Spatial Coordination in Wireless Sensor Network Applications

A Thesis Submitted to the

College of Graduate Studies and Research

in Partial Fulfillment of the Requirements

for the degree of Master of Science

in the Department of Computer Science

University of Saskatchewan

Saskatoon

By

Anil Kumar Keela

Permission to Use

Abstract

In distributed systems, dependency among different computations of an application leads to a problem of deciding the locations of computations. Spatial requirements of a computation can be expressed in terms of spatial relationships with other computations. This research presents programming abstractions and language constructs which can be used for specifying spatial coordination requirements for distributed computations. A spatial coordination middleware has been implemented for satisfying spatial coordination requirements of systems implemented using the Actor model of concurrent computation. Our approach abstracts spatial requirements of concurrent computations and provides key programming primitives for specifying these requirements. We have also implemented a number of higher level spatial coordination primitives which can be translated into the basic primitives. Spatial requirements can be specified using these primitives and then the runtime system converts them into a constraint satisfaction problem and satisfies them. Our approach reduces the programming complexity and provides a middleware which separates spatial requirements from functional code and enables the application programmer to change spatial requirements at runtime without effecting application's functionality. We have identified some of the high level primitives and provided a mechanism to develop high level primitives on top of the basic primitives.

This thesis presents the rationale, design, implementation, and evaluation of spatial coordination. By comparing programs written with and without our spatial coordination primitives, we show how spatial coordination enables a programmer to specify spatial requirements declaratively and simplify the programming task. Experimental results demonstrate the performance of the approach, as the number of constraints increases.

Acknowledgements

I would like to take this opportunity to say thanks from the bottom of my heart to my supervisor, Dr. Nadeem Jamali. His guidance, support, encouragement and suggestions – as well as his flexibility in allowing me to adapt the scope and focus of this work in the face of challenges – were instrumental in completion of this thesis. I am grateful to him for accepting me under his supervision as well as for the financial support he provided me.

I would like to thank Professor Simone Ludwig, Professor Michael Horsch and Professor Ramakirshna Gokaraju for being my Thesis committee members. All of them made many important suggestions to improve the quality of this work.

I would like to thank my friend and colleague Venkat Aveen Reddy for his generous advice which was critical during the prototyping phase of this work. I would also like to thank members of the Agents Lab, Johnson Iyilade, Mehadi Hassan, Xinghui Zhao and Yue Zhang, for their support through the course of my studies. Mehadi and Jonhson spared much time listening to my practice presentations and offering comments.

I am grateful to the Department of Computer Science at the University of Saskatchewan for offering a friendly environment and partial funding during my graduate studies. Special thanks go to departmental administration staff, especially Ms. Jan Thompson, who coordinated and helped whenever needed.

I would like to express gratitude to my parents for their unconditional support and guidance through the challenges of graduate studies. Their emotional and financial support was critical in my successful completion of the program.

Finally, I am indebted to my lovely wife for her patience and encouragement. She always stood by me whenever I needed her. She helped me in improving my writing and presentation skills. She instilled confidence in me that I am capable of doing anything.

This thesis is dedicated to my wife Manorma Anil Keela

Contents

List of Tables

List of Figures

List of Abbreviations

WSN	Wireless Sensor Network
CSP	Constraint Satisfaction Problem
DCSP	Distribute Constraint Satisfaction Problem
APO	Asynchronous Partial Overly
B&B	Branch and Bound

Chapter 1

Introduction

This thesis addresses the problem of deciding where to place computations in a distributed execution environment in general, and in wireless sensor networks in particular. In distributed systems, computations are divided into sub-computations and distributed over a network of nodes to solve problems. These sub-computations are dependent on each other and have partial knowledge of the entire problem, which leads to a degree of uncertainty. A coordination mechanism manages these dependencies among distributed computations in order to reduce the uncertainty and achieve optimal results [10]. Coordination can be functional, temporal, spatial, and resource centered [15]. Functional coordination occurs when a process is dependent on results of other processes. Temporal coordination occurs when interdependent processes are supposed to be coordinated in time. Resource centered coordination occurs when resources are shared among multiple processes and each process competes for the resources to finish its task. Spatial coordination occurs when the location of a process needs to be coordinated with other processes in order to complete its task. This thesis relates to spatial coordination.

There are various applications which have spatial requirements. In particular, computations executing in wireless sensor networks (WSNs) increasingly have spatial requirements expecting computations to be executing at particular locations. In a WSN, application may require two processes to be together on the same node because they are inter-dependent. It may also require that two processes never reside on the same node because one is actively replicating the other for fault tolerance. It is hard to program spatial requirements due to limitations in programming languages. The closest way is either to hard code spatial requirements in the program or to adopt mobile agents. Mobile agents offer too low level an abstraction for programming these spatial requirements. This thesis proposes spatial coordination as an alternative. By spatial coordination, we mean representing spatial relationships between computations as relationships rather than as procedural instructions to be carried out. We want to be able to specify these relationships declaratively.

This chapter is organized in four sections: the first section identifies some scenarios which motivate this work, the second section describes the objectives of this research, the third section describes the contributions of this work, and the last section presents an outline of the thesis.

1.1 Motivation

Here, we present two examples to motivate spatial coordination.

- Consider a WSN for environment monitoring of a forest. In particular, this application monitors fire conditions of the forest. The fire occurrence in the forest is detected by measuring the variations in various parameters such as temperature, humidity, smoke, wind speed [31]. For example, when weather is dry (humidity) and hot (temperature), then there is a greater probability of fire [4]. The goal is to determine the location where the fire is expected to occur. The likelihood of fire at a location can only be accessed if both temperature and humidity data from the location are available. Let us assume that temperature and humidity monitoring tasks are carried out by temperature and humidity agents. The application requires that both temperature and humidity data should come from the same location. It also requires that data should come from all different locations of the network.

  From the above scenario, it is evident that there are two spatial requirements. Firstly, the temperature monitoring agent should be placed with the humidity monitoring agent so that both types of information are received from each location consistently. Secondly, two temperature or two humidity monitoring agents never need to reside on the same node; this satisfies a second requirement of the application to avoid redundant data transfer from any location.

- In case of computer networks, one way to provide fault tolerance is by actively replicating computations. If we consider this in the context of system made up of agents, these may be a data storing agent which stores data on a main server and another agent replicates the data storing agent for fault-tolerance. For effective fault-tolerance, the replicating agent should be executed on a separate server. In this case, we can observe a spatial requirement between the data storing agent and the replicating agent, which is that both of them cannot reside on the same machine.

From the above scenarios, spatial requirements in a variety of applications are evident. In current systems, there is no easy way to program these spatial requirements. This encouraged us to develop a middleware service to manage spatial coordination requirements.

1.2 Research Objectives

Current mobile agent systems provide mobility of agents; however, application programmers have to deal with the low level communication details in order to implement the mobility. The mobility and coordination among agents in mobile agent systems is performed through mixing mobility code into functional codes, some systems require list of predefined locations where an agent has to migrate to; other systems make the mobility of an agent dependent on relationships it has with other agents.

The objective of this research was to find a good abstraction for programming mobile computations. We are interested in providing a middleware service which satisfies the spatial requirements of applications and allows the programmer to write functional requirements of the application separately from its spatial requirements. Our approach has the following characteristics:

Easy to program: Our objective is to simplify the programmer's task. This is done by providing the high level programming constructs to the programmer. These high level constructs hide the low level details of mobility from the programmer. The programmer can specify spatial requirements in a declarative way using our programming constructs.

Modular and reusable: For software engineering reasons, we want our solution to be modular and reusable. We achieved this objective by providing a middleware service which satisfies the spatial requirements of applications without changing the functional code. The middleware service also allows programmers to use our programming constructs along with existing migration service provided by the system.

Identify Spatial requirements: In addition to basic constructs we have identified a number of high level spatial requirements which make our solution applicable for various distributed applications. Programmers can also develop their own high level constructs on top of the basic constructs provided by our middleware.

1.3 Contributions

We intend to provide a programmable mechanism for satisfying the spatial requirements in the distributed systems. Our approach abstracts spatial coordination requirements and provides basic spatial constructs. The spatial constructs allows the programmers to specify spatial coordination requirements declaratively. The programmer can also define his own high level constructs using basic constructs.

- We have identified spatial requirements in various applications and specified them using our constructs to illustrate their expressive power.
- We have implemented prototype of spatial coordination system in wireless sensor networks.
- We have separated functional concerns from spatial concerns.
- We have developed a middleware system to satisfy the spatial constraints.

1.4 Outline

The rest of the thesis is organized as follows: existing work in related areas is reviewed in Chapter 2, spatial coordination is described in Chapter 3, prototype design and implementation are described in Chapter 4, experimental results and evaluation of the work are described in Chapter 5, future work and conclusion are described in Chapter 6.

Chapter 2

Related Work

In this chapter, related work in mobility and spatial coordination is reviewed. Section 2.1 presents research in mobile agent systems and how they provide mobility. Section 2.2 provides the background of the wireless sensor networks. Section 2.3 presents relevant research in spatial coordination and applications of spatial coordination.

2.1 Mobile Agent Systems

In mobile agent systems, agents are autonomous entities which can decide the locations to visit and the function to execute at these locations. When an agent migrates from one location to another location it either carries code, data and execution state, which is known as strong migration or it carries code and data, which is known as weak migration. In strong migration, an agent can migrate at any point of its execution. This section identifies the primitives provided by the mobile agent systems for the mobility of agents. It classifies the mobility of agents into four classes: self migration, dynamic migration, reactive migration and iterative migration.

2.1.1 Self Migration

In self migration, once an agent is invoked it autonomously decides when to migrate. These systems [13, 3, 25] provide the basic migration primitives and they are either developed as a separate system or developed on top of the existing systems.

The D'Agent [13] mobile agent system was previously known as Agent TCL. The architecture of D'Agent is based on the server model of Telescript and it is divided into five layers: transport, server, generic core, interpreter and agent. The transport layer supports transport mechanism, the server layer manages local and incoming agents, the generic core layer provides function library of D'Agent, the interpreter layer provides execution environment for each supported agent language, and the agent layer contains agents themselves. The migration process of the agents is based on strong migration. The agent's migration can take place using two commands: *agent_ jump* and *fork*. The *agent_jump* command captures the internal state of the agent and sends an image to the destination node. The destination node restores agent's state and resumes execution immediately after *agent_jump* statement. The *fork* command allows an agent to clone itself at the

destination nodes.
Mole [3], which was developed in Java language [12], uses notions of agents and places. The agents are active objects that may have one or more than one thread, which can migrate from a location to another location. The places provide an execution environment for the agents and various services which are accessed by the agents. This system supports weak migration which transfers the data and the state of the agent. The migration is carried out by calling the *migrateTo()* method provided by the system. When the agent calls the *migrateTo()* method, all the threads belonging to that agent are suspended and then a system independent representation of the agent is created through serialization. Later it is sent to the destination node. On the destination node, the agent looks for the required classes and creates a thread to execute the agent's code.
The Ara [25] mobile agent platform supports secure execution of the mobile agents in heterogeneous networks. A mobile agent is a program in the system which can decide to move by its own choice. The system consists of a runtime system known as a core, which acts as a middleware between the operating system and the interpreter. The core provides services such as agent migration, access rights to the resources, and the service points for the agent's interaction etc. This system supports strong migration, which allows the agents to migrate at any point of their execution by calling the *ara_go* instruction.
The major trends observed in the above systems are: they provide basic agent mobility, they are modular, and support multiple languages. However, the disadvantages of these systems are that the application programmer has to deal with the low level agent migration: these systems mix up the migration code with the functional code of the agents which makes them difficult to use, and there is no way to control the movement of the agents once it is invoked.

2.1.2 Dynamic Migration

Dynamic migration introduces the concept of relative location of agents. Therefore, an agent changes its location dynamically with respect to the location of other related agents.
In Fargo [14], agents are represented as mobile codes, which are known as complets. A complet is like a module in a programming language with relocation capability. This system consists of two parts: application component and runtime system. The application component allows a programmer to write an application code whereas the runtime system consists of a distributed core object which provides mobility support and interconnects complets across machines.

Fargo introduces the concept of relative location of the complets known as co-location. The co-location can mean two complets on a same host or in a same local area network. The system provides a mechanism to bind one complet with other complets through complet references. It also abstracts some other relative references such as *Link, Pull, Duplicate, Stamp and BPull.* The *Link* reference simply links the source complet to the target complet and relocation of the source will not affect the target and vice versa. The *Pull* reference is a unidirectional reference, it binds the target complet to the source complet. So, if the source complet relocates then it pulls the target complet with it. The *Duplicate* reference is quite similar to the *Pull.*

The difference is that it creates a copy of the target and moves it to the locality of the source. In the *Stamp* reference, if the source relocates to some other locality and finds a local instance of the target there, then it connects with that instance. The *BPull* is also similar to *Pull* except the fact that it supports bi-directional pull and group re-location.

Multi-agent organization [28], presents a self-organization in mobile agents. It follows the concept of hierarchical organization used in Mobile Spaces [26]. In the hierarchical organization, a mobile agent can contain one or more than one mobile agents inside it and can migrate to other locations with mobile agents inside it. The mobile agents are organized in a hierarchal structure. The Parent agent has control over child agent and the child inherits parents' behaviors. The parent agent can move the child agent to other mobile agents or other computers. The system is composed of two parts: run time system and mobile agent. The runtime system is built on top of the Java Virtual Machine [20] and the mobile agents are developed using Java [12].

Multi-agent organization system provides dynamic organization of the agents by defining four migration policies: *follow, dispatch, fill and shift.* In the *follow* policy, if *agent A* writes follow policy for *agent B* then it implies that, as *agent A* moves to the other computer, then *agent B* will follow it. The *Dispatch* policy is quite similar to the follow policy except that a clone of *agent B* will follow *agent A*. In the *shift* policy, if *agent A* moves to the target location then *agent B* moves to the source location of *agent A*. The *Fill* policy is similar to the shift except that it moves a clone of *agent B* to the source location of *agent A*.

The above systems are better than the self migration systems because they provide high level primitives for the agent's mobility. The problem with these systems is that they deal only with the co-location of the agents and they are not considering separation of the agents.

VCEA [7], proposes an application consolidation engine which is based on two parts: the constraints management and the application placement. The purpose of this technique is to find a consolidation solution which minimizes the number of machines to place the applications and satisfy multiple constraints. The placement of the applications is dependent on the resource usage of the applications. The constraints are identified by observing the history of the resource usage and the application traces. The constraints are classified into three categories: *must be, should not be, and can be.* The *must be* represents the applications which should place on the same machine, the *should not be* represents the applications which should not be placed on the same machine. The *can be* has no preference of placement. The user can enter the constraints from the graphical user interface and then the system categorizes the constraints based on the requirements of the applications. Thereafter, the system performs the constraint fusion to provide a compact representation of the various constraints. The advantage of this technique is that it provides a way to identify and manage the constraints between the applications. It also finds a location for an application. However, this technique does not provide how constraints are specified and programmability aspect of these constraints .

2.1.3 Itinerary Based Migration

In itinerary based migration, a predefined list of locations and methods is provided to an agent. The agent visits each location and executes the method on that location.

One example of a system supporting itinerary based migration is Concordia [30]. Concordia was developed in Java and runs on JVM. Agents are Java objects which can migrate from one computer to another. The migration is invoked by the agents through calling the migrate method of the server, which handles the agent's migration. An agent carries its state, code, data and an itinerary. The itinerary is a data structure, stored in a separate location from the agent's data and execution state. The itinerary consists of a list of destinations where an agent has to go and a list of methods which it has to execute at each location. The agent can modify its itinerary at runtime. For mobility, the agent's state is serialized using the Java serialization object and transfered to the destination location.

Another itinerary based approach is presented in network management [27]. This approach uses hierarchical technique of Mobile space MobileSpace [26]; which allows a mobile agent to carry another mobile agent inside it. This approach consists of two layers: the first layer contains the application agents which carry a list of methods and an itinerary. The second layer contains the navigation agents, which provide navigation services to the application agents. The application agent provides its itinerary to the navigation agent and the navigation agent takes the application agent to the locations specified in the itinerary. The navigation agents are carrier agents, they know the network very well and provide navigation services to the application agents. The network keeps multiple navigator agents with the specific itineraries to give navigation services to the application agents for different locations. The navigation agent uses two methods: *moveToNext()* and *moveTo()* for migration. The *moveToNext()* takes next location from the itinerary and moves them there. The *moveTo()* takes the agent back to the starting location. The advantage of itinerary based systems is that they provide a controlled mobility of the agents, which is required in some applications where the itinerary is already known. The disadvantages of the above systems are: they provide a basic migration primitive and the programmers have to implement the low level itinerary details, and these systems only support controlled migration which makes the agents less autonomous and they can not decide themselves when to migrate.

2.1.4 Reactive Migration

Reactive migration takes place in real-time systems. In real-time systems, when an event takes place, agents migrate in reaction to that event.

Agilla [8] was developed for fire detection in forest, using a wireless sensor networks. Its middleware consists of he Agilla engine, which is responsible for the agent's migration, a tuple space for storing tuples, the context manager for storing the list of neighbors, and the reaction manager for storing reactions to the events. In the reaction manager, reactions are registered by the agents and they are fired when a related event occurs. For

example, if a fire tuple appears in the tuple space of a node then a fire tracking agent executing on the node clones and moves itself on other nodes to detect the perimeter of fire area. Agilla supports both weak and strong migration. It provides *move, clone smove and sclone* instructions for the agent migration. The move and clone are using weak migration and smove and sclone are using strong migration. The *move* instruction transfers the code of the agent to the destination location, where the agent resumes its execution from the beginning. The *smove* transfers data, program counter, and stack to the destination and the agent resumes its execution where it left off. The *clone* copies the data and the state of the agent on the destination node and resumes execution on both. The *sclone* copies data, program counter, and stack on the destination and resumes execution on both where it left off.
MARS [6] is a tuple space based mobile agent system. The tuple space is a shared memory model where every datum is a tuple. In this system, the tuple spaces are reactive; they can trigger a computational activity in the response to accesses, made to them by the agents. It does not implement its own mobile agent system. Indeed, it uses current mobile agent systems. The architecture of the MARS consists of mobile agents, an agent server, and the local MARS tuple space. The mobile agent server accepts incoming agents on a node and provides reference to the local tuple space. The tuple space is accessed through well-defined Linda [11] like primitives by the agents. Each tuple space can react to access made to it with differentiated behaviors. These systems introduce the potential reason for the agent's mobility. However these systems are limited to application level reactions. They do not consider resource or spatial reasons for the agent migration.

2.2 Wireless Sensor Network

A Wireless sensor network is composed of a number of resource constrained sensor nodes, which are capable of sensing the environment, processing the data, and communicating on wireless medium with their neighboring sensor nodes. Sensor nodes have very limited resources such as memory, processor power and battery power. Current advancements in the sensor node devices have increased the demand of the wireless sensor networks in many fields. Some of the application areas are: health monitoring, military surveillance, environmental monitoring, industrial monitoring, structural health monitoring, and home security. The basic architecture of sensor networks is given in Figure 2.1. The sensor nodes are scattered in the sensor field. They collect the data and route them to the sink through multi-hop communication architecture.

A *Sensor node* consists of sensors, battery board, and a mote. One sensor node can have multiple sensors (temperature, barometer, acoustic). A mote has a radio, processor and memory.
A *Sink* performs the task of gateway between the users and the sensor network. The sink collects data and disseminates multiple tasks in the form of queries into the sensor network.
Sensor field is an area of interest where sensor nodes are deployed to collect the data. Sinks can communicate with each other and they can communicate with users via the Internet or satellite.
The Battery resource is a primary resource for a sensor node because all other resources are dependent on

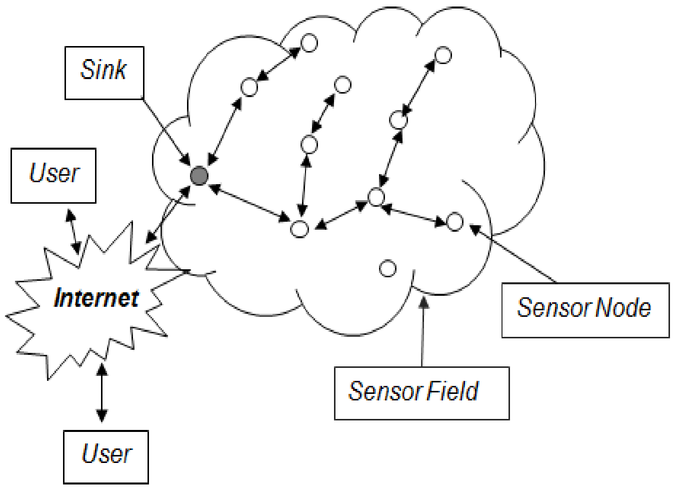

Figure 2.1: Single sink architecture of a WSN

battery power. Once a sensor node runs out of the battery power, it is dead. It is not feasible to recharge or replace the batteries of thousands of sensor nodes, deployed in the remote sensor field. Current techniques, whether it is routing technique or bandwidth congestion reduction technique or data aggregation technique, have to compromise with the battery utilization and other factors such as network performance, data accuracy, and data delay.

Table 2.1: Sleep states of the sensor nodes

Sleep state	Processor	Memory	Sensor	Radio
S_0	Active	Active	On	Tx, Rx
S_1	Idle	Sleep	On	Rx
S_2	Sleep	Sleep	On	Rx
S_3	Sleep	Sleep	On	off
S_4	Sleep	Sleep	Off	Off

There are various sleep states identified [29] based on power consumption and wake up time which are shown in Table 2.1. S_0 in table represents the working state of a sensor node. During the active state a sensor node keeps its processor, memory, sensors, and radio on. The states from S_1 to S_4 belong to different level of sleep states. S_1 is the lowest level sleep state and S_4 represents the deepest sleep state. The deeper the sleep state the less power it consumes and higher is the wakeup time overhead.

2.3 Applications of Spatial Coordination

In this section, we study spatial requirements of various applications and how these requirements can be translated into spatial coordination requirements.

2.3.1 Monitoring Applications

There are various types of monitoring applications. For example, environmental monitoring, habitat monitoring [22] or structural health monitoring [17]. These applications operate in two scenarios: event monitoring and long term periodic monitoring. During an event such as a fire condition in environmental monitoring, earthquake in structural health monitoring, and moment of animal in habitat monitoring requires real time data from the network. In the long term period monitoring, application continuously receives data from the network, after some specified time period.

Let us consider a structural health monitoring application which is a periodical monitoring application. In the structural health monitoring application, sensors are deployed on a bridge to monitor the response of the structure in different environmental conditions such as ambient load from wind or traffic over its life span. The application requires data from each location of the bridge to determine the response of the different portions of the bridge. This requirement can be satisfied in the spatial coordination by colocating an agent with a list of locations. For example, a data collecting agent colocates with the agent executing on node 1, collects data and then it colocates with the agent executing on node 2 and so on. At last, it comes back to the base station with the collected data. In this way, an application would not receive the redundant data and it does not have to maintain the wake up time of the nodes. In extreme events, higher priority should be given to the data of the nodes which are close to the event region. In events like fire, only nodes which are close to the fire should perform data collection process and other node should create a path for data transfer to the base station. This scenario can be satisfied in the spatial coordination by colocating the data collecting agent with the agents on the nodes which are close to the fire.

2.3.2 Network Management Applications

In WSN, applications running in a network can be updated through reprogramming of the nodes. Different techniques have been developed to address the network reprogramming problem [23, 18, 16]. TinyCubus [23] distributes the compiled code to reprogram the network. It also makes sure that code does not affect any other functionality, and reaches at the correct location. It only updates the nodes which satisfy the hardware constraints. Mate [18] transmits the scripts to reprogram the network. Incremental update approach [16] compares the difference between old and new code and transmits their difference to reprogram the node. These techniques, one or the other way, are sending code to reprogram the nodes. The new code is received by the node and then executed on it. There are situations when some nodes should be updated and others should not be updated. For example, multiple applications such as environment monitoring and fire detection are executing in a network. In this case, if a new version of the code comes for the fire detection application then nodes which carry the fire detection application should be updated. For the above reprogramming scenarios it can be observed that there exists spatial requirements between mobile code and nodes. For example, the new code can be carried by the mobile agents. These agents can also have a list of destinations to visit and execute the new version of the code on the nodes to update them. These mobile agents can also have a requirement that they cannot visit certain nodes to avoid the unnecessary updates on some of the nodes.

2.3.3 Spatial Applications

Smart message system is a space-aware programming model which provides a virtual name space over outdoor networks of embedded systems [5]. It allows resources to be referenced by their physical locations and properties. For example, “Hill1:camera[1]” represents *Hill1* as a virtual name space and *camera1* is a resource in the virtual name space. The virtual name space represents the location, where as the resource is a property contained by the location. The spatial programming is developed using a mobile agent like system known as smart messages. The smart messages consist of code, data, and execution state. In order to achieve the resources from a node smart messages are moved to the location using virtual name of the node. The advantage of this approach is that it hides the migration details, data access and services from the programmer and it encapsulates the physical resource into a virtual name space for accessing it. This application represents a spatial coordination requirement which is: the locations or service can be encapsulated within the agents and other agents can access the service.
SpatialViews [24] is another technique which uses virtual naming scheme. In SpatialViews, a *spatialview* is a collection of virtual nodes named by the services and locations. A programmer can define a *spatialview* of a group of nodes of interest and perform the operation on it. For example, a group may contain camera and image processing nodes because camera captures the image and image processing node processes the image to perform some search. This kind of scenario shows another spatial requirement which is: the group of related agents or computations should reside in the same place.

2.4 Summary

In this chapter, we focused on three issues: mobility, WSN and applications. In the mobility, we studied different mobile agent systems to determine how they provide the mobility, how they deal with the spatial aspect with the mobility, and the primitives these systems provide for mobility. In the WSN, we reviewed the background of the sensor networks, their resources and the limitations. In the applications, we tried to identify the various applications requirements which can be converted into the spatial coordination requirements.

Chapter 3

Spatial Coordination

This chapter develops programming constructs and presents the mechanism to specify an application's spatial coordination requirements declaratively. This chapter is organized in three sections: Section 3.1 presents the introduction, Section 3.2 describes the mechanism to construct the spatial coordination primitives and Section 3.3 presents the summary of the chapter.

3.1 Introduction

Communication requirements of applications in a distributed system can lead to a variety of spatial dependencies. However, there is no general way to manage these spatial dependencies. We specify these dependencies using spatial coordination requirements.

Spatial coordination separates the spatial concerns of an application from its functional concerns. Spatial coordination avoids the mixing of functional code and spatial coordination code, which enables the programmer to change spatial requirements at runtime without affecting its functionality. To program spatial coordination, a programmer writes the functional code of an application and then separately specifies the spatial requirements of the application declaratively. At run time, a middleware system monitors dynamically changing spatial requirements and ensures that they are satisfied.

3.2 Spatial Coordination Primitives

We have divided spatial primitives into two categories: basic primitives and high level primitives.

3.2.1 Basic Primitives

We refer to simple primitives involving only two actors as basic primitives. There are two basic primitives: colocate and separate.

Colocate

The colocate primitive specifies colocation constraint between two actors. The colocate constraint specifies the requirement to bring two actors on the same node, perhaps because they frequently communicate. If

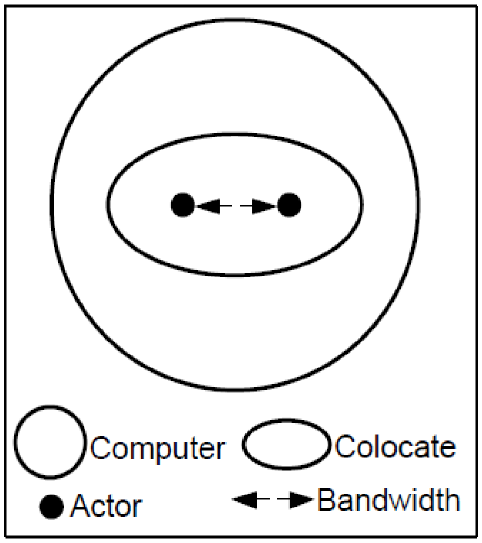

Figure 3.1: Colocate two actors on the same node

either of these actors attempts to relocate, the other must follow it. The colocate primitive ensures that the two actors are colocated so long as the constraint is in place. For example, in the fire monitoring application described in Chapter 1, the temperature and humidity actors should be on same node. This would be specified using a colocate constraint. If either of them changes its location (due to application or resource reasons) then the other is also moved to the same location by the spatial coordination middleware. If the temperature actor and the humidity actor had names 11 and 12, respectively, then the following code would specify the colocate requirement to keep them together.

```
Colocate(11, 12);
```

Colocate primitive can also be useful for network bandwidth management. It brings two actors close to each other by reducing the network distance between them, so that they can satisfy the communication requirement of an application and reduce the bandwidth utilization of the network. The notion of network distance is defined so that if network bandwidth is low then distance between two actors is large, creating an inverse relationship between the network distance and the bandwidth. As shown in Figure 3.1, when two actors are on the same node, the network distance between them is reduced and the bandwidth increased. When two actors are placed on the same computer then they can have the highest bandwidth.

Separate

The separate primitive specifies the separation constraint between two actors. The goal of the separate constraint is to specify the requirement of executing two actors on different nodes. For example, these actors may not be allowed to reside on the same node because of the requirements of a fault-tolerance or redundant data generation protocol. The separate primitive keeps two actors away from each other by increasing the network distance between them. The network distance can be increased by reducing the bandwidth between the two actors. As shown in Figure 3.2, two actors reside on the separate nodes and they are connected

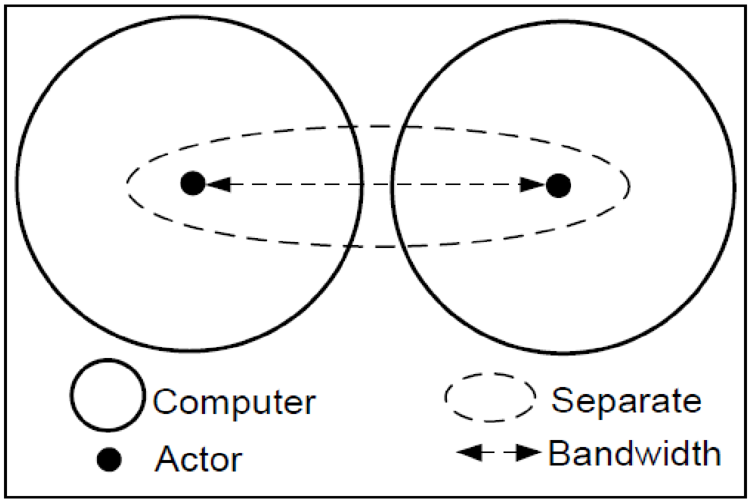

Figure 3.2: Separate two actors on the different node

through a network link. Recall, the active replication and fault-tolerance introduced in Chapter 1, where *dataStoringActor* is an actor executing on the main server, and *dataReplicationActor* actively replicates it. The fault-tolerance protocol requires that *dataStoringActor* and *dataReplicationActor* should always reside on different servers so that if the main server goes down, the *dataReplicationActor* can provide the service without any interruption. The following code shows how the separate requirements can be specified in an application program for two actors named 21 and 22.

```
Separate(21, 22);
```

3.2.2 High Level Primitives

We refer to primitives involving more than two actors or other primitives that can be represented in terms of basic primitives as high level primitives.

Group_Colocate

The group_colocate primitive allows us to specify that a group of actors to be colocated. When group_colocate is specified, the spatial coordination middleware breaks down the requirement into a number of basic colocate requirements for pairs of actors. For example, let us say that actors 11, 21, 31, and 41 need to be group_colocated as shown in Figure 3.3.

The middleware divides the group_colocate into basic pair-wise colocates (11,21),(11,31), and (11,41), because so long as these pair-wise requirements are satisfied, so will the group_colocate requirement. However, note that the two are not equivalent. If actors 11, 21, 31, and 41 are to be colocated, then the actual constraints would be each actor has colocate requirements with all other actors in the group, thereby giving *n(n-1)/2* pairs *(where n= number of actors)*. However, the resolution of the constraints simply requires each actor in the group to get colocated with one other actor in the group, creating n-1 pairs, as shown in Figure 3.4, to

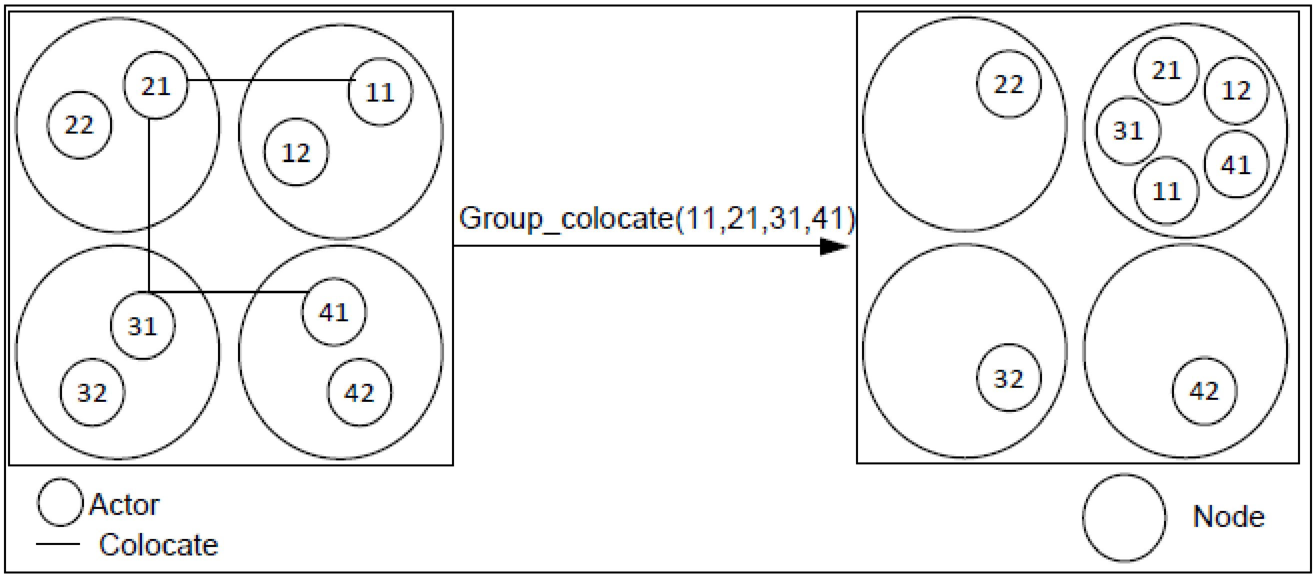

Figure 3.3: Actors before group_colocate and actors after group_colocate

create a minimally connected graphtree.

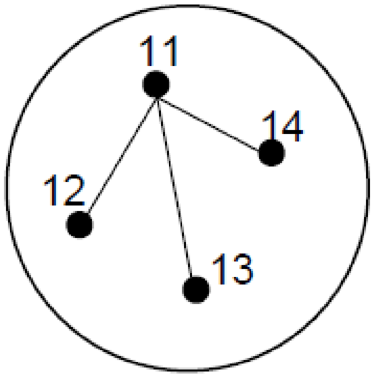

Figure 3.4: Colocate constraint pairs graph

So we can consider its pairs such as {(11,12),(11,13),(11,14)} or {(12,11),(12,13),(12,14)}. The resolution of the constraints and actual constraints are not equivalent to each other. A group_colocate declaration identifies the actors to be colocated as its parameters. The following code shows how the group_colocate can be specified in an application program for a group of actors named 11, 12, 13 and 14.

```
Group_Colocate(11, 12, 13, 14);
```

Group_Separate

The group_separate primitive allows us to specify that a group of actors to be separated. The group_separate primitive can be specified in the same way as group_colocate. When group_separate is specified, the spatial coordination middleware breaks down the requirement into a number of basic separate requirements for pairs of actors. For example, let us say that actors 11, 12, 13 and 14 need to be group_separated as shown in Figure 3.5. The middleware divides group_separate into basic pair-wise separates (11,12),(11,13),(11,14),(12,13),(12,14), (13,14). The difference between group_colocate and group_separate is that in group_separate, actual constraint pairs are considered whereas in group_colocate resolution of the constraint is considered. The reason behind using actual pairs is that each actor cannot reside with any other actor in the group, which results in *n(n-1)/2* pairs.

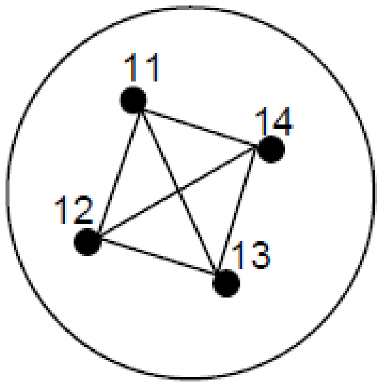

Figure 3.5: Separate constraint pairs graph

A group_separate declaration identifies the actors to be separated as its parameters. The following code shows how the group_separate can be specified in an application program for a group of actors named 11, 12, 13 and 14.

```
Group_Separate(11, 12, 13, 14);
```

3.2.3 Pattern Based High Level Primitives

In distributed systems, application programmers do not care about where a computation is physically located during execution; however, they care about its geographical location. The geographical location can be considered as a location from where data is collected, resources or services can be acquired in a network. The resources or services provided by a computer or a group of computers in a geographical location can be encapsulated into actors and these actors represent a geographical location or a service. Consider a computer network in which each node is encapsulated by an actor, which represents a node. If a network administrator has to install a chat client in the network, the administrator creates an actor of the chat client and colocates it with the list of actors which encapsulates the nodes in the network to perform the installation of the chat client.

Itinerary

Itinerary primitive enables an actor to migrate to different destinations, which are specified in the itinerary. The actor migrates to each of the destination; one by one, in a specified order. The itinerary primitive is built

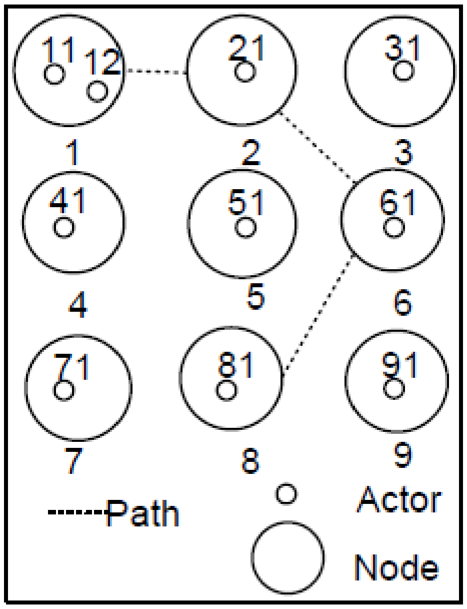

Figure 3.6: Actor 12 visits list of actors in a specified order

on top of the basic primitives. In order to convert the itinerary primitive into basic primitives, the destination nodes are encapsulated into actors and the actors represent the nodes in the network. In Figure 3.6, the number under the big circle represents the node id. The node is encapsulated by the actor inside it and represents the node in the network. For example, Actor 11 represent the Node 1. Therefore, if an application requires an actor to execute certain functionality on the list of locations then the programmer can specify the itinerary in the following way in an application program:

Itinerary (actor12,(destActor21, destActor61, destActor81));

In itinerary, the middleware takes the source actor and colocates it with the list of destination actors, specified in the itinerary. The following code shows a break down of the itinerary primitive into basic pair-wise colocate requirements. The middleware takes a destination from the list and inserts the colocate requirement between the source and the destination. Once the first requirement is satisfied then the middleware removes the first colocate requirement and then takes the next destination from the itinerary and inserts another colocate requirement. It keeps performing it until the itinerary is exhausted.

```
Colocate(actor12,  destActor21)
~Colocate(actor12,destActor21)
Colocate(actor12,destActor61)
~Colocate(actor12,destActor61)
Colocate(actor12,  destActor81)
~Colocate(actor12,  destActor81)
```

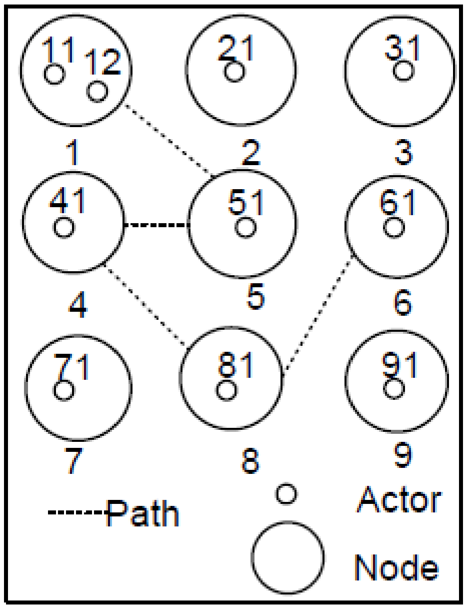

Figure 3.7: Actor 12 visits nodes in a random order

Visit

The *Visit* primitive is like the *Itinerary* primitive, but it can be useful when it is important that an actor visits a number of nodes once but not that it visits them in a particular order. Consider a case of data gathering in wireless sensor networks. The programmer injects a data gathering actor into the network to collect data from the sensor nodes. The data gathering actor collects data from the list of locations, in any order. The middleware makes sure that the actor has visited all the actors which are specified in the visit list. The data gathering actor can begin data collection from any of the actors in the list and then move to the next location. Figure 3.7 shows one example of the path that actor 12 could take. The visit primitive can be specified in the following way:

Visit(actor12, (destActor41, destActor81, destActor51, destActor61));

3.3 Summary

In this chapter, we presented the formalization of the spatial coordination. The application examples and formalization details of the constraints are presented. The list of constraints presented in this chapter, are by no means exhaustive; however, the programmers can build their own primitives. There is an overhead of using high level primitives, however they are necessary and yet useful in some scenarios. We allow the programmers to use the basic migration provided by the communication layer along with spatial coordination.

Chapter 4

Prototype Design and Implementation

This chapter describes the design and proof of concept prototype implementation of spatial coordination. The prototype is developed by simulating the Actor model [1] of concurrent computation.
The organization of this chapter is as follows: Section 4.1 briefly describes the Actor model, Section 4.2 presents the system design, Section 4.3 describes the implementation details of spatial coordination and includes formalization of spatial coordination as a constraint satisfaction problem. Section 4.4 presents the summary of the chapter.

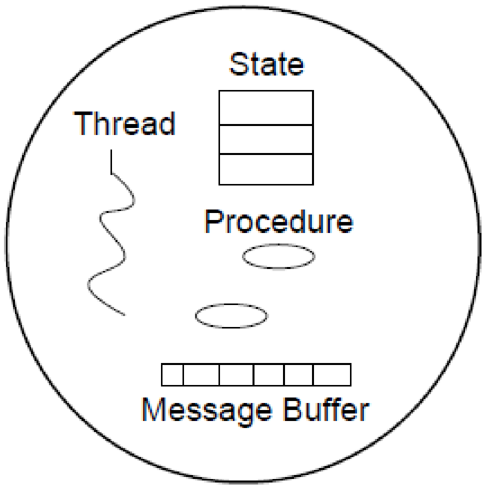

Figure 4.1: Actor

4.1 Actor Model

The Actor model is a flexible model for concurrent computation in distributed systems [1]. As shown in Figure 4.1, an actor can be viewed as an independent process, encapsulating a state, a number of procedures which can change the state, and a thread of control. Actors execute concurrently and communicate with each other through asynchronous message passing. Each actor has a globally unique name and a message queue for buffering the received messages. The messages received by an actor are processed in the order of their arrival. The Actor model defines three basic primitives of an actor:

- An actor can create new actors

- An actor can send messages to other actors whose addresses are known
- An actor can change its behavior

In our approach, we simulate actors and use actors to model primitive agents which carry out computations.

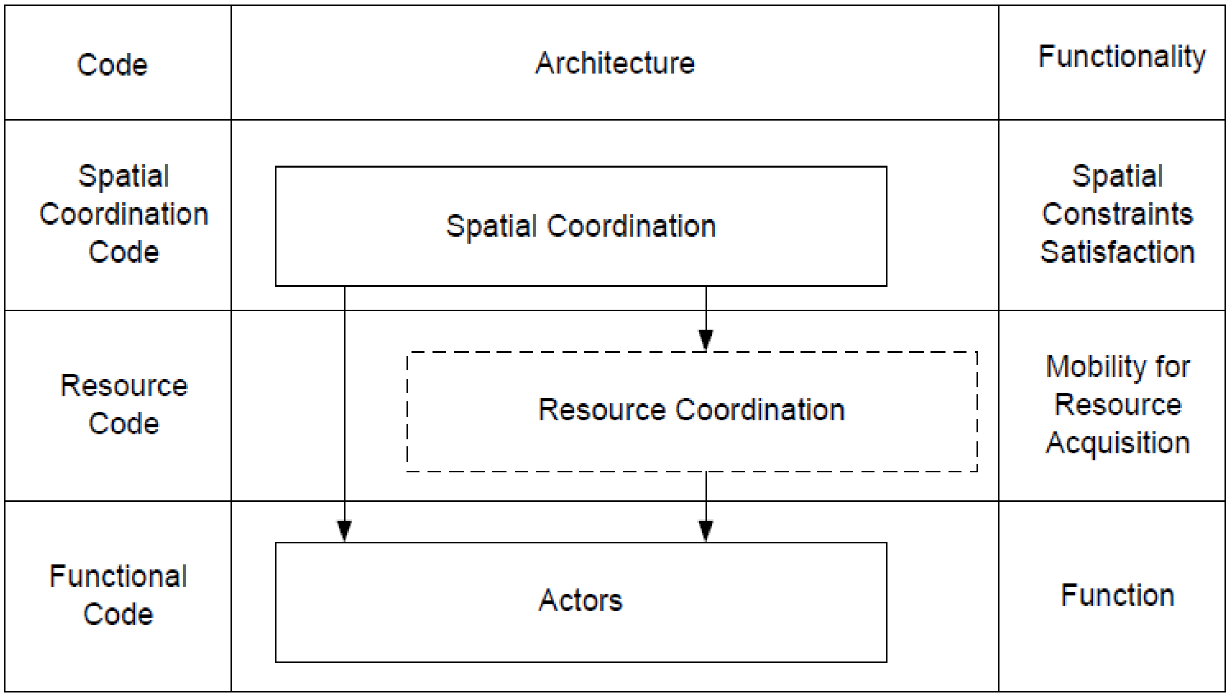

Figure 4.2: System architecture

4.2 System Design

In order to separate concerns, we design the system in a modular way. This system is divided into three layers: actor layer, resources layer, and spatial coordination layer; as shown in the Figure 4.2. The actor layer is where an application's functional code resides. It consists of three parts actor creations, specification of constraints and specification of application's functionality. The actor layer interacts with spatial coordination layer by using the spatial primitives provided by the spatial coordination layer.

The resource coordination layer is an optional layer, it maintains the availability of the resources such as memory, processor cycles, and battery power on a node. The battery power is a key resource in a sensor node because the life of a node depends on it [2]. This layer also provides the spatial requirements on the actors. For example, if node1 does not have a light sensor on it then it can specify the spatial requirement that *light_actor* should never come to node1 because it does not have the required resources.

The spatial coordination layer implements the spatial coordination middleware. It takes spatial requirements from the lower layers and provides the best solution which satisfies these requirements. This layer provides spatial coordination primitives to the other layers and hides implementation details from them. The

spatial coordination layer has four components: constraints assignment to the actors, colocate constraint satisfaction, separate constraint satisfaction, and actors migration. These four components work in a sequence, which reduces the complexity of the problem as the program executes. For example, the constraint assignment process assigns constraints to the actors and selects a head for the actors in a *Group_Colocate* or *Colocate* constraint, which simplifies the *Colocate* constraint satisfaction process. The *Colocate* constraint satisfaction brings actors on the same node and the head actor represents the group which reduces the number of actors participating in the separate constraint satisfaction.
Moreover, there is a coordinator node which basically acts as a barrier in order to synchronize the spatial coordination algorithm at different stages. The coordinator node only interacts with the spatial coordination layer, other layers do not know about it.

4.3 Implementation

This section covers the implementation of the actor layer and the spatial coordination layer and their details.

4.3.1 Actor Layer

The actor layer creates actors and defines the functionality of an application in the actors. Let us recall the example of the environment monitoring application in Chapter 1. If temperature monitoring task is carried out by actor *11* and humidity monitoring task is carried out by actor *12* then Program 1 shows the code of the actors. In our implementation, each actor has a unique name and it cannot be changed even if the actors migrate to any other node in the system. The name of an actor is composed of two parts: the first part represents the *node id* where the actor was created and the second part is the *id* of the actor which makes actor's name unique in the system. Multiple actors can also reside on one node. Once actors are created then constraints can be specified on the created actors through spatial coordination primitives, which are provided by the spatial coordination layer. Here, the assumption is that the constraints are specified only on the existing actors.

Actor Communication

Actors communicate with each other by message passing. There are two types of messages in the system: application and system messages. The application messages are exchanged between the actors for passing application level information. The system messages are used by the spatial coordination layer, to satisfy the spatial requirements. An actor can send a message to its neighbor actors with whom it has spatial relation.

Driver Function

The driver function allows the programmer to specify constraints on the created actors. The code in Program 2 explains the driver function, showing the name of the actors which currently exist in the system.

Program 1 Actors code

```
void Actors(id)
{
  switch(id)
  {
    //actor definition
    case:11
      TemperatureActor();
      break;
    case:12
      PressureActor();
      break;
  }
}
```

The constraints are specified on the existing actors using spatial coordination primitives. The *Spatial.init()* method triggers the spatial coordination layer to satisfy the constraints. The *Spatial.init()* is a call back function which means that when it finishes its execution, it informs the actor layer by calling the *Spatial.spatialDone()* event. The application's functionality executes only after the *Spatial.spatialDone()* event is received.

Program 2 Constraints specification in the actor layer

```
void Driver()
{
  //actors in the systems
  actors 11, 12, 21, 22, 31, 32;
  //Spatial constraints
  Group_Colocate(31, 12, 21);
  Separate(11, 31);
  //satisfy requirements
  call Spatial.init();
}

event result_t Spatial.spatialDone(result_t success)
{
// Execute Application's functionality
}
```

4.3.2 Spatial Coordination Layer

Spatial coordination layer handles the spatial requirements, which may come from the actor layer or the resource layer. It consists of a middleware, which satisfies the spatial constraints of the actors running on a node. It exports the *Spatial.init()* and spatial coordination primitives, which are used by the actor layer. Its operations are divided into three parts: actor's data structure creation and constraints assignment, colocate constraints satisfaction, and separate constraints satisfaction.

Actor_Structure and Constraints Assignment

The actor layer passes the total number of actors on the node information to the spatial coordination layer. The spatial coordination layer creates an actor_structure for each actor on the node. The actor_structure has following fields: *actorid, head, location, mediate, desire to mediate m, domains, constraint_list, good_list, and agent_view.* Each node initializes the actor_structure with initial values. Program 3 shows the initialization of the actor_structure.

Program 3 Actor structure initialization

```
procedure initProc
  actorid → actor_name
  headid →  group_head
  location → current node id
  mediate → FALSE
  m → TRUE
  priority → sizeof(constraint_list)
end procedure;
```

The process of constraints assignment takes place in two steps: firstly, breaking down the group wise constraints into basic pair-wise constraints, secondly assigning of constraints to the concerned actors.

- Step 1: the spatial coordination requirements received from the actor layer through basic or high level primitives are stored in a global constraint data structure of the node. The global constraint data structure has three fields: name of the actor one, name of the actor two, and the constraints between the actors. If spatial requirements are received from basic pair-wise primitives then constraint pairs are stored as it is in the global constraint data structure. If constraints are received from high level primitives then they are broken down into pair-wise constraints and stored in the global constraint data structure. As discussed in Chapter 3, if there are n actors in a *Group_Colocate* then they are broken down into *n-1* pair-wise *Colocate* constraints. For example, if constraint is *Group_Colocate(11, 21, 31)* then the system takes the first actor of the group as a head of the group which is actor 11 in this case and creates *Colocate(11, 21), Colocate(11, 31)* pairs. if there are n actors in a *Group_Separate* then they are broken down into *n(n-1)/2* pair-wise *Separate* constraints.

- Step 2: this step has two parts: the selection of the group head actor and the assignment of constraints to the actors. The head field of an actor_structure may contain any one of the following values: its own name, group head's name or empty. An actor can be a head in two situations either it is a first element in the *Colocate* constraint pair or it has only *Separate* constraints with other actors. For example, if the constraint is *Colocate(21, 22)* then 21 is considered as a head of the group. An actor has group head's name in its head field when it is a member of a group. The head field remains empty when actor does not have any constraint. In constraints assignment process, each actor executes the *InsertConstraints()* function which extracts constraint pairs from the global constraint structure and checks if the pair has its name in it then stores the pair into its constraint list.

Communication Protocol

The communication protocol uses messages of type system in the spatial coordination layer. There are two types of communication between actors: local communication and remote communication.

- The communication between the actors residing on the same node is known as local communication.
- The communication between the actors residing on the different nodes is known as remote communication.

In both, remote and local communication, a message send is handled by the *initSend()* method, as shown in Program 4. An actor creates a message packet, which consists of packet header and data part. The data is inserted into data part of the message packet then it is inserted into an out going message queue. The *initSend()* function takes that message from the queue and checks the destination location. If the destination location is the same as the current location then the message is sent by a function call, otherwise the message is transmitted through radio to the destination location. A *sendDone* event is fired when the message is transmitted to the destination node.

Program 4 Message send procedure

```
procedure initSend
   msg = NULL ;
   if msgQueue != empty do
       msg= get message from msgQueue;

   if msg.destination == node id do
         localReceive(msg);
   else
         send(msg) to msg.destination;

 end procedure;
```

For the message receive, there are two functions *localReceive()* and *SpatialReceive.receive()*. The *localReceive()* function receives the local incoming messages and finds the name of the recipient actor from the message and passes the message to it. The *SpatialReceive.receive()* receives the message from the source node and enqueues it in the received message queue. The *message()* method checks received message queue for any new message. If there is a new message in the received message queue then it takes the message and extracts the recipient's name from the message and passes it to the concerned actor.

Colocate Constraints Satisfaction

As discussed in Chapter 3, colocation of actors means to bring the *colocated* constraint actors on the same node. In *Colocate* constraint satisfaction, all the member actors of the group provide their constraints list to the head of the group and the head represents the group.

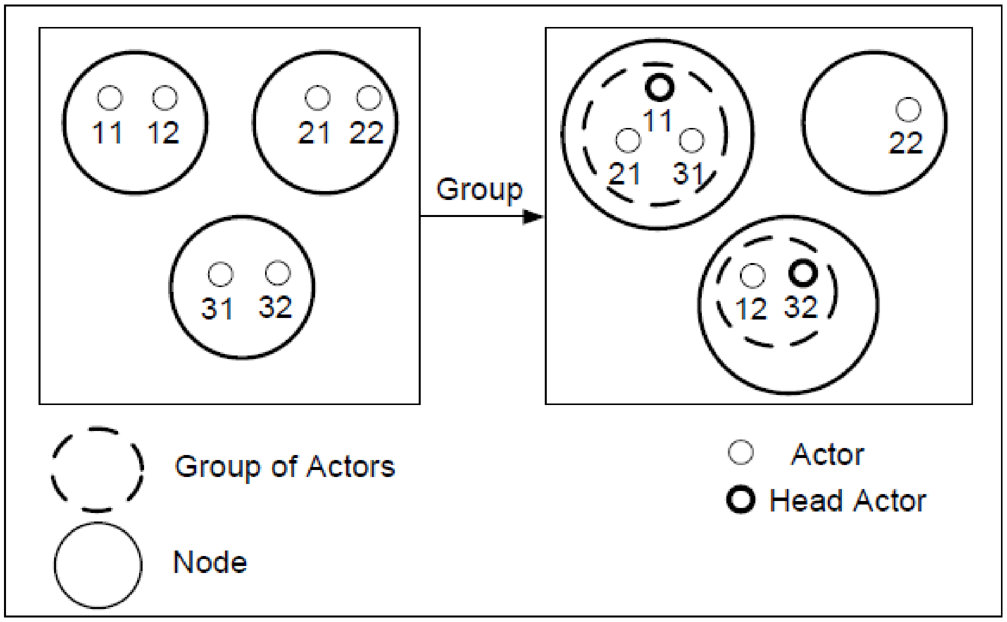

Figure 4.3: Actor grouping

Consider a scenario of colocation in Figure 4.3. There are three nodes and six actors in the system and colocate requirements are *Group_Colocate(11,21,31)* and *Colocate(32,12)*. In this scenario, there will be two groups: group 1 has three actors and actor 11 would be the head of the group, group 2 has two actors and actor 32 would be the head of the group. The entire group can be considered as one actor which is represented by the group head. Before going into details of the colocate constraints satisfaction, let us look at the messages which are required to satisfy the colocate constraints.

Messages

- MEMBER_UPDATE: is exchanged between the members of the group and their separate constraint neighbors. This message contains the name of the group head.
- HEAD_UPDATE: is sent to the head of the group by the member of the group. This message contains the constraints of the member actors.
- COL_DONE: is sent by the *AgentsColocateDone* method of the spatial coordination layer to the coordinator. This message informs the coordinator about the completion of colocate process of the actors on its node.

There are two steps to satisfy the colocate constraints: the first step involves the members of the group updates their separate constraint neighbors about their head. The second step involves the members of the group send their constraint list to the head of the group.

- In Step 1, each actor looks into the head field of its *actor_structure*. If it contains the name of the head then it implies that the actor is a member of a group, if it contains its own id then the actor is a head of the group. If it is a head then it waits for the members to send their constraint list. If an actor is a member of the group then it sends its group head name to the actors with whom it has a *separate* constraint using the MEMBER_UPDATE message and also adds the actor's name into its *init_list*, as shown in Program 5. On the receiving side, receiver finds the sender's name in its constraint list and replaces it with the sender's group head name. If the receiver actor has an empty head field then it sends an acknowledgment message to the sender which informs the sender that receiver itself is a head. When the sender receives replies from its separate constraint neighbors then it finds the name of the neighbor in its *init_list* and removes the name from the list. By doing this the actor confirms that all the separate neighbors have sent their head information.

- In Step 2, the member actors send their updated constraint list along with their initial information to their concerned head using the HEAD_UPDATE message. They also call the *AgentColocateDone()* method of the spatial coordination layer. The head receives the message, adds member information into its *agent_view* and waits until all the member actors have sent their initial information message. As soon as all the messages are received from neighbors then the head performs two tasks. Initially, it updates its constraint list by merging the member's constraint lists. During the merging process if the group head finds any conflicting requirement between actors then it sends a message to the coordinator node to stop the coordination process. Otherwise, it calls the *AgentColocateDone()* method of the spatial coordination layer. The *AgentColocateDone()* method checks whether all the actors residing on the node have finished the colocate process. If all the actors have not finished the colocate process then it waits for a half second and sends a message to the remaining actors to resend the COL_DONE message. Once all of the actors have finished colocate process then the node sends a COL_DONE message to the coordinator.

Coordinator Node

The purpose of the coordinator node is to synchronize the actors at different steps of the algorithm. When each node in the system completes its colocate process then it sends a SEND_DONE message to the coordinator. The coordinator collects all the COL_DONE messages and makes sure that all the nodes have finished the colocate process. As soon as all the COL_DONE messages are received from the nodes then the coordinator broadcasts a START_SEPARATE message. When the nodes receive the START_SEPARATE message then they move on to separate constraints satisfaction.

Program 5 Colocate procedure

```
//initialize actors
//assign constraints
  procedure member_update
     counter 0;
     for each $x_j$ constraint_list do
        if $x_j$.relation == SEPARATE do
           send(MEMBER_UPDATE($x_i$,$x_i$.head))to $x_j$;
           counter++;
        end do;
  end member_update;


  When receive(MEMBER_UPDATE($x_j$.$x_j$.head)) do
     update constraint_list with ($x_i$ with $x_j$.head);
     if $x_i$.head == Null do
         send(MEMBER_UPDATE($x_i$,Null)) to $x_j$;
         counter--;
     if counter == 0 do
         send(HEAD_UPDATE($x_i$,$x_i$.constraint_list, location, m, priority)) to $x_i$.head;
         AgentsColocateDone();
   end do;


   When receive(HEAD_UPDATE($x_j$,$x_j$.constraint_list)) do
      update constraint_list with ($x_j$,$x_j$.constraint_list);
      update agent_view with ($x_j$,$x_j$.constraint_list);
      if HEAD_UPDATE receive from all Colocate do
         AgentsColocateDone();
   end do;


   procedure AgentsColocateDone

      update agent_list with colocate done
         agent_list= colocate_Done;

      if agent_list has colocate done from all actors do
         Send(Col_Done(Coordinator));

   end AgentsColocateDone;
```

Separate Constraints

The last part of the algorithm is to solve the separate requirements. In the separate constraints satisfaction, only the head actors can participate. Before going into the details of the separate constraints, let us discuss what is a constraint satisfaction problem (CSP) and then we formalize our problem into CSP.
A Constraint Satisfaction Problem consists of the following:

- a set of n variables V= $\{x_1,.....,x_n\}$;
- discrete, finite domains for each of the variables D=$\{D_1, ...,D_n\}$;
- a set of constraints R =$\{R_1,...R_n\}$ where each $R_i(d_{i1},...,d_{ij})$ is a predicate on the Cartesian product $D_{i1} \times x .... \times x\ D_{ij}$ that returns true iff the value assignments of the variables satisfy the constraint.

The problem is to find an assignment A= $\{d_1....,d_n \mid d_i \epsilon D_i\}$ such that each of the constraints in R is satisfied. CSP has been shown to be NP-complete, making some form of the search a necessity.
In our case,

- a set of n variables are locations of the actors.
- discrete, finite domains are number of nodes in the network.
- a set of constraints are predicates on Cartesian product.

The identified problem is to find the location of the actors in such a way so that all of the constraints in the system are satisfied.
In order to solve this problem, we applied the Asynchronous Partial Overlay(APO) [21]. The *APO* is based on a cooperative mediation process. The agents in the algorithm construct a connection with other agents. Each agent tries to solve the subproblem within its own constraints. An agent performs the role of the mediator based on its priority. Once an agent becomes the mediator it tries to resolve the conflicts by changing its own value, if a conflict is resolved then it sends its new value to the connected agents. Otherwise this process continues until all the conflicts have been removed or there is no solution for the problem. The algorithm's worst-case runtime complexity, in domains that are exponential, is exponential. The APO algorithm is implemented in several steps. The messages used by the coordination layer of the node to communicate with each other which helps to find out the best solution, are explained below:

Messages

- INIT: is an initialization message which is sent by all the actors to their separate neighbors. It contains the initial information such as name, location, constraints, priority, and m flag.
- EVALUATE: is sent by the actor who wants to be the mediator of the session to its neighbors for their membership in the session.

- MEVALUATE: is sent to the session mediator by the members to inform about their membership in the session.
- WAIT: is sent by the neighbors which are currently involved in a session and cannot be part of any other session.
- ACCEPT: is sent by the mediator to the members and contains the future location of the actor.
- OK: is exchanged between the mediator and members, it contains the updated information of the actors.

APO Algorithm Implementation

The APO algorithm consists of following steps:

- Step 1: Each actor starts sending its initial information to its separate constraint neighbors using the INIT message. Each sender actor maintains the *init_list*, which records the name of the destination actors to which it has sent the messages. On the receive of the INIT message, the receiver actor adds the information of the sender into its *agent_view* and checks its *init_list* whether it has sent initial information to the sender or not. If the information has already been sent then it removes the name of the actor from the *init_list*. If the initial information has not been sent then it sends its initial information to the sender actor. Each time when an actor removes the name of the sender actor from its *init_list* then it checks whether the *init_list* is empty or not. If *init_list* is empty then it moves on to the next step otherwise waits for the remaining actors to send their initial information.
- Step 2: Each actor checks its *agent_view* for any conflict with its separate neighbors. If an actor finds any conflict with one or more than one neighbor then it checks if any higher priority agent is not mediating then the actor assumes the role of mediator.
- Step 3: Once an actor decides to be a mediator for the session then it sends an EVALUATE message to its separate neighbors. On receive, each receiving actor checks whether it is already in a session or expecting a higher priority actor to mediate. If there is no higher priority actor to mediate then the actor sends a MEVALUATE and sets its m flag true. If there is a higher priority actor or m flag is already true then it sends a WAIT message to the mediator. When the mediator receives the MEVALUATE message from the neighbors then it includes the neighbors into find solution process, otherwise it drops the actor from the current session.
- Step 4: The mediator uses Branch and Bound (B&B) [9] search algorithm to find the best solution. The mediator considers the list of constraints received from the neighbors, list of locations available in the system and sets bound N to 1. It runs the B&B algorithm which finds a perfect solution with N' < N constraint violations. If it cannot find the perfect solution then returns the N' which represents that there is no solution for the problem. Once it finds the best solution then it sends its neighbor's

future location through an ACCEPT message. On the receive, the neighbor changes its value of current location to the future location and session variable to false. Once again each actor checks its *agent_view* for any conflict, in case of any conflict Step 3 is repeated again. Once all the conflicts have been resolved then each actor calls the *AgentSeparateDone()* method, which maintains the information of how many actors on its node have finished their separate process. As soon as it receives separate_done from all the actors on the node then it informs the coordinator about completion of the separate process.

Coordinator Node

This time coordinator node waits for all the head actors to send a SEPARATE_DONE message to it. Once it receives the messages from all the head actors then it broadcasts the MIGRATE message. When actors receive the migrate message then they start migrating to their future locations.
Once migration is done then the spatial coordination layer calls the *Spatial_Done* event of the actor layer. The functionality of the application is executed thereafter.

Actor Migration

The actor migration is simulated using the system messages. The migration process is carried out in three steps:

- Source node sends a message of type MIGRATION to the destination node. The message includes actor information such as: name of the actor, name of the actor's group head, agent_view, constraints, priority, and desire to mediate flag.
- On the receive of the MIGRATION message, the destination node creates a new actor with the same name and assigns the values.
- Later, the source node removes the actor from its node.

4.4 Summary

In our implementation, actors are simulated as functions. We have implemented the algorithm in three steps: colocate constraints satisfaction, separate constraints satisfaction and actor migration. The APO algorithm is modified according to our program's need. The algorithm trades off between the correctness and the number of messages send and receive. The number of messages increases because each actor has to inform the coordinator it has completed a certain stage, which helps to synchronize the actors across the network. The number of messages depends on the number of actors participating in the spatial constraints satisfaction process.

Chapter 5

Experimental Results and Evaluation

5.1 Introduction

This chapter analyzes our approach to spatial coordination and the prototype implementation which was discussed in the previous chapter. Section 5.2 analyzes the programmability aspect by comparing the lines of code with and without spatial coordination primitives. Section 5.3 describes the comparison of agent's mobility in spatial coordination with the mobile agent systems reviewed in Chapter 2. Section 5.4 evaluates the performance of the prototype and interprets the result of the experiments carried out using simulated workload. The summary of the chapter is presented in Section 5.5.

5.2 Programmability Evaluation

The objective of spatial coordination was to make programming easier for programmers by hiding the low level details from them and providing them high level constructs. In the chapter on implementation, we have described the details of how these primitives have been implemented. These primitives achieve the objective of improved programmability. Let us re-consider a system back-up application which was discussed in Chapter 1. We are considering only 2 nodes and three actors, 11, 12, 21. The actors 11 and 12 cannot reside on the same node because actor 12 is replicating actor 11. Program 6 shows the lines of code required to write the application without using spatial coordination primitives and Program 7 represents the code using these primitives.

The code without spatial coordination requires a programmer to write the functional code of the agent, low level migration code and agent communication code. All these functions increase the programming task of a programmer. On the other hand, spatial coordination code divides the code mainly into two parts: spatial coordination requirements and actor definitions. In the requirement part, a programmer has to specify the spatial coordination requirements. In the actor definition part, functionality of the actors has to be defined. The *Spatial.spatial()* event is fired by the runtime system once the spatial requirements have been satisfied then actors can execute their functionality. The programmer has to write only two lines of code to perform the task with spatial coordination.

Program 6 Without spatial coordination

```
void BackUpApplication()
{
   //create node
   create_Agent();
}

void Agent()
{
   //actor’s spatial requirements (Separate, Colocate)
   Separate(A, B);
   //agent sends its initial information to neighbors
   SendInitialInfo();
   //agent receives initial information from neighbors
   ReceiveInitialInfo();

   //check for any constraint violations
   if(agent11.location == agent12.location)
   {
       //find new location to satisfy the constraints
       constraint_satisfaction_algorithm();

       //update neighbors with new location information
       update_neighbors();
   }

   //agent receives updated information from neighbors
   Receive_update_info()

   //migration of the agent
   migrate(agent, location);

   //functionality of the agent
   RelpicationAgent();

}
```

Program 7 With spatial coordination

```
void Driver()
{
   //actors in the systems
   actors A, B,.. ;

   //Spatial constraints
   Separate(A, B);

   //Start constraint satisfaction
   call Spatial.init();
}

//This method is called back once constraints satisfaction done
event result_t Spatial.spatial(result_t success)
{
   //Start Actor's runtime
   return SUCCESS;
}
```

5.3 Agent Migration Comparison

Table 5.1 shows the comparison between four approaches for agent migration - reviewed in the related work - in terms of migration, primitives, modularity and high level primitives. *Migration* refers to how programmers have to deal with the agent's mobility in that system. *Primitives* refers to the systems employing it - to provide ways to specify spatial requirements. *Modularity* refers to whether the system separates the functionality and mobility of agents. *High level primitives* refers to the systems incorporating the ways to create high level primitives using basic primitives. After reviewing all these systems we found out that self migration and spatial coordination have opposite features.In self migration, a programmer has to use migration to satisfy spatial requirements, whereas in spatial coordination a programmer does not have to use migration; spatial coordination provides some of the high level primitives and defines the way to create more primitives on top of the basic primitives. Spatial coordination also includes modularity, which self migration does not. Self migration and reactive migration systems have a slight difference in that reactive migration incorporates the concept of primitives while self migration does not. However neither has the concept of modularity nor high level primitives. Dynamic migration system, however, includes the primitives and modularity but does not provide how to develop high level primitives out of given primitives.

5.4 Prototype Performance Evaluation

We carried out a number of experiments to assess the performance and scalability of the implemented prototype. Section 5.4.1 describes the environment of these experiments. Section 5.4.2 discusses the experiment

Table 5.1: Comparison of spatial coordination with mobile agent systems

Type	Migration	Primitives	Modularity	High Level Primitives
$SelfMigration$	Yes	No	No	No
$DynamicMigration$	No	Yes	Yes	No
$ReactiveMigration$	Yes	Yes	No	No
$SpatialCoordination$	No	Yes	Yes	Yes

design and Section 5.4.2 discusses the experimental results.

5.4.1 Experiment Environment

Experiments were carried out using Windows XP running a TOSSIM 1.x simulator and TinyViz plug-in. The machine has a Core2 Duo processor and 2GB of memory. TOSSIM is a discrete event simulator which runs on TinyOS operating system [19]. It provides a scalable, high fidelity simulation, of a sensor network. TinyViz is a GUI tool to interact with the simulation. We are using it to measure the simulation time. The constraints on the actors have been selected in two ways: uniform and random. In uniform selection, each actor has the same number of constraints; for example, if a node has 10 actors named 11, 12, 13,..., 110 and each actor has 1 constraint then for *Separate* constraint pairs would be (11,12), (13,14), (15,16), (17,18), (19,110) and for *Colocate*, we choose constraints involving actors on different nodes then *Colocate* constraints would be (11,21), (12, 22),(13, 23), (14, 24) and so on. In random selection, we run a random number generator function to generate random numbers between 11 and 110. When selecting random constraints, the runtime system often found some conflicting constraints. For example, actor 12 has *Colocate* constraint with actor 14 and it also has a *Separate* constraint with actor 14. The runtime system also found some complex conflicting constraints. For example, actor 29 has *Colocate* constraint with actor 45, 64, but actor 45 has *Separate* constraint with actor 64. In case of conflicting constraints, the runtime system determines this conflicting requirements and stops its execution.

5.4.2 Experiment Design

The scalability of the system is determined by how well the system scales as the number of actors and the number of constraints increases.

1. We designed the first set of experiments to evaluate the performance as the number of uniformly generated *Separate* constraints vary between 100 and 600 to determine the number of constraints that could be handled by our system. We carried out experiments with 10 nodes and 100 actors, 20 nodes and 200 actors and 30 nodes and 300 actors in the system.

2. We carried out the second set of experiments to evaluate the performance as average number of uniformly generated *Colocate* constraints vary between 100 and 600 to determine the number of constraints

that could be handled by our system. We carried out experiments with 10 nodes 100 actors, 20 nodes 200 actors and 30 nodes 300 actors in the system.

3. The third set of experiments is designed to evaluate the performance as number of randomly generated *Separate* constraints vary between 20 and 70. This set of experiments also determines the affect on the time taken by the system to solve the constraints when the total number of constraints differs from the total number of violated constraints. We carried out experiments on 10 nodes and 100 actors.

4. The fourth set of experiments was carried out on randomly generated *Colocate* constraints to evaluate the performance as the number of constraints vary between 20 and 70. It also determines the time taken by the system to solve the constraints when the total number of constraints differs from the total number of violated constraints. The constraints on each actor are selected randomly. We kept the number of actors 100 and number of nodes 10 as constants.

5. The fifth set of experiments involves randomly generated *Separate* and *Colocate* constraints. We carried out experiments on 10 nodes and 100 actors to evaluate the performance as the number of constraints vary between 20 and 70.

6. Finally, we compared random colocate, random separate with randomly chosen constraints of both types to determine the overhead incurred by the both types as compared to single type of constraints.

5.4.3 Results Interpretation

1. **Time to Solve Constraints Satisfaction vs. Number of Separate Constraints**

   In this set of experiments, we determined the scalability of the system by simulating the time taken to solve the number of uniformly generated *Separate* constraints. There is only one coordinator node and we uniformly increased the number of constraints on each actor. Figure 5.1 shows the affect of increasing the number of constraints with 10 nodes and 100 actors, 20 nodes and 200 actors, and 30 nodes and 300 actors.

   We can see that when the number of constraints are increased then the time taken to solve *Separate* these constraints also increases linearly. However, the time to solve constraints is also affected by the location of the actors involved in the constraints because if the actors are on the same node then there is no use of radio which in result takes less time to solve the constraints. We can observe this behavior in 10 nodes 100 actors at constraints 100, 200 and 400; at 400 constraints, time taken to solve the constraints drops because the constraint actors reside on the same node, therefore no remote communication, no message collisions and no message resend between the constraint actors. That is why time taken to solve the constraints drops significantly.

2. **Time to Solve Constraints Satisfaction vs. Number of Colocate Constraints**

   In this set of experiments, we selected the uniformly generated *Colocate* constraints. We kept the same

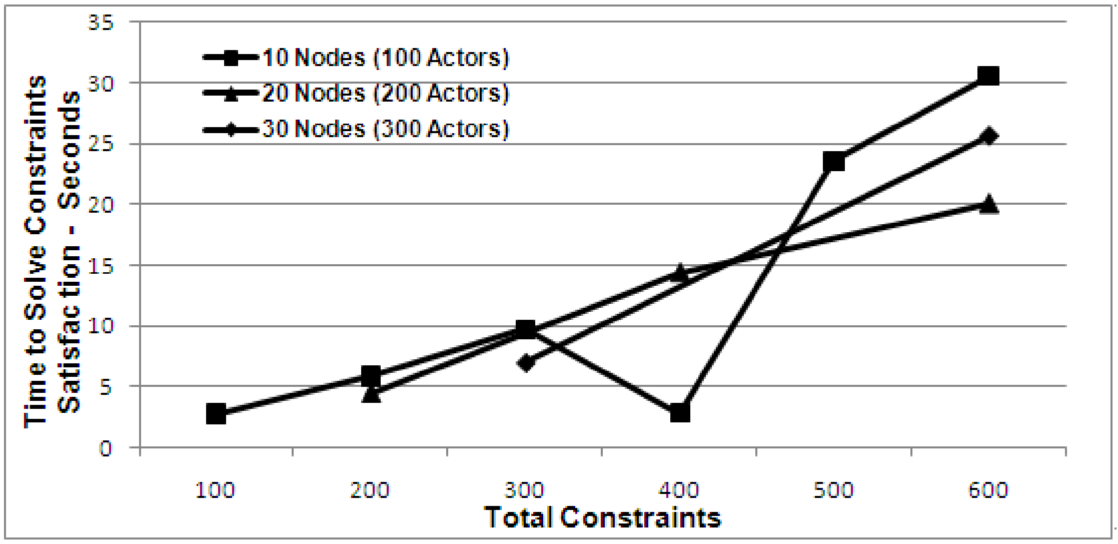

Figure 5.1: Time to solve constraints satisfaction of number of *Separate* constraints

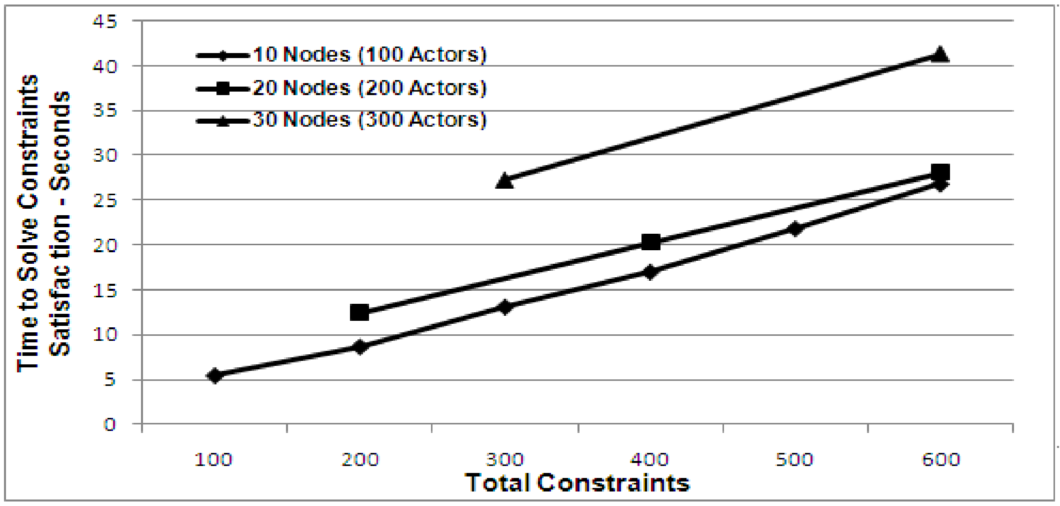

Figure 5.2: Time to solve constraints satisfaction of number of *Colocate* constraints

configuration as used in the previous experiments. Figure 5.2 shows the affect of increasing the number of constraints with 10 nodes and 100 actors, 20 nodes and 200 actors, and 30 nodes and 300 actors. We can see that when the number of constraints are increased then the time taken to solve the *Colocate* constraints increases linearly. Another observation in 30 nodes 300 actors is that it takes more time as compare to 10 nodes 100 actors and 20 nodes 200 actors. This behavior is caused because actors are on different nodes which results in network delay in communication between actors to solve the constraints.

3. **Time to Solve Constraints Satisfaction vs. Separate Random Constraints and Number of Constraint Violations**

   In this set of experiments, we are considering two parameters: the time taken to process the randomly

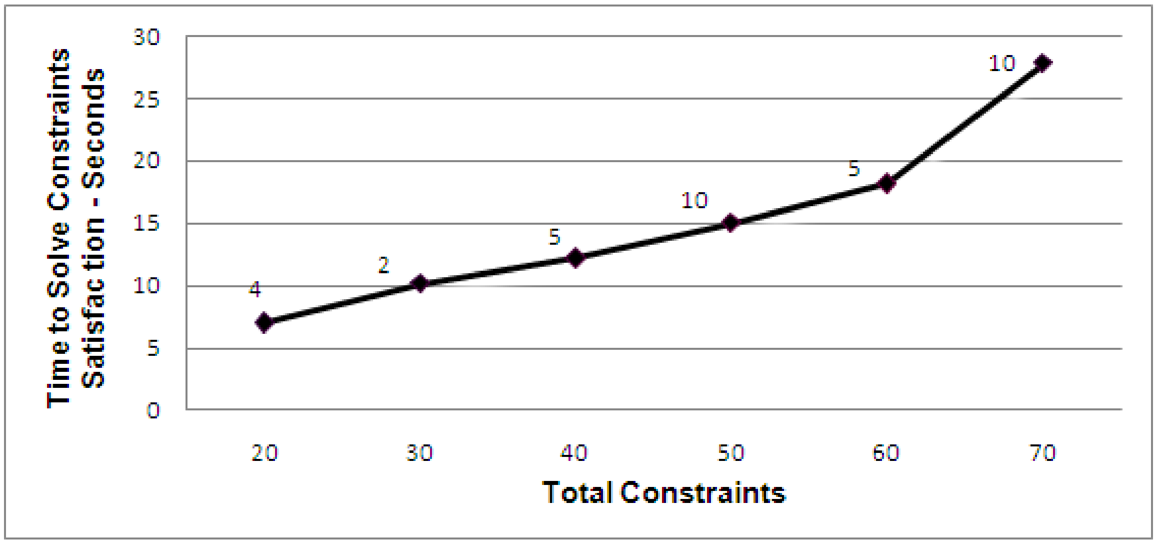

Figure 5.3: Time to solve constraints satisfaction of randomly generated *Separate* constraints

Table 5.2: Experiment data details of randomly generated *Separate* constraints

Participating Actors	Constraints	Time to Solve Constraints	Constraint Violations
25	20	7.10	4
28	30	10.24	2
33	40	12.20	5
36	50	15.06	10
38	60	18.23	5
40	70	27.89	10

chosen *Separate* constraints and the affect of the number of violated constraints. There is only one coordinator node and 100 actors, we varied the number of constraints. The number of nodes and the number of constraint sets both are 10 (see Appendix). Table 5.2 presents data used in the experiments. In Table 5.2, Participating Actors refers to number of actors involved in the constraint satisfaction process, Constraints refers to total number of constraints in the system, Time refers to time taken to solve these constraints, and Constraint Violations refers to number of violated constraints to be solved. For example, 25 actors had 20 constraints and there were 4 violations need to be solved and it took 7.10 sec to solve the constraints. Figure 5.3 shows the affect of increasing the number of constraints and the numbers on the top of the graph line represent the number of violations. We can observe that time is not affected by the number of violated constraints. Because at 40 constraints there are 10 violations, the system takes less time and at 70 constraints there also 10 violation but the system took more time. Hence the time taken to solve the constraints increase linearly as the total number of constraints increases.

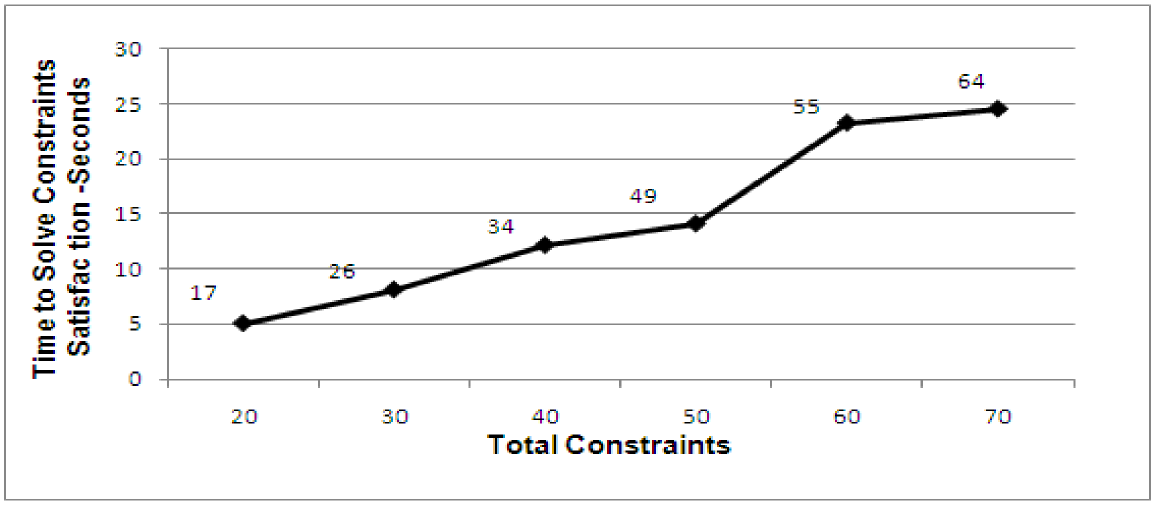

Figure 5.4: Time to solve constraints satisfaction of randomly generated *Colocate* constraints

Table 5.3: Experiment data details of randomly generated *Colocate* constraints

Participating Actors	Constraints	Time to Solve Constraints	Constraint Violations
30	20	5.01	17
39	30	8.07	26
49	40	12.17	34
58	50	14.07	49
63	60	23.21	55
80	70	24.48	64

4. **Time to Solve Constraints Satisfaction vs. Colocate Random Constraints and Number of Constraint Violations**

   We also conducted set of experiments on the randomly selected *Colocate* constraints. We kept the same configuration used in Experiment 3, only constraints have been changed. The selected constraints have been provided in Appendix. Figure 5.4 shows the time taken to the solve random constraints. The numbers at each point on the graph line representing the number of violations. The result shows as the number of constraints increases the time taken to the solve constraints increases linearly.

   Table 5.3 provides the detail of data used in the experiment and time taken to solve the constraints. For example, 30 actors had 20 constraints and there were 17 violations and the system took 5.01 sec to solve the constraints. The result suggests that the time to solve the *Colocate* constraint is also not affected by the number of violations. The time to solve constraints is proportional to the total number of constraints.

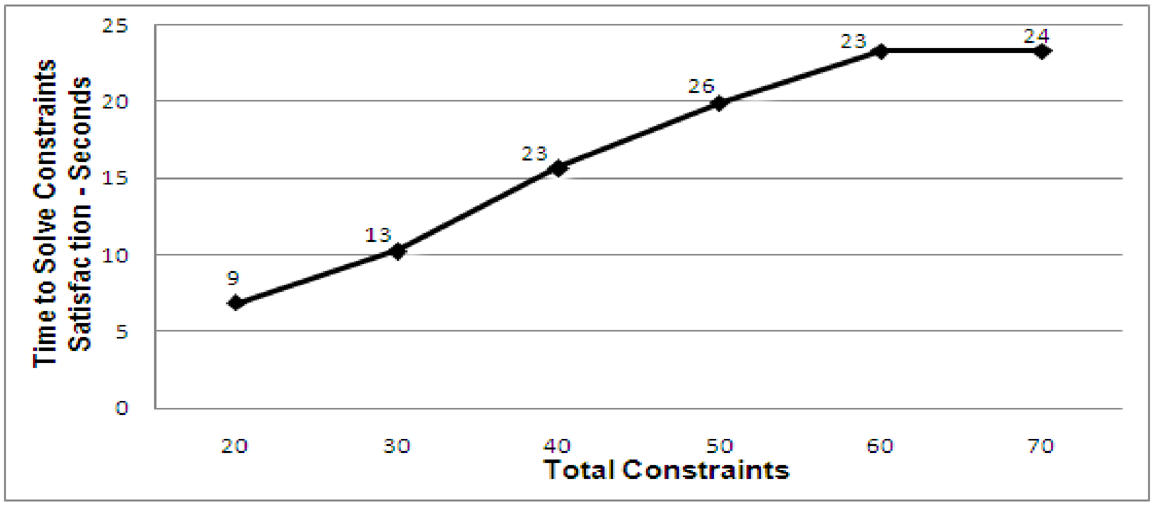

Figure 5.5: Time to solve constraints satisfaction of randomly generated *Colocate and Separate* constraints

Table 5.4: Experiment data details of randomly generated *Colocate and Separate* constraints

Participating Actors	Constraints	Time to Solve Constraints	Constraint Violations
24	20	6.95	9
30	30	10.29	13
41	40	15.71	23
43	50	19.95	27
45	60	25.51	25
47	70	23.43	24

5. **Time to Solve Constraints Satisfaction vs. Colocate and Separate**

This set of experiments is carried out with both random *Colocate* and random *Separate* constraints. In this set of experiments, we determined the performance of the system when *Colocate* and *Separate* constraints are mixed. The experimental setup kept same as used in previous experiments, we increased the number of constraints between 20 and 70. Table 5.4 shows the experimental data and the time taken to solve the constraints. Figure 5.5 shows the affect of increasing the number of constraints. We can see that the time taken to solve the constraints is not affected by the number of violated constraints. The time to solve constraints is affected by two factors: the number of constraints and how many actors are part of the *Colocate* group. As group size increases, communication between actors also increases which affects the time taken to solve the constraints. This can be observed at 60 constraints and 70 constraints. The constraint solving time of the both is same but the number of constraints is different.

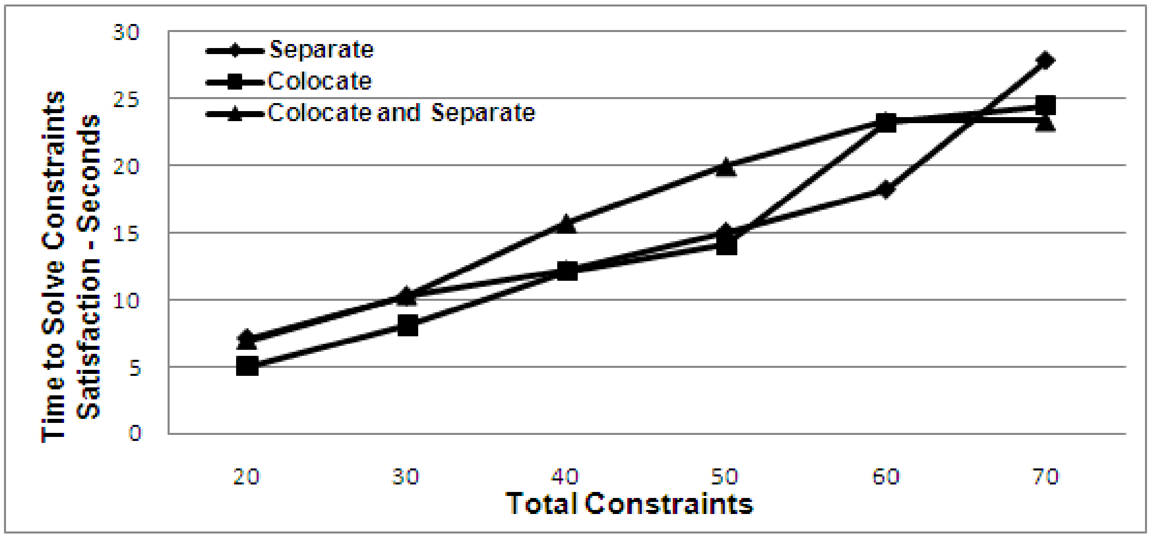

Figure 5.6: Time to solve constraints satisfaction comparison of *Colocate, Separate* and *both Colocate and Separate*

6. **Time Comparison of Colocate, Separate and Both Colocate and Separate**
   Finally, we compared the time to solve the constraints in random separate experiments and random colocate experiments, with the simulation time of both types of constraints experiment to determine the overhead incurred when both types of constraints are involved as compared to single type of constraints. Figure 5.6 shows that there is not a significant simulation time difference when both type of constraints are used as compared to when only one type of constraints are used. When any of the participating actor has both types of constraints then time taken to solve constraints increases linearly because in that scenario the colocate constraint is solved first which creates the group of colocated actors and one actor represents the group. The group formation takes more time which affects the time to solve the constraints. In *separate* part, the number of actors decreases which decreases the time to solve *separate* constraints.

5.5 Summary

This chapter discussed the evaluation of our approach. We evaluated programmability, performance and scalability. Our approach improves the programmability by providing the high level primitives which reduces the line of code and the complexity of the program. Experimental results show the scalability of the approach. The first two sets of experiments suggests that with one coordinator node system can perform well up to 200 actors with 600 constraints. It is also evident from the results that remote communication plays a major role because if the constraint actors are on the same node then time taken to solve constraints decreases dramatically. Another observation is that the time taken to solve spatial constraints increases as the total number of constraints increases. The number of violated constraints does not affect the time taken to solve the constraints.

Chapter 6
Conclusion and Future Work

6.1 Conclusion

In current mobile agent systems, application programmers have to deal with the low level communication details to implement the mobility of agents. Moreover, programmers have to mix code for the functionality of the application and low level communication in order to find the location of the computations. The mixing of code increases the complexity of the programming task. This work, spatial coordination, provides an alternative to deal with mobility and coordination.
Spatial coordination middleware is constructed for wireless sensor networks, which provides high level programming primitives to specify the spatial requirements of the application and decides the location of the computations. The middleware separates the spatial concerns from functional concerns of the application. In our work, an application's functionality is carried out by actors. The spatial requirements of the application draw the relationships among the actors. These relationships are presented in the form of spatial primitives. The programmer specifies the spatial requirements of the application through spatial primitives and the middleware accordingly finds the best location for those actors. A prototype implementation provides basic primitives and this thesis presents how to develop high level primitives on top of the basic primitives. The system also allows programmers to built their own high level primitives using basic primitives.

6.2 Contribution

Following are the key contributions of this work:

- Prototype implementation of a mechanism to declaratively specify the spatial requirements of applications which separates spatial concerns of applications from their functional concerns.

- High level programming constructs which hide details of individual migrations from the programmer, reducing programming complexity.

- Experimental results which show that the system can handle a large number of constraints without affecting performance and that network delay dominates the overhead.

6.3 Future Work

This work attempts to provide a way to handle mobility of mobile agents through spatial coordination. The list of spatial coordination primitives is by no means exhaustive; however, more of these types of primitives can be developed for different situations and implemented for real systems. The prototype implementation uses simulated tasks; real task would lead to authoritative results. We implemented prototype system for wireless sensor networks. However, our approach is not limited to wireless sensor networks, it can also be applied in other distributed systems such as computer networks. Since the prototype system is built for wireless sensor networks, which support unreliable communication, it causes lots of message retransmission as the number of constraints increases. A reliable communication protocol such as those found in computer networks, or an improved message re-transmission mechanism can reduce the message re-transmission and also reduce the time to satisfy spatial constraints.

In current system, constraints can be installed only once which can be extended to runtime installation and de-installation of constraints.

Our implementation is based on one coordinator node which causes a bottleneck when the number of actors increases. This problem can be solved by introducing multiple coordinator nodes so that communication of the actors with coordinator can be divided among multiple coordinator nodes. The multiple coordinators can be implemented in a similar way as multiple sinks in network in sensor networks. In the future, a large prototype system can be built with multiple coordinator nodes and a reliable communication mechanism to execute on real systems.

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.

[2] Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. A Survey on Sensor Networks. *Communications Magazine, IEEE*, 40(8):102–114, March 2002.

[3] Joachim Baumann, Fritz Hohl, Kurt Rothermel, and Markus Straer. Mole – Concepts of a Mobile Agent System. *World Wide Web*, 1(3):123–137, 1998.

[4] Sanja Bonevska, Andrijana Jankova, Aleksandra Mateska, Vladimir Atanasovski, and Liljana Gavrilovska. Early Fire Detection with WSN. In *17th Telecommunication forum TELFOR*, pages 1428–1485, 2009.

[5] Cristian Borcea, Chalermek Intanagonwiwat, Porlin Kang, Ulrich Kremer, and Liviu Iftode. Spatial Programming Using Smart Messages: Design and Implementation. In *ICDCS '04:Proceedings of the 24th International Conference on Distributed Computing Systems*, pages 690–699, Washington, DC, USA, 2004. IEEE Computer Society.

[6] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. MARS: A Programmable Coordination Architecture for Mobile Agents. *IEEE Internet Computing*, 4(4):26–35, 2000.

[7] Haifeng Chen, Hui Kang, Guofei Jiang, Kenji Yoshihira, and Akhilesh Saxena. VCAE: A Virtualization and Consolidation Analysis Engine for Large Scale Data Centers. *IEEE International Conference on Self-Adaptive and Self-Organizing Systems* , 0:1–10, 2010.

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

[9] Eugene C. Freuder and Richard J. Wallace. Partial constraint satisfaction. *Journal Artificial Intelligence*, 58:21–70, December 1992.

[10] Les Gasser. *DAI Approaches to Coordination*, pages 31–51. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[11] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7:80–112, 1985.

[12] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[13] Robert S. Gray, Cybenko, George, Kotz, David, Peterson, Ronald A., Rus, and Daniela. D'Agents: Applications and Performance of a Mobile-agent System. *Software–Practice & Experience*, 32(6):543–573, 2002.

[14] Ophir Holder, Israel Ben-Shaul, and Hovav Gazit. Dynamic Layout of Distributed Applications in FarGo. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 163–173, New York, NY, USA, 1999. ACM.

[15] Nadeem Jamali. *Cyberorgs: A Model for Resource Bounded Complex Agents.* PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2004.

[16] Jaein Jeong and David Culler. Incremental Network Programming for Wireless Sensors. In *Proceedings of the First IEEE International Conference on Sensor and Ad Hoc Communications and Networks*, pages 25–33, 2004.

[17] Venkata A. Kottapalli, Anne S. Kiremidjian, Jerome P. Lynch, Ed Carryer, Thomas W. Kenny, Kincho H. Law, and Ying Lei. Two-tiered Wireless Sensor Network Architecture for Structural Health Monitoring. volume 5057, pages 8–19. SPIE, 2003.

[18] Philip Levis and David Culler. Mate: A Tiny Virtual Machine for Sensor Networks. *SIGPLAN Notices*, 37:85–95, October 2002.

[19] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, pages 126–137, New York, NY, USA, 2003. ACM.

[20] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[21] Roger Mailler and Victor R. Lesser. Asynchronous Partial Overlay: A New Algorithm for Solving Distributed Constraint Satisfaction Problems. *Journal of Artificial Intelligence Research*, 25(1):529–576, 2006.

[22] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless Sensor Networks for Habitat Monitoring. In *WSNA '02: Proceedings of the 1st ACM Iinternational Workshop on Wireless sensor Networks and Applications*, pages 88–97, New York, NY, USA, 2002. ACM.

[23] Pedro Jos Marrn, Andreas Lachenmann, Daniel Minder, Jrg Hhner, Robert Sauter, and Kurt Rothermel. TinyCubus: A Flexible and Adaptive Framework Sensor Networks. In *EWSN'05: Proceeedings of the Second European Workshop on Wireless Sensor Networks*, pages 278–289, 2005.

[24] Yang Ni, Ulrich Kremer, and Liviu Iftode. A Programming Language for Ad-hoc Networks of Mobile Devices. In *LCR '04: Proceedings of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, pages 1–12, New York, NY, USA, 2004. ACM.

[25] Holger Peine and Torsten Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In *Proceedings of the First International Workshop on Mobile Agents*, pages 50–61, London, UK, 1997. Springer Verlag.

[26] Ichiro Satoh. MobileSpaces: A Framework for Building Adaptive Distributed Applications Using a Hierarchical Mobile Agent System. pages 161–168, 2000.

[27] Ichiro Satoh. Building and Selecting Mobile Agents for Network Management. *Journal of Network and Systems Management*, 14(1):147–169, 2006.

[28] Ichiro Satoh. Bio-Inspired Deployment of Software over Distributed Systems. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E90-A:2449–2457, November 2007.

[29] Amit Sinha and Anantha Chandrakasan. Dynamic Power Management in Wireless Sensor Networks. *IEEE Design & Test*, 18:62–74, March/April 2001.

[30] David Wong, Paciorek a d Noemi, Walsh, Tom, DiCelie, Joe, Young, Mike, Peet, and Bill. Concordia: An Infrastructure for Collaborating Mobile Agents. In *MA '97: Proceedings of the First International Workshop on Mobile Agents*, pages 86–97, London, UK, 1997. Springer-Verlag.

[31] Liyang Yu, Neng Wang, and Xiaoqiao Meng. Real-time Forest Fire Detection with Wireless Sensor Networks. In *Proceeding of the International Conference on Wireless Communications, Networking and Mobile Computing*, volume 2, pages 1214–1217. IEEE, 2005.

Appendix A

Sample Appendix

A.1 Data Used in Experiments

Table A.1: 20 Randomly generated separate constraints

Actor1	Actor2	Actor3	Actor4
105	44	34	17
24	42	-	-
87	84	45	48
85	37	-	-
14	61	15	-
25	102	-	-
74	21	27	-
47	79	-	-
210	108	-	-
66	83	-	-

Table A.2: 30 Randomly Generated Separate Constraints

Actor1	Actor2	Actor3	Actor4	Actor5
56	910	58	-	-
67	24	-	-	-
99	34	33	77	56
25	57	-	-	-
53	37	108	-	-
89	101	-	-	-
29	76	55	-	-
56	45	-	-	-
89	39	17	210	-
96	47	-	-	-

Table A.3: 40 Randomly Generated Separate Constraints

Actor1	Actor2	Actor3	Actor4	Actor5
12	32	55	-	-
13	14	56	-	-
16	18	66	77	-
78	81	71	-	-
52	25	45	-	-
21	65	58	55	-
310	410	910	-	-
56	57	-	-	-
21	110	17	11	49
29	39	15	-	-

Table A.4: 50 Randomly Generated Separate Constraints

Actor1	Actor2	Actor3	Actor4	Actor5
74	32	55	-	-
13	14	53	-	-
16	18	66	77	66
78	81	71	-	-
52	46	45	-	-
21	56	58	91	33
310	410	910	-	-
56	57	21	-	-
21	110	17	210	49
49	42	47	-	-

Table A.5: 60 Randomly Generated Separate Constraints

Actor1	Actor2	Actor3	Actor4	Actor5	Actor6
45	64	710	-	-	-
57	85	65	-	-	-
61	48	62	62	17	-
51	81	95	-	-	-
104	28	910	-	-	-
109	53	77	103	28	33
86	104	19	-	-	-
78	48	21	-	-	-
26	43	25	81	32	103
17	510	58	-	-	-

Table A.6: 70 Randomly Generated Separate Constraints

Actor1	Actor2	Actor3	Actor4	Actor5	Actor6
41	32	45	410	-	-
13	74	56	-	-	-
36	26	68	77	71	98
82	81	89	-	-	-
52	54	45	-	-	-
21	15	58	97	19	99
310	510	-	-	-	-
56	57	21	95	-	-
11	110	94	210	49	41
49	62	15	-	-	-

Table A.7: 20 Randomly Generated Colocate Constraints

Actor1	Actor2	Actor3	Actor4	Actor5
105	44	34	-	-
24	42	-	-	-
87	84	45	48	39
85	37	-	-	-
14	61	15	-	-
25	102	-	-	-
74	21	27	106	17
47	79	43	-	-
210	108	-	-	-
66	83	71	-	-

Table A.8: 30 Randomly Generated Colocate Constraints

Actor1	Actor2	Actor3	Actor4	Actor5	Actor6
56	910	58	16	25	-
67	24	85	82	-	-
99	34	33	-	86	-
25	57	11	51	-	-
53	37	108	79	-	-
89	101	73	-	59	610
29	76	55	-	-	-
78	45	-	95	-	-
13	39	17	210	14	-
96	47	310	-	-	-

Table A.9: 40 Randomly Generated Colocate Constraints

Actor1	Actor2	Actor3	Actor4	Actor5	Actor6	Actor7	Actor8
12	32	59	53	710	89	104	108
13	14	56	-	-	-	-	-
96	18	66	77	24	48	-	-
78	81	71	-	-	-	-	68
52	25	45	51	-	37	-	-
102	65	58	55	44	-	-	-
310	410	910	510	-	-	-	-
56	57	38	-	-	-	-	-
21	101	17	11	49	1010	-	-
29	39	15	94	31	79	-	-

Table A.10: 50 Randomly Generated Colocate Constraints

Actor1	Actor2	Actor3	Actor4	Actor5	Actor6	Actor7	Actor8	Actor9	Actor10
101	74	32	55	19	12	-	-	-	-
72	63	14	53	11	-	-	-	-	-
99	107	18	66	77	61	510	87	-	-
810	78	81	71	-	-	-	-	-	-
89	52	46	45	29	-	-	-	-	-
102	23	65	58	91	33	92	34	62	22
78	310	410	910	-	-	-	-	-	-
104	56	57	31	51	-	-	-	-	-
46	21	110	17	210	49	98	109	106	-
18	48	42	47	82	79	-	-	-	-

Table A.11: 60 Randomly Generated Colocate Constraints

Actor1	Actor2	Actor3	Actor4	Actor5	Actor6	Actor7	Actor8	Actor9	Actor10
46	310	52	104	17	32	16	61	92	38
78	110	58	106	51	210	57	48	710	-
19	39	27	810	19	47	29	96	63	-
32	81	95	89	-	-	-	-	-	-
108	28	910	84	29	-	-	-	-	-
109	53	77	103	91	33	99	-	-	-
86	105	19	-	-	-	-	-	-	-
97	98	21	42	66	-	-	-	-	-
26	43	25	88	410	1010	72	-	-	-
17	510	58	37	87	45	41	34	-	-

Table A.12: 70 Randomly Generated Colocate Constraints

Actor1	Actor2	Actor3	Actor4	Actor5	Actor6	Actor7	Actor8	Actor9	Actor10
75	11	23	61	97	44	67	-	-	-
25	85	56	88	76	51	55	37	18	-
63	59	46	35	99	108	79	64	610	32
103	27	36	510	89	47	-	-	-	-
74	48	45	42	96	210	38	-	-	-
69	104	22	65	17	310	98	410	82	-
31	910	12	13	34	41	52	72	-	-
101	102	86	66	43	106	91	-	-	-
26	87	110	14	93	77	54	1010	710	19
21	71	49	16	15	33	39	-	-	-

Table A.13: 20 Randomly Generated Colocate and Separate Constraints

Constraint Type	Actor1	Actor2	Actor3	Actor4	Actor5
Separate	105	44	34	-	-
Separate	24	42	-	-	-
Separate	87	84	45	48	-
Separate	85	37	-	-	-
Separate	14	61	15	-	-
Colocate	25	102	-	-	-
Colocate	74	21	27	106	48
Colocate	47	79	43	-	-
Colocate	210	108	-	-	-
Colocate	66	83	71	-	-

Table A.14: 30 Randomly Generated Colocate and Separate Constraints

Constraint Type	Actor1	Actor2	Actor3	Actor4	Actor5
Separate	66	910	58	-	-
Separate	67	42	-	-	-
Separate	99	34	33	77	56
Separate	25	57	-	-	-
Separate	53	37	108	-	-
Colocate	87	101	-	-	-
Colocate	29	76	55	68	-
Colocate	65	45	-	-	-
Colocate	89	39	17	210	56
Colocate	96	47	67	11	-

Table A.15: 40 Randomly Generated Colocate and Separate Constraints

Constraint Type	Actor1	Actor2	Actor3	Actor4	Actor5	Actor6	Actor7	Actor8
Separate	12	32	55	-	-	-	-	-
Separate	13	14	56	-	-	-	-	-
Separate	16	18	66	77	-	-	-	-
Separate	78	81	71	-	-	-	-	-
Separate	52	25	45	-	-	-	-	-
Colocate	31	65	58	56	99	101	-	-
Colocate	310	410	810	39	-	-	-	-
Colocate	68	57	-	-	-	-	-	-
Colocate	21	102	17	11	49	34	36	37
Colocate	29	35	15	55	78	910	210	33

Table A.16: 50 Randomly Generated Colocate and Separate Constraints

Constraint Type	Actor1	Actor2	Actor3	Actor4	Actor5	Actor6	Actor7	Actor8
Separate	12	32	55	910	-	-	-	-
Separate	13	14	56	27	-	-	-	-
Separate	16	18	66	77	-	-	-	-
Separate	78	81	71	-	-	-	-	-
Separate	52	25	45	-	-	-	-	-
Colocate	31	65	58	56	99	101	12	-
Colocate	310	410	810	39	-	-	-	-
Colocate	68	57	79	1010	-	-	-	-
Colocate	21	102	17	11	49	34	36	37
Colocate	29	38	15	55	78	91	210	62

Table A.17: 60 Randomly Generated Colocate and Separate Constraints

Constraint Type	Actor1	Actor2	Actor3	Actor4	Actor5	Actor6	Actor7	Actor8
Separate	49	64	710	91	42	-	-	-
Separate	57	85	65	27	-	-	-	-
Separate	61	48	11	46	17	-	-	-
Separate	51	81	95	99	-	-	-	-
Separate	104	28	91	-	-	-	-	-
Colocate	109	53	77	103	88	33	12	-
Colocate	86	104	19	64	-	-	-	-
Colocate	78	48	21	37	1010	-	-	-
Colocate	26	43	25	81	32	102	106	-
Colocate	29	38	15	55	910	210	45	-

Table A.18: 70 Randomly Generated Colocate and Separate Constraints

Constraint Type	Actor1	Actor2	Actor3	Actor4	Actor5	Actor6	Actor7	Actor8	Actor9
Separate	41	32	45	510	88	34	-	-	-
Separate	13	74	56	-	-	-	-	-	-
Separate	36	26	68	77	71	98	-	-	-
Separate	82	81	89	19	610	-	-	-	-
Separate	52	54	45	-	-	-	-	-	-
Colocate	109	53	77	103	28	33	-	-	-
Colocate	86	104	19	44	-	-	-	-	-
Colocate	78	48	21	37	1010	-	-	-	-
Colocate	26	43	25	81	32	102	-	-	-
Colocate	29	38	15	55	64	910	210	45	14