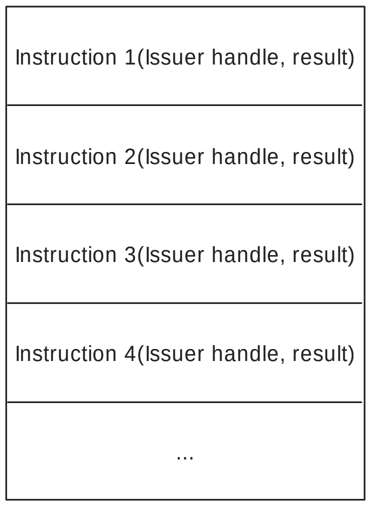

Instruction 1(Issuer handle, result)
Instruction 2(Issuer handle, result)
Instruction 3(Issuer handle, result)
Instruction 4(Issuer handle, result)
...

Figure 4.10: Token Layout

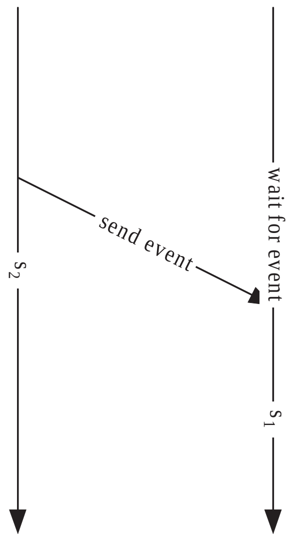


Figure 4.11: Synchronization of Transitions

CHAPTER 5

PROCESS OVER NETWORKS

5.1 Overview

In the previous chapters, we describe a specification language, NetSpec, to let users specify the desired functionalities of applications, and a virtual machine over networks, which supports an instruction set so that programs encoded using the instruction set can run on different networks provided the networks support the virtual machine. Essentially, NetSpec specifies how groups of network nodes evolve. Looking from the angle of each individual node, such an evolution can be characterize as a thread of atomic transitions, each of which is local; e.g., involving two network nodes. Inspired by this view, in this chapter, we propose a form of network processes, called network process graphs, and study, theoretically, their computability and realization on a physical network.

A network process is a executing program running over network objects. In a real network, a network process first grabs some network entities, and consume their resources. When some entities finish their computation tasks, the process would release the entities and then move on to others. In our model, we abstract this phenomena as: the process walks through objects, and objects change their states as the process passing by. We discuss only the single process model, in which only one process is running in the network.

Traditionally, automata theory addresses the computability problem by studying the form of languages accepted by an automaton. Similar to the idea of graph representation tools usually used by automata theory, we propose a class of process graphs to depict the behavior of a network process. Essentially, a process graph is an automaton and a word accepted by the automaton representing a process demonstrated by the application that is modeled by the graph.

This chapter[1] is organized as follows. We first present an example that is modeled using a process graph and, informally, interpret its semantics. Then, we formally define the syntax and semantics of the process graphs. We will also prove that the computing power of the process graphs is equivalent to VASS. Finally, we will discuss how to implement a process graph on a network.

[1]This chapter is part of a joint work with Linmin Yang and Prof. Zhe Dang, who both agree to include the part in this dissertation.

5.2 Related Work

Mobile agent concept is close to the concept of process over networks. Straber, Baumann, and Hohl [79] model agents as clusters of objects without references to the outside. The agent is the transitive closure over all the objects the main agent object contains a reference to. They can communicate with other agents either locally inside one location or globally with agents on other locations. Mobile Agents furthermore can migrate from one location to another. Mechanisms for the communication between agents and for the migration of agents have to be provided by the Mobile Agent System.

Serugendo, Muhugusa, and Tschudin [92] presents a comparative survey of formalisms related to mobile agents. It describes the π-calculus and its extensions, the Ambient calculus, Petri nets, Actors, and the family of generative communication languages. Each of these formalism defines a mathematical framework that can be used to reason about mobile code. They also show how these formalisms can be used to represent the mobility and communication aspects of two mobile code environment: Obliq [93] and Messengers [94].

The Chemical Abstract Machine [95] is an abstract machine designed to model a situation in which components move about a system and communicate when they come into contact. A concept of enforced locality using membranes to confine subsolutions allows the machines to implement classical process calculi or concurrent generalizations of the lambda calculus.

Angluin et al. [96] explore the power of small resource-limited mobile agents. They define the concept of stable computation of a function or predicate with a fairness condition on interactions. They show that all stably computable predicates are in *NL*.

Baumann and Radouniklis [80] propose to group agents in mobile agent environment. Agent groups consist of agents working together on a common task. Each agent works on a subtask. In order to perform their subtasks, agents themselves may dynamically create subgroups of agents.

This framework for process over networks emphasizes on high-level programming approaches for network applications. Mobile agent technology provides a method to migrate executable codes among computing devices in the network. Its essential idea is used by the network virtual machine, which is a part of the framework, as the underlay implementation utilities.

5.3 Example Scenario

We use the helicopter rescue application described in Chapter 3 to illustrate how to depict the behavior of the execution process using the process graph. When a people gets sick, he or she will find a helicopter to pick up him or her. When a helicopter carrying a sick people finds that there is a hospital available, it will send that sick people to that hospital and leave the hospital. When a sick people becomes healthy, he or she will leave the hospital.

5.4 Process Graph

The system that we define consists a collection of objects, which are representation of physical entities in a network. Objects are typed but addressless. Objects have their own (internal) states (drawn from a finite state space) and change the states as the process proceeds. The critical part of our systems is the rules, which specify how the system evolves. In particular, each rule describe how a process runs over a network, and how objects change their states.

A process graph depicts the behavior of processes running on the network. The basic composition blocks are objects, which are shown by circles. There could be many objects of the same type; when a process visits objects of the same type represented the same circle more than once, the objects visited before and after can be either exactly the same instance or a different one. The object that the process is currently visits is called the *active* object. There is exactly one active object at any time.

To represent the state change of an object, we have *state circle*s inside each object to represent the possible states of it. We assume that there can only be a finite number of states for each object. In the process graph (of the helicopter rescue system) shown in Figure 5.1, there are three types of objects, which are represented by three big circles labeled by people, helicopter, and hospital, respectively. Each object has two states, UB and B, which mean unbonded and bonded, respectively.

In a process graph, rules are represented as arrows. Each rule triggers a state change of one or two objects. A rule is either represented by an *internal state transition* or an *external state transition*.

An internal state transition connects from an internal state s of an object to another internal state s' of the same object to represent an internal state change of the object. The internal state change can be non-deterministic. The transition is firable if the object involved in the transition is active and is currently at the state s. After firing the transition, the object stays active and with s' being the current state.

An external state transition connects from a state edge of an object to a state edge of a possibly different object. The transition means that the two internal state changes, that it connects, will be fired at the same time. For example, in Figure 5.1, the external state transition represented by transition edge 1 will make a people object change the state from "UB" to "B" and a helicopter object change the state from "UB" to "B". To fire the transition, the two involved objects should be at the starting states. The transition represented by transition edge 1 can only be fired when the people object is the active object, the people object is in state "UB" and the helicopter object is in state "UB". After the transition is fired, only the object that the arrow points to will be active. That is, after the transition represented by transition edge 1 is finished, the helicopter object will become the current active object.

In Figure 5.1, edges labeled 1, 2, and 3 are external state transitions. If transition 2 is fired, the helicopter will remain bonded and a hospital object will become bonded. After transition 2 is finished, the hospital object will become active. Then the hospital object can fire transition 3, which makes the helicopter object become unbonded.

5.5 Computability of Process Graphs

In this section, we will study the computing power of process graphs. We start with definitions.

Let

$$\Sigma = \{A_1, ..., A_k\}$$

($k \geq 1$) be an alphabet of symbols. An instance of a symbol A_i, for some i, in Σ is called an *object* of type A_i, or simply an A_i-object. Without loss of generality, we call A_1 to be the *initial type*.

Each A_i is associated with a (nondeterministic) finite automaton (we still use A_i to denote it), which is a 3-tuple

$$A_i = (\mathcal{S}_i, \delta_i, q_{i0}),$$

where $\mathcal{S}_i = \{S_{i1}, ..., S_{il}\}$ (some $l \geq 1$) is a finite set of *internal states* (one can assume that the $\mathcal{S}_i$'s are disjoint), $\delta_i \subseteq \mathcal{S}_i \times \mathcal{S}_i$ is the set of the *internal* state transitions, and $q_{i0} \in \mathcal{S}_i$ is the initial state of the automaton A_i. We use $t_i : S_{iu} \rightarrow S_{iv}$ to denote a transition $t_i = (S_{iu}, S_{iv}) \in \delta_i$. In this way, an A_i-object

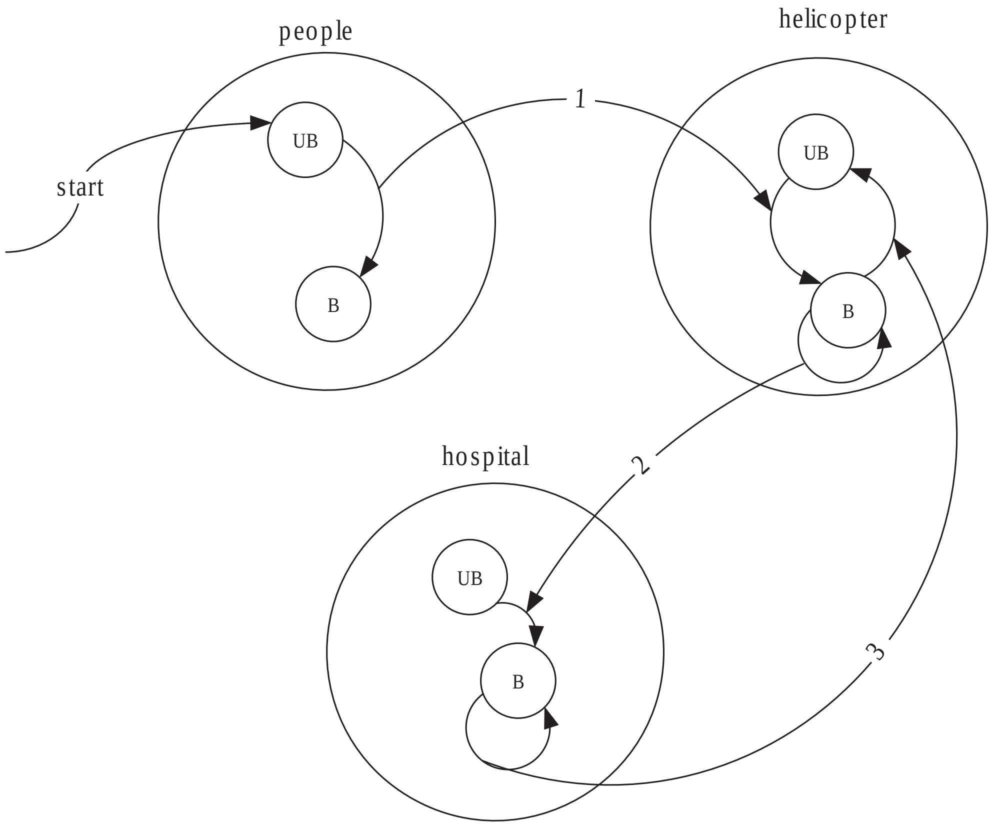


Figure 5.1: A Process Graph for The Helicopter Rescue System

itself is simply an instance of the finite automaton A_i.

Let $\mathcal{S} = \bigcup \mathcal{S}_i$ and $\delta = \bigcup \delta_i$. Inter-object communications are achieved by *external* transitions in Δ, defined as

$$\Delta \subseteq \Sigma \times \delta \times \Sigma \times \delta.$$

We denote an external transition $r \in \Delta$ is in the following rule-form:

$$r: (A_i, t_i) \rightarrow (A_j, t_j),$$

where $t_i \in \delta_i$ and $t_j \in \delta_j$ are internal state transitions.

In summary, a *process graph* is a tuple

$$G = \langle \Sigma, \Delta \rangle$$

where each component is defined in above.

We now define the semantics of the G. To specify an object O, we need only know its (unique) type A and its (unique) current state s of the finite automaton that is associated with the type; i.e., the O is an instance of (A, s), where for some i, $A = A_i \in \Sigma$ and $s \in \mathcal{S}_i$.

A *collection* $(\mathcal{C}, O)$ is a multiset $\mathcal{C}$ of objects with $O \in \mathcal{C}$ being the only *active* object.

Let $(\mathcal{C}, O)$ and $(\mathcal{C}', O')$ be two collections and r be a transition. We use

$$(\mathcal{C}, O) \rightarrow^r (\mathcal{C}', O')$$

to denote the fact that the collection $(\mathcal{C}, O)$ changes to the collection $(\mathcal{C}', O')$ by firing the transition r, which is defined formally as follows.

We first consider the case when r is an internal transition, say $t_i: S_{iu} \rightarrow S_{iv}$ in δ_i (i.e., the transition is inside an A_i-object specified by the automaton A_i). In this case, we require that the t_i does not appear in the left hand side of any external transition (i.e., there is no external transition in the form of $(A_i, t_i) \rightarrow (A_j, t_j)$, for any j). Hence, an internal transition can not fire alone when there is an external transition connects from the internal transition. We say that $O \rightarrow^{t_i} O'$ when O is at state S_{iu} and is of type A_i, and O' is the result of changing the current state in O with S_{iv}. Now,

$$(\mathcal{C}, O) \rightarrow^r (\mathcal{C}', O')$$

if the following conditions are satisfied:

- $O \rightarrow^{t_i} O'$.
- $\mathcal{C}'$ is the same as $\mathcal{C}$ except that the object O is changed into O'.

Therefore, when the internal transition t_i is fired, the active object must be at state S_{iu} and, after firing the transition, the current state of the object is S_{iv} and it remains as the active object.

Then, we consider the case when r is an external transition, say $r: \ (A_i, t_i) \rightarrow (A_j, t_j)$, where $t_i \in \delta_i$ and $t_j \in \delta_j$ are internal state transitions. In this case,

$$(\mathcal{C}, O) \rightarrow^r (\mathcal{C}', O')$$

if, for some $O'' \in \mathcal{C}$, and some object O''',

- $O \rightarrow^{t_i} O'''$,
- $O'' \rightarrow^{t_j} O'$, and
- $\mathcal{C}'$ is the result of, in $\mathcal{C}$, replacing O with O''' and replacing O'' with O'.

Therefore, when the external transition r is fired, the active object O must be an A_i-object in state S_{iu} and an A_j-object O'' in state S_{jp} is nondeterministically chosen from the collection. The A_i-object O will transit from state S_{iu} to S_{iv} (and evolve into O''' defined in above), and the A_j-object O'' will transit from state S_{jp} to S_{jq} (and evolve into O' defined in above), in parallel. After the transition is fired, the active object is changed from O to O'.

The $(\mathcal{C}, O)$ is *initial* if all objects in $\mathcal{C}$ are in their initial states, the O is the designated initial object (i.e., the type of O is the initial type A_1 and O is at the initial state of automaton A_1), and a pre-defined *initial constraint* is satisfied. The initial constraint comes along with the definition of the process graph to restrict the number of objects for certain types. In this chapter, we use the simplest form of initial constraints: for each type A_i, the number of objects of type A_i is either constrained to be a constant (like 5) or is unconstrained.

We write

$$(\mathcal{C}, O) \rightsquigarrow_G (\mathcal{C}', O') \tag{5.1}$$

if there are collections $(\mathcal{C}_0, O_0), \cdots, (\mathcal{C}_m, O_m)$, for some m, such that

$$(\mathcal{C}_0, O_0) \xrightarrow{r_1} (\mathcal{C}_1, O_1) \cdots \xrightarrow{r_m} (\mathcal{C}_m, O_m), \tag{5.2}$$

for some transitions $r_1, \cdots, r_m$ in G.

In fact, G defines a computing model that modifies a multiset $(\mathcal{C}, O)$ into another multiset $(\mathcal{C}', O')$ through $(\mathcal{C}, O) \rightsquigarrow_G (\mathcal{C}', O')$. To characterize the quantitative relationships that the G can compute, we need more definitions.

Consider a set $T \subseteq \Sigma \times \mathcal{S}$. For each pair $t = (A, s) \in T$, we use $\#_t(\mathcal{C}, O)$ to denote the number of the objects in $\mathcal{C}$ such that, each of which is of type A and at state s. Clearly, when a proper ordering is applied on T, we may collect the numbers $\#_t(\mathcal{C}, O)$, $t \in T$, into a vector called $\#_T(\mathcal{C}, O)$. We use $R_{G,T}$, called the binary reachability of G wrt T, to denote the set of all vector pairs $(\#_T(\mathcal{C}, O), \#_T(\mathcal{C}', O'))$ for all initial collections $(\mathcal{C}, O)$ and collections $(\mathcal{C}', O')$ satisfying $(\mathcal{C}, O) \rightsquigarrow_G (\mathcal{C}', O')$. In particular, when $T = \Sigma \times \mathcal{S}$, we simply use R_G to denote the $R_{G,T}$.

Before we show a characterization of the binary reachability, we need more definitions. An n-dimensional *vector addition system with states* (VASS) M is a 5-tuple

$$\langle V, p_0, p_f, S, \delta \rangle$$

where V is a finite set of *addition vectors* in Z^n, S is a finite set of *states*, $\delta \subseteq S \times S \times V$ is the *transition relation*, and $p_0, p_f \in S$ are the *initial state* and the final state, respectively. Elements (p, q, v) of δ are called *transitions* and are usually written as $p \to (q, v)$. A *configuration* of a VASS is a pair (p, u) where

$p \in S$ and $u \in \mathrm{N}^n$. The transition $p \rightarrow (q, v)$ can be applied to the configuration (p, u) and yields the configuration $(q, u+v)$, provided that $u + v \geq \mathbf{0}$ (in this case, we write $(p, u) \rightarrow (q, u+v)$). For vectors x and y in N^n, we say that x can *reach* y, written $x \rightsquigarrow_M y$, if for some j,

$$(p_0, x) \rightarrow (p_1, x + v_1) \rightarrow \cdots \rightarrow (p_j, x + v_1 + ... + v_j)$$

where p_0 is the initial state, p_j is the final state, $y = x + v_1 + ... + v_j$, and each $v_i \in V$. It is well-known that Petri nets and VASS are equivalent. Consider a number $k \leq n$. We use $x(k)$ to denote the result of projecting the n-ary vector x on its first k components, and use $R_M(k)$ to denote all the pairs $(x(k), y(k))$ with $x \rightsquigarrow_M y$. When $k = n$, we simply write R_M for $R_M(k)$. We say that a graph process G can be simulated by a VASS M if for some number k, $R_G = R_M(k)$. We say that a VASS M can be simulated by a graph process G if for some T, $R_{G,T} = R_M$. If both ways are true, we simply say that they are equivalent (in terms of computing power).

Theorem 1. Process graphs are equivalent to VASS.

Proof. The proof consists of two parts. First, we prove that the process graph can simulate n dimension VASS M.

It is well known that VASS with only one state (or essentially no state) are equivalent to VASS (with many states), and therefore, we assume that M is specified by a number of addition vectors

$$v_i = (v_{i,1}, v_{i,2}, ..., v_{i,n}),$$

with $1 \leq i \leq u$, and the VASS is of n dimensions.

Now we describe how to construct a process graph to simulate the VASS M. To make the description more concrete we first give the process graph that simulates the addition vector $(-3, +2)$ in Figure 5.2.

We have three kinds of objects in the process graph, namely, *counter object*, *control object*, and *addition vector object*. Each object is associated with internal states.

Since the VASS M is of n dimensions, we have totally n types of counter objects, namely, i-th type counter objects, for $1 \leq i \leq n$. For each i-th type counter object, CO_i, its internal states are $CS_{i,1}$, $CS_{i,2}$,

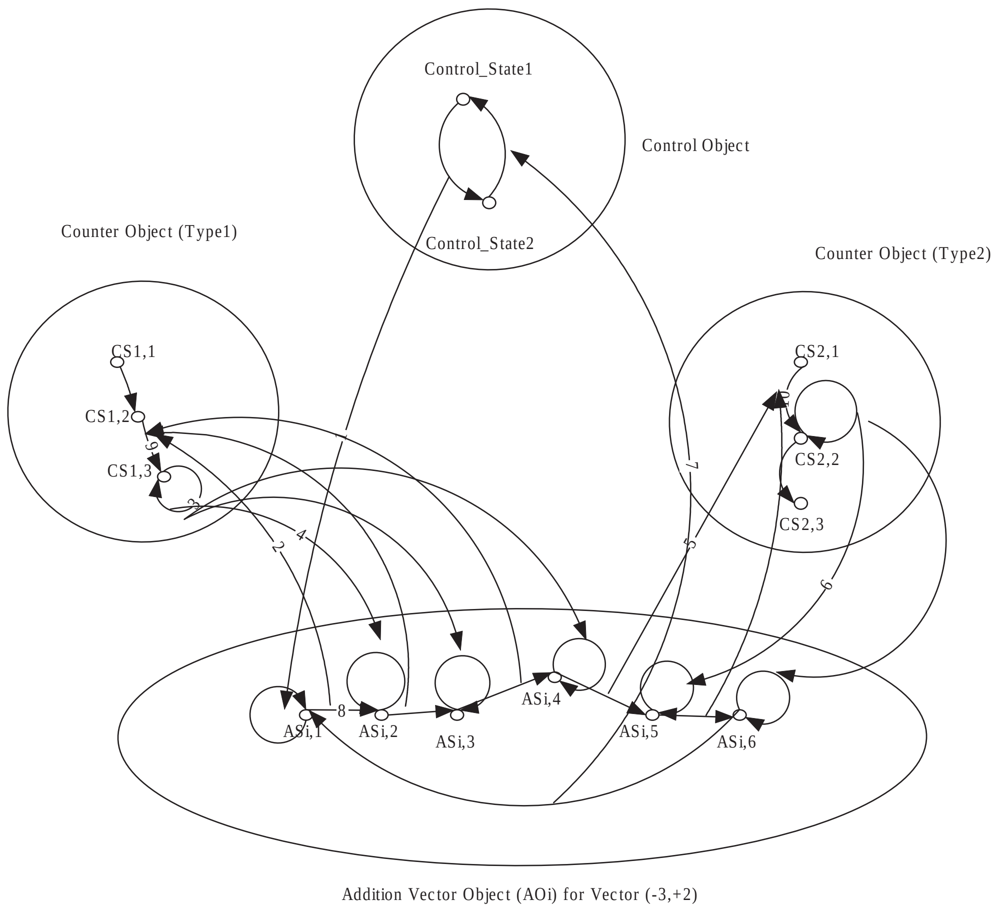


Figure 5.2: Simulate VASS Using Process Graph

and $CS_{i,3}$. Recall that when the VASS M runs, it changes the value of its configuration (which is simply a vector, called the configuration vector). The number of i-th type counter objects in state $CS_{i,2}$ represents the current value of the i-th component of the configuration vector in the VASS M. In Figure 5.2, there are two kinds of counter objects, namely Type1 and Type2.

We also have a *control object* in the system. For the whole system, there is only one *control object* (Recall that, this constitutes part of the initial constraint of the process graph). This object has two states, namely $Control\_State1$ and $Control\_State2$. The *control object* is also the starter of the process graph. Its initial state is $Control\_State1$.

For each addition vector, $v_i = (v_{i,1}, v_{i,2}, ..., v_{i,n})$, we have a corresponding *addition vector object* type. For such type, there is exactly one *addition vector object* in the whole system. We use $|v_{i,j}|$ to represent the absolute value of $v_{i,j}$. The number of internal states *addition vector object* AO_i, which corresponds to the addition vector v_i, is

$$l = \left(\sum_{j=1}^{n} |v_{i,j}| \right) + 1.$$

In Figure 5.2, we have six internal states for the addition vector object that represents the addition vector $(-3, +2)$.

Now, we describe the transitions.

Suppose now that we consider the moment when M fires its i-th addition vector, say, $v_i = (v_{i,1}, v_{i,2}, ..., v_{i,n})$. At this time in the process graph, the active object is the control object. The i-th addition vector is simulated by a (rather long) sequence of transitions in the process graph. Such simulation is shown in 5.2 for an example $v_i = (-3, +2)$ (with $n = 2$). Initially, the control object fires an external transition, that changes the control object's state from $Control\_State1$ to $Control\_State2$. This external transition, at the same time, also triggers the state change of the *addition vector object* representing v_i, AO_i. More precisely, AO_i issues a transition from $AS_{i,1}$ to $AS_{i,1}$. In Figure 5.2, this external transition is represented by the transition edge labeled 1. Now the active object is AO_i and at state $AS_{i,1}$. Then AO_i will perform the first operation specified in the vector, $v_{i,1}$. AO_i will issue an external transition. It will change its state from $AS_{i,1}$ to $AS_{i,2}$. At the same time, this external transition will make the counter object, CO_i, change the state. Depending on whether $v_{i,1}$ is positive or not, there are two cases.

If $v_{i,1}$ is negative, this means we need to decrease the number of CO_1 type objects in state $CS_{1,2}$. Thus the CO_1 object will change the state from $CS_{1,2}$ to $CS_{1,3}$. This accomplishes one subtraction. In Figure 5.2, this external transition is represented by the transition edge labeled 2. Now CO_1 becomes the current active object. For the process to continue, it needs to transfer the active role back to the *addition vector object*. Then, CO_1 will issue an external transition. It will change its state to change from $CS_{1,3}$ to $CS_{1,3}$. The *addition vector object* AO_i will change the state from $AS_{i,2}$ to $AS_{i,2}$. Now the *addition vector object* will become the current active object. In Figure 5.2, this external transition is represented by the transition edge labeled 4.

If $v_{i,1}$ is positive, this means we need to increase the number of CO_1 type objects in state $CS_{1,2}$. Thus the CO_1 object will change the state from $CS_{1,1}$ to $CS_{1,2}$. This accomplishes one addition. In Figure 5.2, this external transition is represented by the transition edge labeled 5. Now CO_1 becomes the current active object. For the process to continue, it needs to transfer the active role back to the *addition vector object*. Then, CO_1 will issue an external transition. It will change its state from $CS_{1,3}$ to $CS_{1,3}$. The *addition vector object* AO_i will change the state from $AS_{i,2}$ to $AS_{i,2}$. Now the *addition vector object* will become the current active object. In Figure 5.2, this external transition is represented by the transition edge labeled 6.

At this point, the *addition vector object* is at the state $AS_{i,2}$ and it is the current active object.

To make this description general, let's assume the *addition vector object* is at the state $AS_{i,j}$, where $j = \left(\sum_{h=1}^{k} |v_{i,h}|\right) + l$ with $1 \leq l \leq v_{i,k+1}$. To accomplish the operation of $v_{i,k+1}$, there are still $|v_{i,k+1}| - l$ more steps to go.

AO_i will issue an external transition. It will change its state from $AS_{i,j}$ to $AS_{i,j+1}$. At the same time, this external transition will make the counter object, CO_{k+1}, change the state. Depending on whether $v_{i,k+1}$ is positive or not, there are two cases.

If $v_{i,k+1}$ is negative, this means we need to decrease the number of CO_{k+1} type objects in state $CS_{k+1,2}$. Thus the CO_{k+1} object will change the state from $CS_{k+1,2}$ to $CS_{k+1,3}$. This accomplishes one subtraction. Now CO_{k+1} becomes the current active object. For the process to continue, it needs to transfer the active role back to the *addition vector object*. Then, CO_{k+1} will issue an external transition. It will change its state to change from $CS_{k+1,3}$ to $CS_{k+1,3}$. The *addition vector object* AO_i will change the state from $AS_{i,j+1}$ to $AS_{i,j+1}$. Now the *addition vector object* will become the current active object and its state is in $AS_{i,j+1}$.

If $v_{i,k+1}$ is positive, this means we need to increase the number of CO_{k+1} type objects in state $CS_{k+1,2}$. Thus the CO_{k+1} object will change the state from $CS_{k+1,1}$ to $CS_{k+1,2}$. This accomplishes one addition. Now CO_{k+1} becomes the current active object. For the process to continue, it needs to transfer the active role back to the *addition vector object*. Then, CO_{k+1} will issue an external transition. It will change its state to change from $CS_{k+1,2}$ to $CS_{k+1,2}$. The *addition vector object* AO_i will change the state from $AS_{i,j+1}$ to $AS_{i,j+1}$. Now the *addition vector object* will become the current active object and its state is in $AS_{i,j+1}$.

As the above process continues, at the point that AO_i is active and at the state $AS_{i,z}$, $z = \left(\sum_{j=1}^{n} |v_{i,j}|\right) + 1$, and AO_i has performed the internal transition from $AS_{i,z}$ to $AS_{i,z}$, all the operations of $v_i = (v_{i,1}, v_{i,2}, ..., v_{i,n})$ have been finished. Now we need to make the control object to be active to fire a next addition vector. Thus AO_i fires an external transition where AO_i changes its state from $AS_{i,z}$ to $AS_{i,1}$. At the same time, the control object changes its state from $Control\_State2$ to $Control\_State1$. Thus the control object becomes the active object again. In Figure 5.2, this external transition is represented by the transition edge labeled 7. The addition vector $v_i = (v_{i,1}, v_{i,2}, ..., v_{i,n})$'s function is accomplished. The control object can continue to simulate firing a next addition vector.

For each addition vector in V, we just repeat the previous process. Because of the internal states of each object, the process graph will simulate the behavior of the VASS M exactly.

For the second part of the proof, we need show that a VASS M can simulate a process graph G. Suppose that there are m types of objects in G, and without loss of generality, we assume that internal states in different types of objects are all distinct. Therefore, an internal state of an object can uniquely tell the type of the object (hence, we do not need to refer an object type in an internal transition and an external transition, in below). Suppose that all the internal states, S, are properly ordered (we use $position(s)$ to indicate the position of state s in the ordering), and we use $\#_s$ to indicate the current number of objects in state s at some moment when the process graph runs. We use $\#$ to indicate the array of all the $\#_s$. The VASS M constructed in below is to update the vector $\#$ while transitions in G is executed.

In M, the states are exactly those in S, the internal states of objects in G.

For each internal transition, say $s \rightarrow s'$, we have an addition vector along with a state transition in M as follows: $\langle s, (0, \cdots, 0, -1, 0 \cdots, 0, +1, 0, \cdots, 0), s' \rangle$, where the -1 is at position $position(s)$ and the $+1$ is at position $position(s')$. The vector corresponds to the fact that, after firing the internal transition, there is one object in state s evolving into an object in state s'. The state transition (from s to s') in M corresponds to the fact that, after firing the internal transition, the active object is transferred from an object at state s to an object at state s'.

For each external transition, say $(s_1, s_2) \rightarrow (s_3, s_4)$ (where (s_1, s_2) and (s_3, s_4) represent two internal transitions), we have an addition vector along with a state transition in M as follows:

$$\langle s_1, (0, \cdots, 0, -1, 0 \cdots, 0, +1, 0, \cdots, 0, -1, 0 \cdots, 0, +1, 0, \cdots, 0), s_4 \rangle,$$

where the two -1's are at positions $position(s_1)$ and $position(s_3)$, respectively, and the two $+1$'s are at positions $position(s_2)$ and $position(s_4)$, respectively. The vector corresponds to the fact that, after firing the external transition, there is one object in state s_1 evolving into an object in state s_2, and at the same time, there is one object in state s_3 evolving into an object in state s_4. The state transition (from s_1 to s_4) in M corresponds to the fact that, after firing the external transition, the active object is transferred from an object at state s_1 to an object at state s_4.

Clearly, M faithfully simulates G.

$\square$

The above theorem characterizes the computing power of process graphs, when the graphs are understood as computation devices. In the following, we will treat process graphs as language acceptors and therefore, we can characterize the processes that are defined by such graphs. We need more definitions.

Let

$$\Pi = \{a_1, \cdots, a_n\}$$

($n \geq 1$) be an alphabet of *(activity) labels*. Now, we are given a function that assigns each external transition with either Λ (empty label) or an activity label in Π. Recall that we write $(\mathcal{C}, O) \rightsquigarrow_G (\mathcal{C}', O')$ if there are collections $(\mathcal{C}_0, O_0), \cdots, (\mathcal{C}_m, O_m)$, for some m, such that

$$(\mathcal{C}_0, O_0) \xrightarrow{r_1} (\mathcal{C}_1, O_1) \cdots \xrightarrow{r_m} (\mathcal{C}_m, O_m),$$

for some transitions $r_1, \cdots, r_m$ in G. We use α to denote the sequence of such labels for external transitions in $r_{i_1}, \cdots, r_{i_m}$. To emphasize the α, we simply write $(\mathcal{C}, O) \rightsquigarrow_G^{\alpha} (\mathcal{C}', O')$ for $(\mathcal{C}, O) \rightsquigarrow_G (\mathcal{C}', O')$ in this case. We say that a word $\alpha \in \Pi^*$ is a *process* defined in the process graph G if there is an initial collection $(\mathcal{C}, O)$ such that $(\mathcal{C}, O) \rightsquigarrow_G^{\alpha} (\mathcal{C}', O')$ for some $(\mathcal{C}', O')$. We use $L(G)$ to denote the set of all processes defined by G.

A multicounter machine M is a nondeterministic finite automaton (with one-way input tape) augmented with a number of counters. Each counter takes nonnegative integer values and can be incremented by 1, decremented by 1, and tested for 0. It is well known that when M has two counters, it is universal. A counter is blind if it can not be tested for 0, however, when its value becomes negative, the machine crashes. A blind counter machine is a multicounter machine M whose counters are blind. It is known that blind counter machines are essentially VASS treated as a language acceptor. Therefore,

Theorem 2. The class of processes defined by process graphs are exactly the class of languages accepted by blind counter machines.

From the above theorem, it is clear that process graphs can define fairly complex processes, which are not necessarily regular, context free, or semilinear.

Currently, it is a difficult problem to characterize a nontrivial class of process graphs that exactly define regular processes. In the following, we will define a special class of process graphs, by looking at their graph structures, such that their computing power is weaker (and hence they have nicer and stronger properties). Recall that, in a process graph, the initial constraint comes implicitly. Such a constraint specifies, in an initial collection, for each type A_i, the number of objects of type A_i is either constrained to be a constant (like 5) or is unconstrained. When an object type is constrained, each object of the type is called a *server*. An *all-server* process graph is one where each object in its initial collection is a server. Clearly, since there are only finitely many objects in an initial collection of servers, we have:

Theorem 3. The class of processes defined by all-server process graphs are exactly the class of regular languages.

A k-*server* process graph is one where there are k-servers, along with some other unconstrained objects. We further require that every external transition in the graph can only connect between a server and an unconstrained object or between the servers (hence, unconstrained objects can not communicate between each other directly). From the construction in a previous theorem, a VASS can be simulated by a k-server process graph for some k. The interesting case is when the k is fixed. In below, we are going to show that 1-server process graphs are strong enough to simulate process graphs and hence to simulate VASS. The proof idea is to simulate direct communications between two clients (unconstrained objects) by indirect communications via the server.

Theorem 4. Each process graph can be simulated by a 1-server process graph.

Proof. For a process graph G depicted in Figure 5.3, which has n external transitions, we can construct the 1-server process graph, G' shown in Figure 5.4, in the following way, which can be generalized to any process graph.

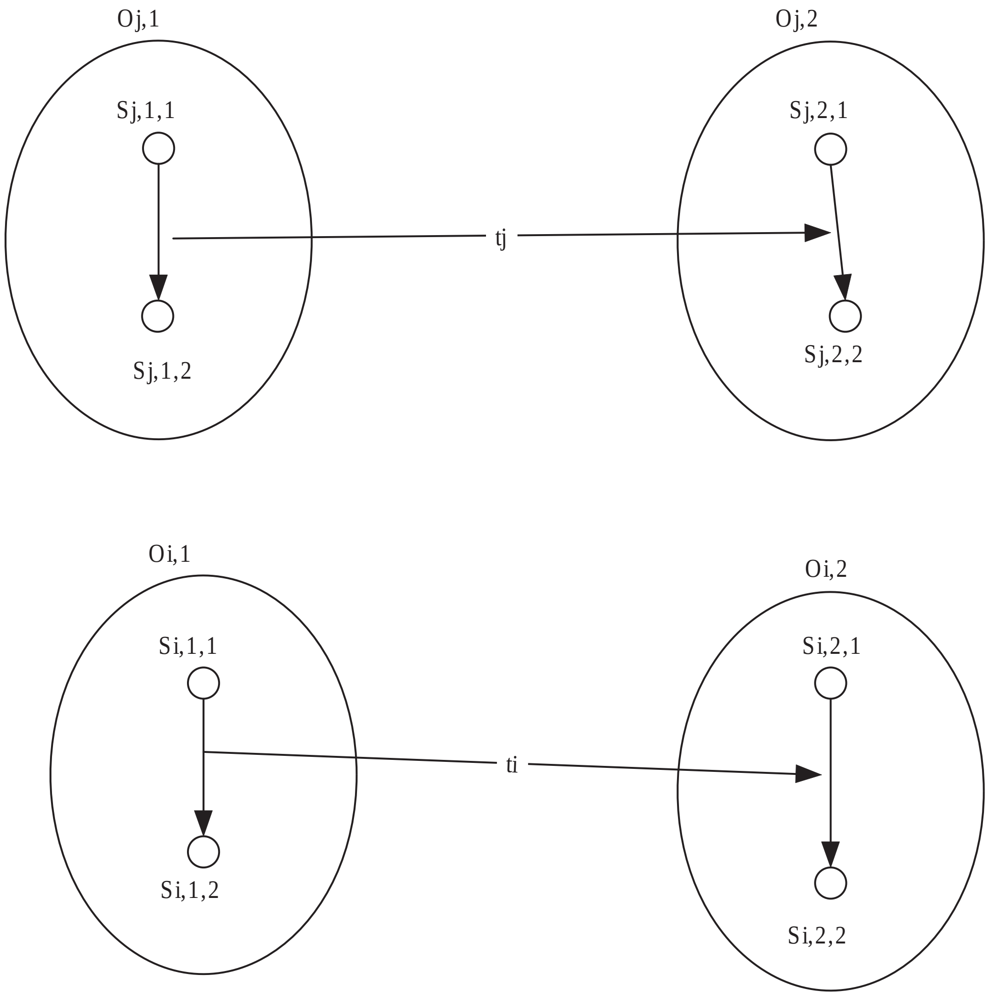


Figure 5.3: Process Graph G

The server object, SO, has one initial state, S_{start}. For each external transition, t_i, in G, SO in G' has a corresponding internal state, CS_i. All the client objects in G' are exactly the same as those objects in G.

Without losing generality, for external transition t_i in G, it involves the internal state transition of two objects, $O_{i,1}$ and $O_{i,2}$. $O_{i,1}$ changes the internal state from $S_{i,1,1}$ to $S_{i,1,2}$. $O_{i,2}$ changes the internal state from $S_{i,2,1}$ to $S_{i,2,2}$. In G', this can be simulated in the following way. The corresponding client objects for $O_{i,1}$ and $O_{i,2}$ are $O'_{i,1}$ and $O'_{i,2}$. We have two external transitions, namely $t'_{i,1}$ and $t'_{i,2}$ to represent t_i. For transition $t'_{i,1}$, the client object $O'_{i,1}$ changes the internal state from $S'_{i,1,1}$ to $S'_{i,1,2}$. At the same time, the server object, SO, changes the internal state from S_{start} to the corresponding state of t_i, CS_i. Now, SO is the active object. Then for the second external transition, $t'_{i,2}$ in G', SO changes the internal state from CS_i

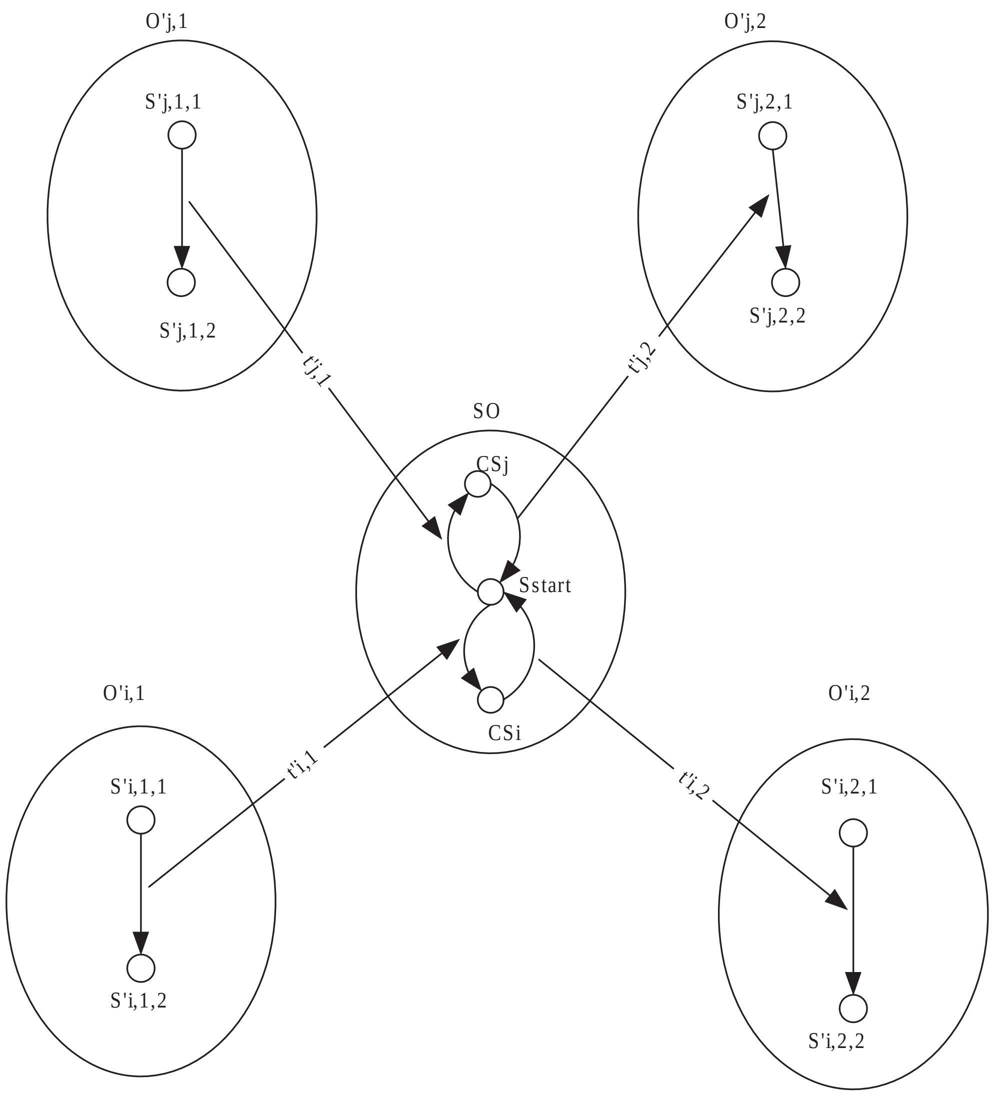


Figure 5.4: 1-server Process Graph G'

to S_{start}. At the same time, $O'_{i,2}$ changes the internal state from $S'_{i,2,1}$ to $S'_{i,2,2}$. $O'_{i,2}$ is the current active object. Thus the transition t_i is simulated by two transitions, $t'_{i,1}$ and $t'_{i,2}$.

It is left to the reader to check that G' indeed simulates G. □

5.6 Implementation of Functions Represented by Process Graph Using Underlay Networks

In this section, we describe how to implement the functions described by a given process graph by using under layer network protocols.

5.6.1 Process ID and Token

In a large distributed system, without a central server, it is not feasible to designate some site to assign the unique process ID to each process to uniquely differentiate them. We adopt a randomized approach to this problem. At the beginning of a process, the beginner object generates a random number as the process ID. If the value space of the random number is big enough, the probability that two processes have the same process ID is really small.

For each process, there is a token passed among the involved objects. In the token, the context of the process, such as the previous execution results, is also recorded for the next active object to check whether it can start the execution and obtain the desired execution results of the instructions preceding it. The constraints can be checked by looking at the context information recorded in the token. If they are satisfied, it will start the execution. After the execution, the current active object needs to pass the token and informs the next active object to start the execution.

5.6.2 Locating and Locking Entities

To find the interactive objects, we need to provide an efficient method to locate objects of desired types. There are two approaches. One is a completely distributed approach. Each object is responsible for finding out the desired objects through cooperations among them by broadcasting or multicasting (to a limited scope) the lookup request to other objects in the network. The other approach is a hierarchical approach, which deploys some servers in the network to maintain objects information, which is similar to the naming and discovery approach, such as Jini [56], Salutation [89], and Service Location Protocol [90]. All objects need to register their properties with one of the servers. An object can submit the lookup request to the server, which returns the handle of the desired object by looking up its local lookup table, or collaborating with other servers in the system. In our system, we choose the distributed approach to avoid the bottleneck problem that may be brought by hierarchical approach.

In the single process system, only one process is active in the system to run from the beginning to the end. Thus it is not necessary to lock involved objects to ensure correctness and consistency. While to ensure the correctness of operations in a multiple process system, it is necessary for the process starter to lock all the objects involved in the process at the beginning and unlock them at the end. Only operations among the objects in the same process can interact with each other.

5.6.3 Message Mechanism

The external transition involves the communications, or message exchanges, among objects. Except the start messages that starts the process execution in the network, all other messages in the process graph are unicast messages. A unicast message is from a single source to a single destination.

When an active object is triggering an external transition, it informs the destination object to make the corresponding changes. For example, in Figure 5.1, when the external transition labeled 1 is triggered, the active people object also sends a message to the helicopter object to inform the helicopter object to change the state from “UB” to “B”.

5.6.4 Synchronization Among Objects

In the process graph, when an external transition is fired, the two involved objects change the state at the same time. In real networks, we can not always have two state changes on different sites happen at exactly the same time. To implement this, we need to ensure that before the two state changes are finished, no other state change can happen. To achieve this, each time, the external transition between two involved objects should not be interrupted. In the multiple process system, this can be achieved by the lock mechanism to ensure the operations among the involved objects to be correct and atomic. In the single process system we discuss, if we have an transition edge connecting a and b with the arrowhead pointing at b, the two involved objects are a and b and b is the active object after the transition is fired. Then a should report to b when its state change is finished. b will know when a and it both finish the state change. After both a and b finish, b will continue to fire the next transition and the role of active object will be passed on.

5.7 Summary

In this chapter, we discuss the concept of network process. A network process is a executing program running over network objects. In a real network, a network process first grabs some network entities, and consume their resources. When some entities finish their computation tasks, the process would release the entities and then move on to others. In our model, we abstract this phenomena as: the process walks through objects, and objects change their states as the process passing by. We discuss only the single process model, in which only one process is running in the network. Adopting the idea of graph representation tools usually used by automata theory, we propose a class of process graphs to depict the behavior of a network process. We also prove that the computing power of the process graphs is equivalent to VASS. Finally, we discuss how to implement a process graph on a network.

CHAPTER 6

CONCLUSION

In this dissertation, we describe a framework for grouping through virtual machine over networks. Provided the increasing number of devices involve in network applications, it is necessary to investigate approaches to organize those devices based on application requirements and make them perform the given tasks. The essential functionalities of computer network is to establish relations among them, or grouping. It is natural to investigate how to group entities in the network efficiently according to application requirements, which gives motivations to researches on clustering. We describe the clustering approaches we propose, Size-bounded Multi-hop Clustering (SMC), Bandwidth-adaptive Clustering (BAC), Dual-clusterhead Clustering (DCC), and Typed Clustering (TC).

The current approaches to programming for network applications are not proper, especially for large scale networks. One challenge is that programmers have to focus on underlay details instead of functionalities of applications. Device heterogeneity is another challenge brought by large-scale networks. This can lead to duplicated different versions of the same application for different computation devices due to the fact that some existing applications are typically developed for specific devices or system platforms.

In this dissertation, we proposed a high-level solution to these issues, by defining a specification language, NetSpec, for (functional) grouping over network and also a script language of a generalized version of a molecular computing model BCS. The specification language can be translated into under layer network protocols running on a common virtual machine. In the dissertation we in detail described a compiler that translates a NetSpec specification into instructions supported by the network virtual machine. We described how to implement the compiler and a transition scheduler based on the essential functionalities of NetSpec. The instruction set supported by the network virtual machine is powerful enough to encode complicated applications, and yet simple enough to efficiently parse into network protocols. To support the instruction set, the network virtual machine further deals with synchronization, group communication control, and concurrency control.

Essentially, NetSpec specifies how groups of network objects evolve. Looking from the angle of each individual node, such an evolution can be characterize as a thread of atomic transitions, each of which is local; e.g., involving two network nodes. Inspired by this view, we defined network process graphs and studied, theoretically, their computability and realization on a physical network.

APPENDIX

APPENDIX A

SYNTAX OF NETSPEC

```
Rule format: symbol ::= symbols [ | symbols |  ]

Nonterminals are surrounded by angle brackets.

Terminals are surrounded by single quotes.

Curly braces are used for symbol grouping.

* means zero or more of the previous symbol.

+ means one or more of the previous symbol.

? means zero or one ofthe previous symbol.

. means any character.

[XYZ] means the same thing as {’X’ | ’Y’ | ’Z’}.

[A-Z] means the same thing as {’A’ | ’B’ | ’C’ |  | ’Z’}.

[ˆA] means any character except for ’A’.

<program> ::= <def>+ <def> ::= <const_def>
        | <class_def>
        | <bond_def>
        | <transition_def>

<primitive_def> ::= <type> <id> ’;’

<const_def> ::= ’const’ <type> <id> { ’=’ <literal> } ? ’;’

<class_def> ::= ’Class’ <id> ’{’ {<primitive_def> | <const_def>}*’}’

<bond_def> ::= ’Bond’ <id> ’(’ <set_type> <id> {’,’ <set_type>
<id>}* ’)’ ’{’ <select_stmt>* <expr> ’;’ ’}’

<set_type> ::= <id> ’Set’

<select_stmt> ::= <select> ’(’ <type> <id> {’,’ <type> <id>}* ’)’
’From’ <location> {’Where’ <expr>}? ’;’

<bond_select_stmt> ::= <bondselect> ’(’ <type> <id> {’,’ <type>
<id>}* ’)’ ’From’ <location> {’Where’ <expr>}? ’;’

<select> ::= ’Select’

<bondselect> ::= ’BondSelect’ | ’BondSelectEach’

<transition_def> ::= ’Transition’ <id> ’(’ <transition_param> <id>
                     {’,’ <transition_param> <id>}* ’)’ ’{’
                     <select_stmt>* ’if’ ’(’ <expr> ’)’ ’{’
                     <transition_stmt>* ’}’ ’}’
```

```
<transition_param> ::= <set_type> | <id>

<transition_stmt> ::= <atom_stmt>
                    | ’doparallel’ ’{’ <atom_stmt>* ’}’

<atom_stmt> ::= ’new’ ’bond’ <id> <id> ’;’
              | <id> ’join’ <id> ’;’
              | <id> ’leave’ ’;’
              | <id> ’switch’ <id> ’;’
              | <location> <assign_op> <expr>

<expr> ::= <expr> <binary_op> <expr>
         | <unary_op> <expr>
         | <unit>

<assign_op> ::= ’=’
              | ’+=’
              | ’-=’
              | ’*=’
              | ’/=’
              | ’%=’

<binary_op> ::= ’||’
              | ’&&’
              | ’==’
              | ’!=’
              | ’<=’
              | ’>=’
              | '<'
              | '>'
              | '*'
              | '/'
              | '%'
              | '+'
              | '-'

<unary_op> ::= '!'
             | '+'
             | '-'
             | '#'

<location> ::= <id>

<unit> ::= <location>
         | <literal>

<type> ::= 'Integer'
         | 'Real'
         | 'Boolean'
         | 'String'

<id> ::= {<letter> | <digit>} {<letter> | <digit> | '_'}*

<literal>::= <integer_literal>
           | <real_literal>
           | <boolean_literal>
```

```
            | <string_literal>
<integer_literal> ::= <digit>+

<real_literal> ::= <digit>* '.'<digit>+

<boolean_literal> ::= 'true' | 'false'

<string_literal> ::= '"' {[^"] | '\' .}* '"'

<letter> ::= [a-zA-Z]

<digit> ::= [0-9]
```

APPENDIX B

SPECIFICATION FOR EXAMPLE SCENARIOS

B.1 Helicopter Rescue System Specifications

```
Class People {
    Boolean healthy;
    Integer age;
};

Class Helicopter {
    Integer color;
    const Integer capacity;
};

Class Hospital {
    Integer ID;
    const Integer capacity;
}


Bond People_Helicopter (People Set ps, Helicopter Set hes) {
    BondSelectEach (People p) From ps Where p.healthy==false && #(ps)>=1;
    BondSelect (Helicopter h) From hes Where #(hes)==1 && #(ps) <=h.capacity;
};

Bond Hospital_People (Hospital Set hos, People Set ps) {
    BondSelectEach (People p) From ps
    Where p.healthy==false || p.healthy==true;
    BondSelect (Hospital h) From hos
    Where #(ps)<=h.capacity && #(hos)==1 && #(ps)>=1;
};

Transition T1 (People Set ps, People_Helicopter ph) {
    Select (People p) From ps;
    Select (Helicopter h) From ph;
    if(p.healthy==false && #(ph.ps)<h.capacity)
    {
        p join ph;
    }
};

Transition T2 (People Set ps, Helicopter Set hes) {

    Select (People p) From ps;
    Select (Helicopter h) From hes;
    if(p.healthy==false)
    {
        new Bond People_Helicopter ph(p, h);
    }
};

Transition T3 (People_Helicopter ph, Hospital_People hp) {
    Select (People p) From ph;
    Select (Hospital h) From hp;
    if(#(hp.ps)<h.capacity)
    {
        p switch hp;
    }
};
```

```
Transition T4 (People_Helicopter ph, Hospital Set hos){
    Select (Hospital hospital) From hos;
    Select (People p) From ph;
    new Bond Hospital_People hp(p, hospital);
};

Transition T5 (Hospital_People hp) {
   Select (People p) From hp;
   if (p.healthy==true)
   {
      p leave;
   }
};
```

B.2 Highway Information System Specifications

```
Class Vehicle {
    Integer plate_number;
};

Class Infocenter {
    Integer capability;
};

Bond Infocenter_Vehicle (Infocenter Set is, Vehicle Set vs) {
    BondSelect From Where #(is)==1 && #(vs)>=1;
};

Transition T1 (Infocenter_Vehicle iv, VehicleSet vs) {
    Select (Vehicle v) From vs;
    v join iv;
};
```

```
Transition T2 (Infocenter Set is, Vehicle Set vs) {
    Select (Vehicle v) From vs;
    Select (Infocenter i) From is;
    new Infocenter_Vehicle iv (i,v);
};
```

B.3 Pervasive Marketing System Specifications

```
const Integer initial_money=1000;

Class Bank {
    Integer money;
};

Class Customer{
    Integer money;
    Boolean initialized;
};

Class Vender{
    Integer money;
};

Class Goods{
    Integer value;
};


Bond Vender_Goods(Vender Set vs, Goods Set gs) {
    BondSelect From Where #(vs)==1 && #(gs)>=1;
```

```
};

Bond Customer_Goods(Customer Set cs, Goods Set gs){
    BondSelect From Where #(cs)==1 && #(gs)>=1;
};



Transition T1(Customer Set cs, Vender_Goods vs){
    Select (Customer c) From cs;
    Select (Goods g) From vs;
    select (Vender v) From vs;
    if(c.money>=g.value)
    {
        c.money-=g.value;
        v.money+=g.value;
        new Bond Customer_Goods cg (c,g);
    }
};

Transition T2(Customer_Goods gs, Vender_Goods vs){
    Select (Customer c) From gs;
    Select (Goods g) From vs;
    select (Vender v) From vs;
    if(c.money>=g.value)
    {
        c.money-=g.value;
        v.money+=g.value;
        g switch vs;
    }
};

Transition T3 (Bank Set bs, Customer Set cs){
    Select (Customer c) From cs;
    Select (Bank b) From bs;
    if(c.initialized==false)
    {
        b.money-=initial_money;
        c.money=initial_money;
        c.initialized=true;
    }
};
```

APPENDIX C

HELICOPTER RESCUE SYSTEM INSTRUCTION SEQUENCE PROGRAM

```
Class People {
    Boolean healthy;
    Integer age;
};

Class Helicopter {
    Integer color;
    Integer capacity;
};

Class Hospital {
    Integer ID;
    Integer capacity;
};


program T1 {

test(this, [type==People]);

test(this, [type==People_Helicopter]);

b=findlockbond([type==People_Helicopter]]);

p=findlocknode([type==People]);

h=findinternalnode([type==Helicopter], b);
```

```
jump([NOT(p.healthy==false && numnode(b.ps)<h.capacity)], end);

join(p, b);

end: checkbondintegrity(b);

unlock(p,b);

 };

program T2

{

test(this, [type==People]);

test(this, [type=Helicopter]);

h=findlocknode([type==Helicopter]);

p=findlocknode([type==People]);

jump([NOT(p.heathy==false)], end);

ph=bond(p, h, [type==People_Helicopter]);

end: checkbondintegrity(ph);

unlock(p,h);
```

```
};

program T3

{

test(this, [type==People_Helicopter]);

test(this, [type==Hospital_People]);

ph=findlockbond([type==People_Helicopter]);

hp=findlockbond(type==Hospital_People);

p=findinternalnode(ph, [type==People]);

h=findinternalnode(hp, [type==Hospital]);

jump([NOT(numnode(hp.ps)<h.capacity)], end);

switch(p, hp);

end: checkbondintegrity(hp);

checkbondintegrity(ph);

unlock(hp, ph);

};

program T4
```

```
{

test(this, [type==People_Helicopter]);

test(this, [type==Hospital]);

ph=findlockbond([type==People_Helicopter]);

hospital=findlocknode([type==Hospital]);

p=findinternalnode(ph, [type==People]);

hp=bond(p, hospital, [type==People_Hospital]);

checkbondintegrity(ph);

checkbondintegrity(hp);

unlock(hospital, ph);

};

program T5

{

test(this, [type==Hospital_People]);

hp=findlockbond(type==Hospital_People);

p=findinternalnode(hp, [type==People]);
jump([NOT(p.healthy==true)],end);

leave(p);

end: checkbondintegrity(hp);

unlock(hp);

};
```

BIBLIOGRAPHY

[1] J. Macker and M. S. Corson, "Mobile ad hoc networking and the IETF," *ACM Mobile Computing and Communications Review*, vol. 2, no. 1, pp. 9–14, January 1998.

[2] C.-C. Chiang, H.-K. Wu, W. Liu, and M. Gerla, "Routing in clustered multihop, mobile wireless networks with fading channel," in *IEEE Singapore International Conference on Networks*, 1997, pp. 197–211.

[3] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan, "Energy-efficient communication protocol for wireless microsensor networks," in *Procs. Hawaii International Conference on System Sciences*, vol. 2, 2000, p. 10.

[4] L. Zhou and Z. J. Haas, "Securing ad hoc networks," *IEEE Network*, vol. 13, no. 6, pp. 24–30, 1999.

[5] R. Grimm, J. Davis, B. Hendrickson, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Wetherall, "Systems directions for pervasive computing," in *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, 2001, p. 147.

[6] Sun, "Java remote method invocation," *http://java.sun.com/j2se/1.4.2/docs/guide/rmi/*, 2007.

[7] L. Yang, Z. Dang, and O. H. Ibarra, "Bond computing systems: a biologically inspired and high-level dynamics model for pervasive computing," in *Unconventional Computation*, 2007.

[8] G. Paun, "Computing with membranes," *Journal of Computer and System Sciences*, vol. 61, no. 1, p. 108C143, 2000.

[9] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan, "An application-specific protocol architecture for wireless microsensor networks," *IEEE Trans. Wireless Comm.*, vol. 1, no. 4, pp. 660–667, Oct 2002.

[10] S. Basagni, "Distributed clustering for ad hoc networks," in *IEEE Fourth International Symposium on Parallel Architectures, Algorithms, and Networks*, 1999, pp. 310–315.

[11] M. Chatterjee, S. Das, and D. Turgut, “WCA: A weighted clustering algorithm for mobile ad hoc networks,” *Journal of Cluster Computing (Special Issue on Mobile Ad hoc Networks)*, vol. 5, no. 2, pp. 193–204, April 2002.

[12] S. K. Dhurandher and G. V. Singh, “Weight based adaptive clustering in wireless ad hoc networks,” in *IEEE International Conference on Personal Wireless Communications*, Jan 2005, pp. 95 – 100.

[13] O. Younis and S. Fahmy, “HEED: a hybrid, energy-efficient, distributed clustering approach for ad hoc sensor networks,” *IEEE Trans. on Mobile Computing*, vol. 3, pp. 366–79, 2004.

[14] A. Ephremides, J. Wieselthier, and D. Baker, “A design concept for reliable mobile radio network with frequency hopping signaling,” *Procs. IEEE*, vol. 75, pp. 56–73, 1987.

[15] I. I. Er and W. K. G. Seah, “Mobility-based d-hop clustering algorithm for mobile ad hoc networks,” in *IEEE WCNC*, vol. 4, 2004, pp. 2359–2364.

[16] S. Sivavakeesar, G. Pavlou, and A. Liotta, “Stable clustering through mobility prediction for large-scale multihop intelligent ad hoc networks,” in *IEEE WCNC*, vol. 3, 2004, pp. 1488–93.

[17] K. Xu and M. Gerla, “A heterogeneous routing protocol based on a new stable clustering scheme,” in *Proc. IEEE MILCOM*, 2002, pp. 838–43.

[18] P. Basu, N. Khan, and T. D. C. Little, “Mobility based metric for clustering in mobile ad hoc networks,” in *Workshop on Distributed Computing Systems*, 2001, pp. 413–418.

[19] S. B. Lee and A. T. Campbell, “HMP: Hotspot mitigation protocol for mobile ad hoc networks,” in *IEEE/IFIP International Workshop on Quality of Service*, 2003.

[20] C. Liu, K. Wu, and J. Pei, “A dynamic clustering and scheduling approach to energy saving in data collection from wireless sensor networks,” in *IEEE SECON*, 2005, pp. 374–385.

[21] C. F. Chiasserini, I. Chlamtac, P. Monti, and A. Nucci, “Energy efficient design of wireless ad hoc networks,” in *Proceedings of European Wireless*, 2002.

[22] M. Yang, J. Wang, Z. Gao, Y. Jiang, and Y. Kim, “Coordinated robust routing by dual cluster heads in layered wireless sensor networks,” in *International Symposium on Parallel Architectures,Algorithms and Networks*, 2005, pp. 454–459.

[23] C. T. Ee and R. Bajcsy, “Congestion control and fairness for many-to-one routing in sensor networks,” in *ACM SenSys*, 2004, pp. 148–161.

[24] M. X. Gong, S. F. Midkiff, and R. M. Buehrer, “A self-organized clustering algorithm for uwb ad hoc networks,” in *IEEE Wireless Communications and Networking Conference*, vol. 3, 2004, pp. 1806–11.

[25] A. D. Amis, R. Prakash, T. H. P. Vuong, and D. T. Huynh., “Max-min d-cluster formation in wireless ad hoc networks,” in *Procs. IEEE INFOCOM*, vol. 1, Mar 2000, pp. 32–41.

[26] R. Krishnan and D. Starobinski, “Efficient clustering algorithms for self-organizing wireless sensor networks,” in *Journal of Ad-Hoc Networks*, vol. 4, no. 1, Jan 2006, pp. 36–59.

[27] S. Bandyopadhyay and E. J. Coyle, “An energy efficient hierarchical clustering algorithm for wireless sensor networks,” in *IEEE INFOCOM*, vol. 3, 2003, pp. 1713 – 1723.

[28] T. Ohta, S. Inoue, Y. Kakuda, and K. Ishida, “An adaptive maintenance of hierarchical structure in ad hoc networks and its evaluation,” in *22nd International Conference on Distributed Computing Systems Workshops*, 2002, pp. 7–13.

[29] A. Manjeshwar and D. P. Agrawal, “TEEN: a routing protocol for enhanced efficiency in wireless sensor networks,” in *IEEE IPDPS*, 2001, pp. 2009 – 2015.

[30] R. Krishnan and D. Starobinski, “Efficient clustering algorithms for self-organizing wireless sensor networks,” in *Journal of Ad-Hoc Networks*, vol. 4, no. 1, 2006, pp. 36–59.

[31] S. Karmakar and A. Gupta, “Bounded clustering with low node-clusterhead separation in wireless sensor networks,” in *IEEE International Symposium on Parallel Architectures, Algorithms and Networks*, 2005, pp. 268 – 273.

[32] A. Durresi and V. Paruchuri, “Adaptive clustering protocol for sensor networks,” in *IEEE Conference on Aerospace*, 2005, pp. 1–8.

[33] C. Li, M. Ye, G. Chen, and J. Wu, “An energy-efficient unequal clustering mechanism for wireless sensor networks,” in *IEEE International Conference on Mobile Ad Hoc and Sensor Systems*, no. 7, 2005, pp. 597 – 604.

[34] G. Venkataraman, S. Emmanuel, and S. Thambipillai, “DASCA: A degree and size based clustering approach for wireless sensor networks,” in *IEEE International Symposium on Wireless Communication Systems*, 2005, pp. 508 – 512.

[35] R. Rajagopalan, “Topology control and routing in ad hoc networks: a survey,” *ACM SIGACT News*, vol. 33, no. 2, pp. 60–73, 2002.

[36] S. Kutten and D. Peleg, “Fast distributed construction of small k-dominating sets and applications,” *Journal of Algorithms*, no. 28, pp. 40–66, 1998.

[37] L. D. Penso and V. C. Barbosa, “A distributed algorithm to find k-dominating sets,” *Discrete Applied Mathematics*, no. 141, pp. 243–253, 2004.

[38] Y. Fernandess and D. Malkhi, “K-clustering in wireless ad hoc networks,” in *The second ACM international workshop on Principles of mobile computing*, October 2002, pp. 31–37.

[39] F. Dai and J. Wu, “On constructing k-connected k-dominating set in wireless networks,” in *IEEE IPDPS*, April 2005, p. 81a.

[40] S. Dhar, M. Q. Rieck, and S. Pai, “On shortest path routing schemes for wireless ad hoc networks,” in *International Conference on High Performance Computing*, 2003, pp. 130–141.

[41] D. P. Dubhashi, A. Mei, A. Panconesi, J. Radhakrishnan, and A. Srinivasan, “Fast distributed algorithms for (weakly) connected dominating sets and linear-size skeletons,” *Journal of Computer and System Sciences*, vol. 71, pp. 467–479, 2005.

[42] J. Wu and H. Li, “On calculating connected dominating set for efficient routing in ad hoc wireless networks,” in *3rd ACM International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, 1999, pp. 7–14.

[43] http://www.isi.edu/nsnam/ns/: NS-2 Network Simulator.

[44] D. Johnson and D. Maltz, “Dynamic source routing in ad hoc wireless networks,” in *Mobile Computing*, T. Imielinski and H. Korth, Eds. Kluwer Academic Publishers, 1996, pp. 153–181.

[45] C. E. Perkins and E. M. Royer, “Ad-hoc on-demand distance vector routing,” in *2nd IEEE Workshop on Mobile Computing Systems and Applications*, February 1999, pp. 90 – 100.

[46] R. de Renesse, M. Ghassemian, V. Friderikos, and A. H. Aghvami, “Adaptive admission control for ad hoc and sensor networks providing quality of service,” Center for Telecommunications Research, Kings College London, UK, Tech. Rep., 2005.

[47] C.-Y. Wan, S. B. Eisenman, and A. T. Campbell, “CODA: congestion detection and avoidance in sensor networks,” in *ACM SenSys*, 2003.

[48] C. Bettstetter and J. Eberspacher, “Hop distances in homogeneous ad hoc networks,” in *IEEE Vehicular Technology Conference*, vol. 4, 2003, pp. 2286–2290.

[49] D. West, *Introduction to Graph Theory*, 2nd ed. Prentice Hall, 2001, ch. 2, p. 72.

[50] F. Buckley and F. Harary, *Distance in Graphs*. Addison-Wesley, 1990.

[51] P. Zimmer, “A calculus for context-awareness,” BRICS Research Series, Tech. Rep., 2005.

[52] O. M. Group, *The Common Object Request Broker: Architecture and Specification, revision 2.3.1*, 1999.

[53] M. Weiser, “The computer for the 21st century,” *Sci. American*, September 1991.

[54] U. Saif, H. Pham, J. M. Paluska, J. Waterman, C. Terman, and S. Ward, “A case for goal-oriented programming semantics,” in *Fifth Annual Conference on Ubiquitous Computing, Workshop on System Support for Ubiquitous Computing (UbiSys)*, 2003.

[55] L. Kagal, T. Finin, and J. Anupam, “A policy language for a pervasive computing environment,” in *Proc. of IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, 2003, pp. 63–74.

[56] J. Waldo, “The jini architecture for network-centric computing,” *Communications of the ACM*, vol. 42, no. 7, pp. 76–82, 1999.

[57] T. Sivaharan, G. Blair, and G. Coulson, “Green: A configurable and re-configurable publish-subscribe middleware for pervasive computing,” in *Proc. of DOA 2005*, 2005.

[58] H. A. de O. Coelho, R. de O. Anido, and R. Drummond, “Quickframe-a fast development tool for mobile applications,” in *Innovations in Information Technology*, 2006, pp. 1–5.

[59] M. Roman, C. K. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt, “Gaia: A middleware infrastructure to enable active spaces,” *IEEE Pervasive Computing*, pp. 74–83, October 2002.

[60] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. D. Mickunas, “Olympus: A high-level programming model for pervasive computing environments,” in *IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2005, pp. 7– 16.

[61] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste, “Project aura: Towards distraction-free pervasive computing,” *IEEE Pervasive Computing*, April-June 2002.

[62] G. Chen and D. Kotz, “Context-sensitive resource discovery,” in *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications(PerCom)*, 2003, pp. 243 – 252.

[63] A. Nandan, S. Das, G. Pau, M. Gerla, and M. Sanadidi, “Co-operative downloading in vehicular ad-hocwireless networks,” in *Proc. of Second Annual Wireless On-Demand Network Systems and Services(WONS 2005)*, 2005.

[64] G. V. Chockler, I. Keidar, and R. Vitenberg, “Group communication specifications: a comprehensive study,” *ACM Computing Surveys (CSUR)*, vol. 33, no. 4, pp. 427 – 469, 2001.

[65] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, 1978.

[66] G. Chockler, N. Huleihel, I. Keidar, and D. Dolev, “Multimedia multicast transport service for groupware,” in *TINA conference on the convergence of telecommunications and distributed computing technologies*, September 1996.

[67] F. Cristian, “Reading agreement on processor group membership in synchronous distributed systems,” *Distributed computing*, vol. 4, no. 4, pp. 175–187, April 1991.

[68] F. Cristian and F. Schmuck, “Agreeing on processor group membership in asynchronous distributed systems,” Department of Cmputing Science and Engineering, University of California at San Diego, Tech. Rep. 95-428, 1995.

[69] M. Fischer, N. Lynch, and M. Paterson, “Impossibility of distributed consensus with one faulty process,” *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, April 1985.

[70] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, “On the impossibility of group membership,” in *Proc. of the fifteenth annual ACM symposium on Principles of distributed computing*, 1996, pp. 322–330.

[71] K. P. Birman and R. van Renesse, Eds., *Reliable Distributed Computing with the Isis Toolkit*. Los Alamitos: IEEE Computer Society Press, 1994.

[72] Y. Amir, D. Dolev, and S. K. ad Dalia Malki, “Transis: A communication subsystem for high availability,” in *Proc. of the Twenty-Second International Sysmposium on Fault-Tolerant Computing*, 1992, pp. 76–84.

[73] Z. Ge, D. R. Figueiredo, S. Jaiswal, J. Kurose, and D. Towsley, “Modeling peer-peer file sharing systems,” in *IEEE INFOCOM*, vol. 3, 2003, pp. 2188 – 2198.

[74] “Napster protocol specification,” http://opennap.sourceforge.net/napster.txt, March 2001.

[75] Clip2, “The gnutella protocol specification v0.4,” http://www.clip2.com/GnutellaProtocol04.pdf, 2000.

[76] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” in *Proc. of ACM SIGCOMM*, August 2001.

[77] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, “A scalable content-addressable network,” in *Proc. of ACM SIGCOMM*, August 2001.

[78] A. I. T. Rowstron and P. Druschel, “Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems,” in *Middleware*, 2001, pp. 329–350.

[79] M. Straber, J. Baumann, and F. Hohl, “Mole-a java based mobile agent system,” in *Special Issue Object-Oriented Programming: Workshop Reader of the 10th European Conf. Object-Oriented Programming*, 1996, pp. 327–334.

[80] J. Baumann and N. Radouniklis, “Agent groups for mobile agent systems,” in *Distributed Applications and Interoperable Systems*, K. G. H. Konig and T. Preus, Eds. London, UK: Chapman and Hall, 1997, pp. 74–85.

[81] I. Satoh, “Reusable mobile agents for cluster computing,” in *Proc. of IEEE International Conference on Cluster Computing*, 2003, pp. 270–279.

[82] J. Elson, L. Girod, and D. Estrin., “Fine-grained network time synchronization using reference broadcasts,” in *Proc. Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, vol. 36, 2002, p. 147C163.

[83] S. Palchaudhuri, A. K. Saha, and D. B. Johnson, “Adaptive clock synchronization in sensor networks,” in *3rd International Symposium on Information Processing in Sensor Networks (IPSN ’04)*, 2004.

[84] Q. Li and D. Rus, “Global clock synchronization in sensor networks,” in *IEEE INFOCOM*, May 2004.

[85] I. standards department, “Wireless lan medium access control (mac) and physical layer (phy) specitications,” *IEEE standard 802.11*, 1997.

[86] S. Deering and R. Hinden, *Internet Protocol, Versino 6 (IPv6), Request for Comments 2460*. Reston, VA: The Internet Society, December 1998.

[87] C. Perkins, Ed., *IP Mobility for IPv4 [memo], Requst for Comments 3344*. Reston, VA: The Internet Society, August 2002.

[88] B. Aboba, D. Thaler, and L. Esibov, “Linklocal multicast name resolution (LLMNR),” http:/tools.ietf.org/wg/dnsext/draft-ietf-dnsext-mdns/draft-ietf-dnsext-mdns-39.txt, March 2005.

[89] “Salutation arhitecture,” http://www.salutation.org.

[90] E. Guttman, “Service location protocol: automatic discovery of ip network services,” *IEEE Internet Comput.*, vol. 3, no. 4, pp. 71–80, 1999.

[91] “Common object services specification,” Volume 1, OMG Document number 94-1-1, March 1994.

[92] G. D. M. Serugendo, M. Muhugusa, and C. F. Tschudin3, “A survey of theories for mobile agents,” *World Wide Web*, vol. 1, no. 3, pp. 139–153, 1998.

[93] L. Cardelli, “A language with distributed scope,” *Computing Systems*, vol. 8, no. 1, pp. 27–59, 1995.

[94] C. Tschudin, “On the structuring of computer communications,” Ph.D. dissertation, University of Geneva, 1993.

[95] G. Berry and G. Boudol, “The chemical abstract machine,” *Theoretical Computer Science*, vol. 96, no. 1, pp. 217–248, 1992.

[96] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta, “Computation in networks of passively mobile finite-state sensors,” in *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, July 2006.