INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Mobile-Agent-Based Dynamic Channel Allocation with Waiting Queue

Zhang Yan-Mei

School of Computer Science

McGill University, Montreal

January, 1999

A Thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Master of Science

Abstract

The explosive growth of the wireless users requires efficient channel utilization. In this thesis, we present a new channel allocation strategy with waiting queue model instead of the block-and-clear model so that the quality of service in cellular network can be improved.

From a computational and network resource viewpoint, our model is based on the Mobile Agent Paradigm that has great advantages on reducing network traffic and dynamic adaptation. We also show the possibility to implement our algorithm on the Grasshopper mobile agent environment, which is compliant to the MASIF standard.

In order to evaluate the performance of our proposed model, a simulator is written in Java, which shows that our algorithm gives the lowest average blocking probability under different traffic load comparing with LP Scheme, First Available DCA Scheme and Simple FCA Scheme. Specifically, our algorithm has a tremendous capability of alleviating congestion in the hot spots of a cellular system.

Our research has been done within the group on "Advanced Studies on Mobile Environments" (ASME) as a part of the Consortium MIAMI (Mobile Intelligent Agents for Managing the Information Infrastructure).

Key Words: mobile agent, channel allocation, cellular system

Résumé

La croissance explosive des utilisateurs sans fil exige l'utilisation efficace de canal. Dans ce mémoire, nous présentons une nouvelle stratégie d'allocation des canaux avec le modèle de file d'attente au lieu du modèle *block-and-clear* de sorte que la qualité du service dans le réseau cellulaire puisse être améliorée.

Du point de vue de calcul et ressource de réseau, notre modèle est basé sur le Paradigme d'Agent Mobile qui a de grands avantages pour la réduction du trafic de réseau et l'adaptation dynamique. Nous montrons également la possibilité d'application de notre algorithme dans l'environnement d'agent mobile de Grasshopper qui est conforme à la proposition de MASIF, en vue de devenir la norme.

Afin d'évaluer l'exécution de notre modèle proposé, un simulateur est écrit en Java, ce qui prouve que notre algorithme donne la plus basse probabilité de blocage moyenne sous des contraintes de traffic varié, rivalisant avec *LP Scheme*, *First Available DCA Scheme* et *Simple FCA Scheme*. Spécifiquement, notre algorithme a une capacité plus grande d'alléger l'encombrement dans les points achalandés d'un système cellulaire.

Notre recherche a été faite chez le groupe sur "*Advanced Studies on Mobile Environments*" (ASME) comme partie du consortium MIAMI (*Mobile Intelligent Agents for Managing the Information Infrastructure*).

Mots Clés: mobile d'agent, d'allocation de canal, système cellulaire

Acknowledgements

I would foremost like to thank my Research Director, Dr. Petre Dini, for his direction and guidance, and for showing me the pleasure one can get out of research. This collaboration was for me a true enrichment.

I would also like to acknowledge my co-supervisor Prof. Gerald Ratzer for his understanding and helpful advice during my research period.

I would like to express my gratitude to the Computer Research Institute of Montreal (CRIM) for its financial support and providing me with the facilities, which I needed to carry out this research.

As I look back to remember the people I want to thank on a more personal level, particularly, enthusiastic thanks must be given to my friends Bonnie Wu, Linda Gu, Xiao-Bo Fan and Jane Fu for their friendship. Sincere thanks must also be given to the secretaries in school of Computer Science of McGill University , especially to Ms. Franca Cianci for her kind help and consideration.

Special thanks go to my parents and three older sisters and brother for their constant insistence that I could accomplish anything that I set out to do. Their support sustained me throughout my 21 years study life, and I own more to them than I can repay.

Most of all, my husband Song Hu deserves much of the credit for being there for me all the time when I need him, taking great patience to listen to me and explain things I never understood. I couldn't finish this thesis without a lot of support and encouragement from him.

Contents

List of Figures

Chapter 1

Introduction

Technological advances and rapid development of handheld wireless terminals have facilitated the rapid growth of wireless communications and mobile computing. Cellular technology is undoubtedly one of the most exciting and significant technological developments of the late 20th century. The number of cellular subscribers world-wide has grown from zero to 40 million in about twelve years and a recent forecast estimated that there would be 160-200 million cellular subscribers in over 120 countries by the year 2000 [URL1]. Already 40 per cent of new lines being installed in developing countries are using cellular technology. Since 1992 growth in the telecommunications industry was come only from the cellular market, with the number of new subscribers to the fixed PSTN (Public Switched Telephone Network) remaining static [URL1].

1.1 Structure of Cellular Communication System

A cellular communication system consists of mobile units linked via a radio network to an infrastructure of switching equipment interconnecting the different parts of the system, and allowing access to the normal (fixed) PSTN. Numerous transceivers called Base Stations (BS) are located at strategic places, and cover a given area or cell. A number of cells grouped together form an "area" and all its BS are in contact with a Mobile Switching Center (MSC) which stores information and directs calls to mobiles within its area. The MSCs of each area can communicate with each other, with a

special Gateway MSC that allows access to other cellular networks (e.g. the GSM system) or to the PSTN. This hierarchical structure can be seen in Figure 1.1.

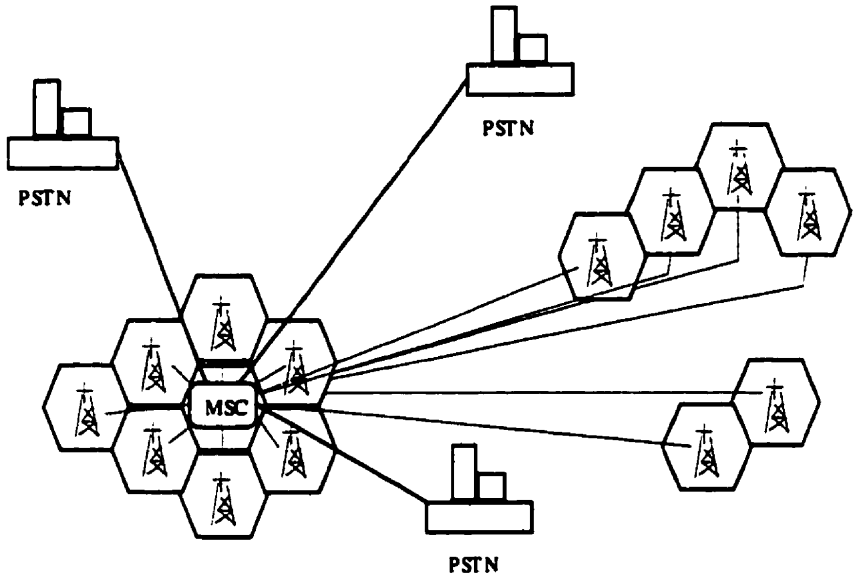

Figure 1.1 Structure of Cellular Communication

1.2 Motivation for Channel Allocation

A fundamental problem in wireless/mobile communication is that the electromagnetic spectrum is a scarce shared resource, and needs to be managed efficiently to provide an acceptable quality of service to communication-intensive applications. Federal Communications Commission (FCC) has allotted channels on the 824-849 MHz bands for transmission from mobiles and on 869-894 MHz for transmission from base station. The channel spacing is 30 KHz. This frequency band can accommodate 832 *duplex channels*. Among them, 21 channels are reserved for call setup, and the rest are used for voice communications [Ming89]. A *duplex channel* is also referred to as a *full-duplex channel*. It is used when data is to be exchanged between the two connected devices in both directions simultaneously, for example, if for throughput reasons data can flow in each direction independently [Fred96].

To satisfy the large demand of mobile telephone service, researchers at Bell Labs presented the ingenious idea of *frequency re-use* [URL9]. Each cell in the network uses only a given subset of all available channels, which are different from those used by adjacent cells. This ensures that interference does not occur between these cells. By restricting the power of transmission, signals will not travel too far outside each cell (due to the inverse square law) [URL9]. The same frequencies can then be re-used by cells that are sufficiently far away. All such sets that use the same channel are referred to as *co-channel sets*, or simply *co-channels*. The minimum distance at which co-channels can be reused with acceptable interference is called the "*co-channel reuse distance*" [Katzela96].

The tremendous growth of the wireless/mobile user population coupled with the bandwidth requirement demands efficient reuse of the scarce radio spectrum allocated to wireless/mobile communications. How the channels are to be assigned for simultaneous use in different cells directly affects the throughput of such systems. Efficient use of radio spectrum is also important from a cost-of-service point of view [Katzela96]. A reduction in the number of base stations and hence, a reduction in the cost-of-service can be achieved by more efficient reuse of the radio spectrum.

Up to now many channel allocation strategies have been suggested. We will proceed to briefly discuss different channel allocation algorithms in Chapter 2.

1.3 Objectives and Scope of the Thesis

The main purpose of this thesis is to design and implement efficient methods and algorithms in channel allocation. In our thesis, we present a new strategy in channel allocation schemes. Instead of changing hardware connections between base stations and MSC, we place the blocked calls in a waiting queue for a limited amount of time so that to decrease the blocking probabilities of the cellular system. And we use Mobile Agent approach to make dynamic decision and do the computation in the remote

destination in order to reduce the network traffic and improve the efficiency of resource allocation.

The rest of the thesis includes the following: Chapter 2 reviews the existed channel allocation schemes. Chapter 3 introduces the basic concept and architecture of Mobile Agent, lists the advantages of using Mobile Agent in network and communication. In Chapter 4, we present our new channel allocation algorithm and show how to use Mobile Agent to manage efficiently the resource in cellular communication network. Chapter 5 gives a brief introduction to Mobile Agent System Interoperability Facilities Specification (MASIF), and the platform Grasshopper used in our implementation that is compliant to MASIF standard. The implementation details and the obtained results are described in Chapter 6. Finally, we give a brief summary of our study and make recommendation for further work in the last chapter.

Chapter 2

Channel Allocation Schemes

A given radio spectrum (or bandwidth) can be divided into a set of disjoint or non-interfering radio channels. All such channels can be used simultaneously, while maintaining an acceptable received radio signal.

2.1 Channel Division Techniques

In order to divide a given radio spectrum into such channels, the following techniques can be used [Fred96]:

Frequency division (FD): divide the spectrum into disjoint frequency bands.

Time division (TD): divide the usage of the channel into disjoint time periods that called time slots.

Code division (CD): use different modulation codes to achieve the channel separation.

Furthermore, more elaborate techniques can be designed to divide a radio spectrum into a set of disjoint channels based on the combination of the above techniques.

Channel allocation schemes can be divided into a number of different categories depending on the comparison basis. When channel assignment algorithms are compared based on the manner in which co-channels are separated, they can be divided into *Fixed Channel Allocation* (FCA), *Dynamic Channel Allocation* (DCA), and *Hybrid Channel Allocation* (HCA).

2.2 Fixed Channel Allocation

In *Fixed Channel Allocation* (FCA) schemes, the area is partitioned into a number of cells, and a number of channels is assigned to each cell according to some reused pattern depending on the desired signal quality [Katzela96]. A definite relationship is assumed between each channel and each cell in accordance to co-channel reuse constraints.

2.2.1 Simple FCA Scheme

In the *Simple FCA Strategy*, the same number of nominal channels is allocated to each cell. *Nominal channels* are a set of channels, which are assigned to a given cell. This uniform channel distribution is efficient if the traffic distribution of the system is also uniform. In that case, the overall average blocking probability of the mobile system is the same as the call blocking probability in a cell. Since traffic in cellular systems can be non-uniform with temporal and spatial fluctuations, a uniform allocation of channels to cells may result in high blocking in some cells while others might have a sizeable number of spare channels. This could result in poor channel utilization [Katzela96]. Therefore, many *Non-uniform Channel Allocation Algorithms*, which are variations of FCA strategy, are adopted so that heavily loaded cells are assigned more channels than lightly loaded ones.

2.2.2 Non-uniform Compact Pattern Allocation

Non-uniform Compact Pattern Allocation is proposed by allocating channels to cells according to the traffic distribution in each of them [Ming91]. Thus, heavily loaded cells are assigned more channels than lightly loaded ones. The algorithm attempts to minimize the average blocking probability as nominal channels are allocated one at a time.

Let there be N cells and M channels in the system. The allocation of a channel to the set of co-channel cells forms a pattern that is referred to as the *Allocation Pattern*. In addition, the *Compact Allocation Pattern* of a channel is defined as the pattern with minimum average distance between cells. Given the traffic loads in each of the N cells and the possible compact pattern allocations for the M channels, the *Non-uniform Compact Pattern Allocation* algorithm attempts to find the compatible compact patterns that minimize the average blocking probability in the entire system as nominal channels are assigned one at the time.

2.2.3 Static Borrowing Schemes

Static Borrowing Schemes proposed in [Lewis73] re-assigns unused channels from lightly loaded cells to heavily loaded ones at distance less than the minimum reuse distance. Although in *Static Borrowing Schemes* channels are permanently assigned to cells, the number of nominal channels assigned in each cell may be reassigned periodically according to spatial inequities in the load. This can be done in a scheduled or predictive manner, with changes in traffic known in advance in the former, or based on measurements in the later.

2.2.4 Channel Borrowing Schemes

In *Channel Borrowing Schemes*, an acceptor cell that has used all its nominal channels can borrow free channels from its neighboring cells to accommodate new calls. A channel can be borrowed by a cell, if the borrowed channel does not interfere with existing calls.

When a channel is borrowed, several other cells are prohibited from using it. This is called channel locking. In contrast to *Static Borrowing Schemes*, *Channel Borrowing Schemes* deal with short term allocation of borrowed channels to cells, and once a call is completed, the borrowed channel is returned to its nominal cell. According to the different ways a free channel selected from a donor cell, many channel borrowing

strategies have been proposed [Lewis73][Ming89]. Normally, the *Channel Borrowing Schemes* can be divided into simple and hybrid.

Simple Channel Borrowing Strategy

In the *Simple Channel Borrowing Schemes* [Engel73], a channel set is nominally assigned to each cell. When a call arrives in a cell, a nominal channel is assigned to serve the call. If all nominal channels are busy, a nominal channel of the neighboring cells is borrowed to serve the call if that borrowing does not interfere with existing calls. Otherwise, the call is blocked.

The *Simple Channel Borrowing* strategy gives lower blocking than the traditional *Fix Channel Allocation* strategy under light and moderate traffic conditions. In heavy traffic conditions, however, channel borrowings may proliferate to such an extent that the channel usage efficiency drops drastically, causing a rapid increase in blocking probability [Kahwa78].

Hybrid Assignment Strategy

In the *Hybrid Assignment Strategy* [Kahwa78], the set of nominal channels assigned to each cell is divided into two subsets A and B. Subset A channels are used in the local cell only, while subset B channels can be lent to the neighboring cells.

In *Simple Hybrid Channel Borrowing Strategy* (SHCB), the ratio #A to #B is determined *a priori*, depending on an estimation of the traffic conditions and it can be adapted dynamically in a scheduled or predictive manner.

The *Borrowing with Channel Ordering* (BCO) strategy introduced in [Elnoubi82], outperforms SHCB by dynamically varying the local to borrowable channel ratio according to changing traffic conditions. In BCO strategy, all nominal channels are ordered such that the first channel has the highest priority to be assigned to the next local call, and the last channel is given the highest priority to be borrowed by the neighboring cells.

In BCO strategy, a channel is suitable for borrowing only if it is simultaneously free in the three nearby co-channel cells. This requirement is too stringent and decreases the number of channels available for borrowing. So [Ming89] presented a new strategy called *Borrowing with Directional Channel Locking* (BDCL) in 1989. In this strategy, when a channel is borrowed, the locking of this channel in the co-channel cells is restricted only to those affected by this borrowing. Thus the number of channels available for borrowing is greater than that of the BCO strategy. To determine in which case a "locked" channel can be borrowed, "lock directions" are specified for each locked channel. The scheme also incorporates reallocation of calls from borrowed to nominal channels and between borrowed channels in order to minimize the channel borrowing of future calls, and especially the multiple channel borrowing observed during heavy traffic.

2.3 Dynamic Channel Allocation

Due to short term temporal and spatial variations of traffic in cellular systems, *Fix Channel Allocation* (FCA) schemes are not able to attain a high channel efficiency. To overcome this, *Dynamic Channel Allocation* (DCA) schemes have been studied during the past twenty years.

In contrast to FCA, all channels are potentially available to all cells and are assigned to cells dynamically as calls arrive. If this is done right, it can take advantage of temporary changes in the spatial and temporal distribution of calls in order to serve more users. For example, when calls are concentrated in a few cells, these cells can be assigned more channels without increasing the blocking rate in the lightly used cells.

In DCA, a channel is eligible for use in any cell provided that signal interference constraints are satisfied. The main idea of all DCA schemes is to evaluate the cost of using a candidate channel, and select the one with the minimum cost provided that certain interference constraints are satisfied. The selection of the cost function is what differentiates DCA schemes [Cox72].

The selected cost function might depend on the future blocking probability in the vicinity of the cell, the usage frequency of the candidate channel, the reuse distance, channel occupancy distribution under current traffic conditions, radio channel measurements of individual mobile users or the average blocking probability of the system. DCA schemes can be divided into centralized and distributed schemes with respect to the type of control they employ. Several simulation and analysis results have shown that centralized DCA schemes can produce near optimum channel allocation, but at the expense of a high centralization overhead [Nettleton89].

2.3.1 Centralized DCA Schemes

In the centralized DCA schemes, a channel from the central pool is assigned to a call for temporary use by a centralized controller. The difference between these schemes is the specific cost function used for selecting one of the candidate channels for assignment.

First Available strategy (FA) assigns the first available channel within the reuse distance encountered during a channel search to the call [Cox72]. In the *Locally Optimized Dynamic Assignment* (LODA) strategy [Ming89], the concept of nominal channels is not used. Instead, a particular cell having a call to serve evaluates the cost of using each candidate channel. The channel with the minimum cost is then assigned. The selected cost function is based on minimizing the channel reuse distance. In other words, the cells using the same channel are packed as compactly as possible so that the channel could again be reused in the closest possible range. *Channel Reuse Optimization Schemes* attempt to maximize the efficiency of the system by optimizing the reuse of a channel in the system area [Cox72][Kazunori92]. *Channel Rearrangement Schemes* are to switch calls already in process, whenever possible, from channels that these calls are using, to other channels with the objective of keeping the distance between cells using the same channel simultaneously to a minimum [Donald73]. Thus, the channel reuse is more concentrated and more traffic can be carried per channel at a given blocking rate.

2.3.2 Distributed DCA Schemes

The proposed distributed DCA schemes use either local information about the current available channels in the cell's vicinity (cell based) schemes [Chih93] or signal strength measurements [Mutsumu93][Furuya91].

In the *Cell Based Schemes*, a channel is allocated to a call by the base station where the call is initiated. The difference with the centralized approach is that each base station keeps information about the current available channels in its vicinity. The channel pattern information is updated by exchange of status information between base stations. The *Cell Based Schemes* provide near optimum channel allocation at the expense of excessive exchanged of status information between base stations, especially under heavy traffic loads.

In the *Interference Adaptation Schemes* which rely on signal strength measurements [Mutsumu93], a base station uses only local information, without the need to communicate with any other base station in the network. Thus, the system is self-organizing, and channels can be placed or added everywhere. These schemes allow fast real time processing and maximal channel packing at the expense of increased co-channel interference probability with respect to ongoing calls in adjacent cells, which may lead to undesirable effects such as interruption, deadlock and instability.

Local Packing algorithm presented in [Chih93] can be implemented distributively at the base stations with a simple Augmented Channel Occupancy (ACO) table, or centrally at the Mobile Switching Center without the need of a distributed database. [Chih93] implemented this algorithm distributively and showed that, unlike some other DCA algorithms, even when the network has a large number of channels, it maintains a favorable performance over FCA under uniform traffic in the region of interest. More importantly, the LP algorithm has a tremendous capability of alleviating congestion at traffic hot spots.

In our thesis, we use the main idea of LP algorithm because its higher performance over other algorithms. But instead of keeping the database distributed in each base station, we keep the database which contains the information of each base station in Mobile Switching Center (MSC), and then using Mobile Agent Technology to analyze if there is channel available during considered period. We will describe this algorithm in detail in Chapter 4. In the next chapter, we will introduce the basic concept and architecture of Mobile Agent Technology.

Chapter 3

Mobile Agent Technology

With the recent explosive development of computer networking and the Internet, a gap has developed between the sheer amount of information that is available and the ability to process or even locate the interesting pieces. This does not only apply to "application-oriented data" such as scholarly publications, stock exchange quotes, or weather satellite images, but also to the monitoring and operation of huge computer networks. Agents show a possible way out of this dilemma [Anselm95]. Agent-based systems have recently gained considerable attention in many areas of computer science and information processing such as software engineering, human interfaces, and network management. The incorporation of intelligence implies dealing in an adaptive way with unforeseeable changes in the remote environment [Chess95].

3.1 Basic Concept

3.1.1 Agent

An agent is a computational entity, which acts on behalf of other entities in an autonomous fashion, performs its actions with some level of *proactivity* and/or *reactivity*. For instance, an estate agent who simply places a "for sale" sign outside a property for sale and waits for purchasers to come into his shop is behaving in a much more reactive fashion, than an agent who proactively advertises the property in the local press. The same agent can display high amounts of both proactivity and reactivity

at different time [URL2]. An agent also exhibits some level of the key attributes of learning, co-operation and mobility.

In the area of computing and information systems, the notion of an agent implies a remotely executing program with a certain degree of autonomy, usually helping with the tasks of information processing or retrieval [Hosoon96].

3.1.2 Stationary Agent and Mobile Agent

Conceivably, agents can be stationary or mobile. A stationary agent executes only on the system where it begins execution. If the agent needs information that is not on that system, or needs to interact with an agent on a different system, the agent typically uses a communications transport mechanism such as Remote Procedure Call (RPC) [GMD97]. For example, in the Simple Network Management Protocol (SNMP) which is designed to give a user the capability to remotely manage a computer network by polling and setting terminal values and monitoring network events, the stationary agent is used to run off of each node on the network [URL15]. Many UNIX software vendors include this with their terminal software. It collects network and terminal information as specified in the Management Information Base (MIB) [URL10].

In contrast to a stationary agent, a mobile agent is not bound to the system where it begins execution. It has the unique ability to transport itself from one system in a network to another. This submission is primarily concerned with mobile agents. The ability to travel permits a mobile agent to move to a destination agent system that contains an object with which the agent wants to interact. Moreover, the agent may utilize the object services of the destination agent system [GMD97].

3.1.3 Agent System

An agent system is a platform that can create, interpret, execute, transfer and terminate agents. An agent system is associated with an authority that identifies the person or

organization for whom the agent system acts. An agent system is uniquely identified by its name and address. A host can contain one or more agent systems. Figure 3.1 describes an agent system [GMD97].

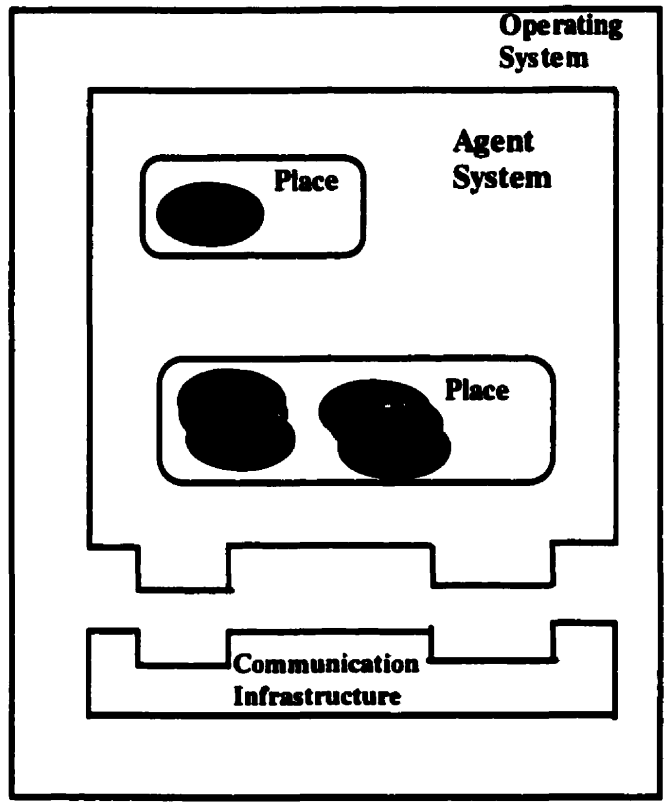

Figure 3.1 Agent System

In above figure, when an agent transfers itself, the agent travels between execution environments called *places*. We will introduce the *place* concept in Section 3.1.4.

An agent system type describes the profile of an agent. For example, if the agent system type is Aglet, the agent system is implemented by IBM, supports Java as the Agent Language, uses Itinerary for travel, and uses Java Object Serialization for its serialization [GMD97]. This specification recognizes agent system types that support multiple languages, and languages that support multiple serialization methods. Therefore, a client requesting an agent system function must specify the agent profile

(agent system type, language, and serialization method) to identify uniquely the desired functionality.

All communication between the agent systems is through the Communication Infrastructure (CI) which provides communications transport services, naming, and security services for an agent system. The region administrator defines communication services for intra-region and inter-region communications. The *region* concept will be introduced in section 3.1.5. Figure 3.2 describes agent system to agent system interconnection [GMD97].

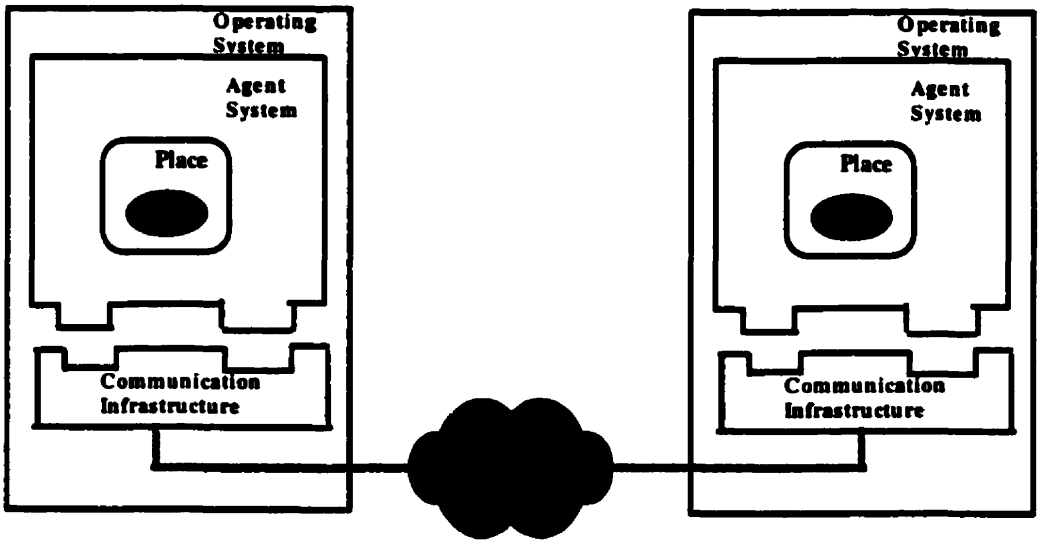

Figure 3.2 Agent System to Agent System Interconnection

In general, the core actions among agent systems are:

- Transferring an agent
- Creating an agent
- Providing globally unique agent names and locations
- Supporting the concept of a region
- Finding a mobile agent
- Ensuring a secure environment for agent operations

3.1.4 Place

A place is a context within an agent system in which an agent can execute. This context can provide functions such as access control. The source place and the destination place can reside on the same agent system, or on different agent systems that support the same agent profile [GMD97].

A place is associated with a location, which consists of the place name and the address of the agent system within which the place resides. An agent system can contain one or more places and a place can contain one or more agents. When a client requests the location of an agent, it receives the address of the place where the agent is executing.

3.1.5 Region

A region is a set of agent systems that have the same authority, but are not necessarily of the same agent system type. The concept of region allows more than one agent system to represent the same person or organization.

Regions are interconnected via one or more networks and may share a naming service based on an agreement between region authorities and the specific implementation of these regions. A non-agent system may also communicate with the agent systems within any region as long as the non-agent system has the authorization to do so. Figure 3.3 describes region to region interconnection [GMD97].

3.2 Mobile Agent Architecture

3.2.1 Why Use an Agent Architecture

Agent architecture provides a flexible alternative to client/server and distributed object architectures. It contains many advantages. Three of the most important advantages are [Steven97]:

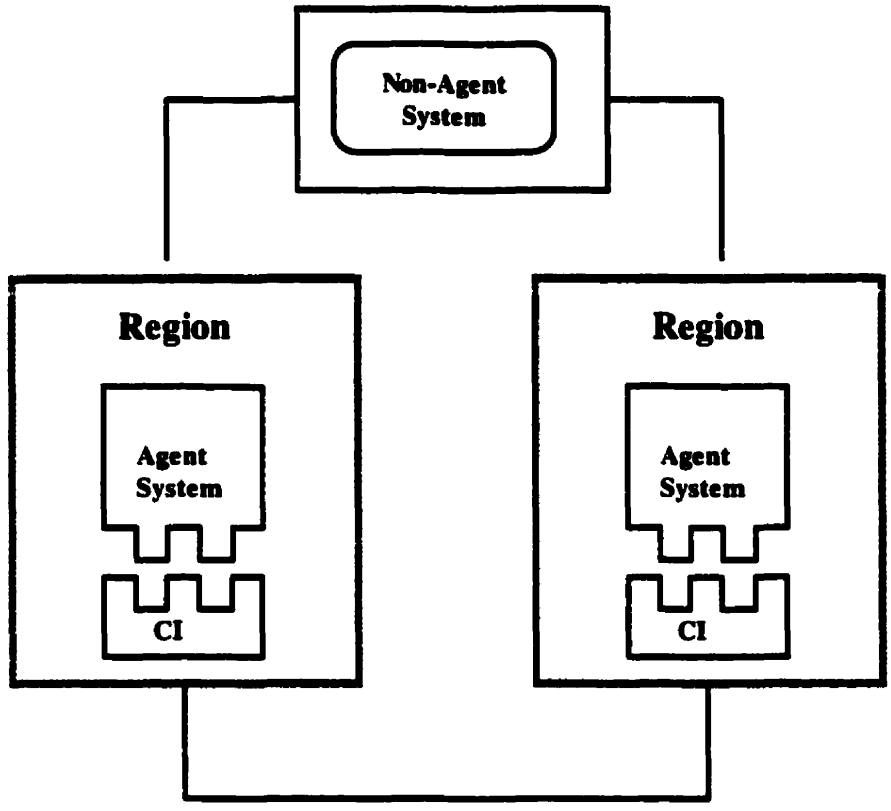

Figure 3.3 Region to Region Interconnection

1. Performs much processing at the server where local bandwidth is high, thus reducing the amount of network bandwidth consumed and increasing overall performance.

2. Operates independently on the application from which the agent was invoked. The agent operates asynchronously, meaning that the client application does not need to wait for the results. This is especially important for mobile users who are not always connected to the network.

3. Allows for the injection of new functionality into a system at run time. An agent system essentially contains its own automatic software distribution mechanism that has built-in support for mobile code, new functionality generally can be installed automatically at run time.

3.2.2 Structure of an Mobile Agent

Because a mobile agent is a software entity that exists in a software environment, it inherits some of the characteristics of an agent. A mobile agent must contain all of the following models [Chess95][White95]:

Agent model: defines the internal structure of the intelligent agent part of a mobile agent. In essence, it defines the autonomy, learning and co-operative characteristics of an agent. Additionally, it specifies the reactive and proactive nature of agents.

Life-cycle model: defines the different execution states of a mobile agent and the events that cause the movement from one state to another.

Computational model: defines how a mobile agent executes when it is in a running state.

Security model: mobile agent security can be split into two broad areas. The first involves the protection of host nodes from destructive mobile agents, while the second involves the protection of mobile agents from destructive hosts.

Communication model: communication is used when accessing services outside of the mobile agent, during co-operation and co-ordination between mobile agents and other entities, and finally to facilitate competitive behavior between self-interested agents.

Navigation model: concerns itself with all aspects of agent mobility from the discovery and resolution of destination hosts to the manner in which a mobile agent is transported.

These models are highly integrated and interdependent. Figure 3.4 gives a simple view of the structure of a mobile agent [URL2].

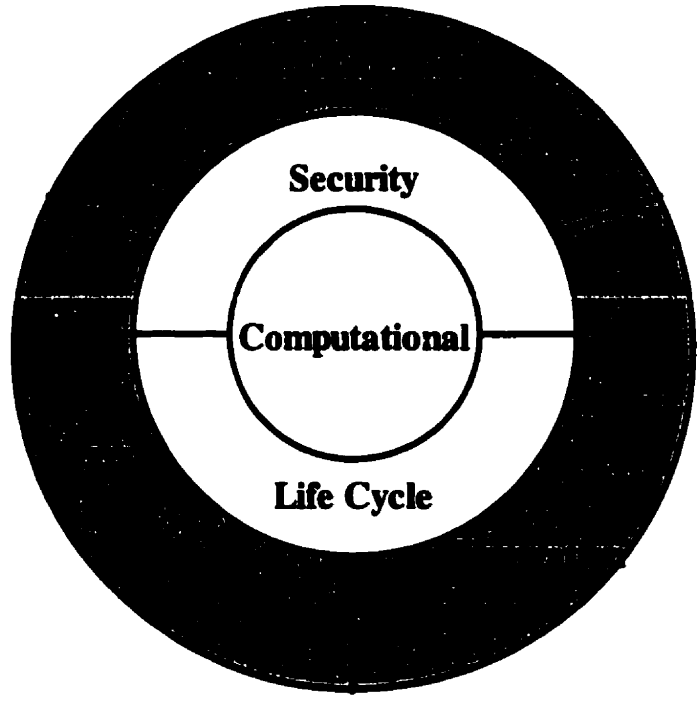

Figure 3.4 Simple View of the Structure of a Mobile Agent [URL2]

The core of the structure is based on the computational model. This has significant impact on the other models. It defines how we address other agents, hosts and resources, which is important to the security model. The type of life cycle model adopted is dependent on the facilities of the computational model.

Both the security and life cycle models are structurally very close to the core. Security issues permeate every aspect of a mobile agent and therefore must be provided for at the most basic level. The life cycle model defines the valid states for an agent.

The outer layer contains the communication, navigation and agent models. The agent model defines the "intelligent agent" aspects of a mobile agent such as learning and collaboration functions. The communication model is heavily dependent on the security model so that agents are not corrupted by other agents or hosts. Finally, the navigation model is also dependent on the security model as it hands itself over to the host to be transported to another node.

3.2.3 Mobile Agent Environment

A mobile agent environment is a software system, which is distributed over a network of heterogeneous computers. Its primary task is to provide an environment in which mobile agents can execute. The mobile agent environment implements the majority of the models that appear in the mobile agent definition. It may also provide: support services which relate to the mobile agent environment itself, support services pertaining to the environments on which the mobile agent environment is built, support services accessing to other mobile agent systems, and finally support for openness when accessing non-agent-based software environments [URL2].

The basic mobile agent architecture is illustrated in Figure 3.5 [URL2]. The mobile agent environment is built on top of a host system. The smiling faces are mobile agents that travel between mobile agent environments. Communication between mobile agents (local and remote) is represented by bi-directional arrows. Communication can also takes place between a mobile agent and a host service.

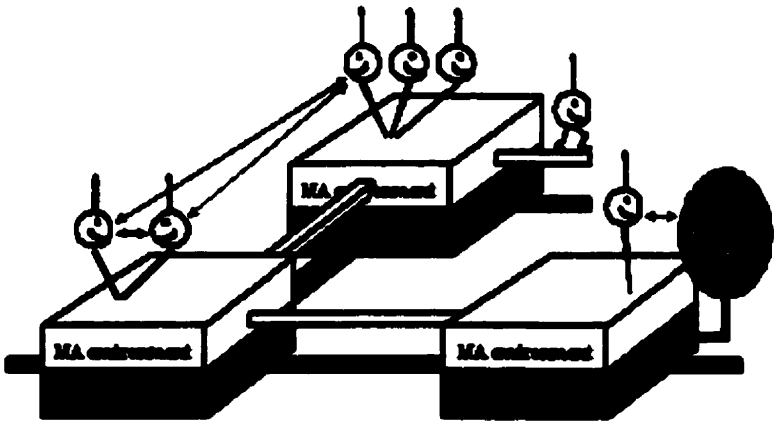

Figure 3.5 Basic Mobile Agent Architecture [URL2]

3.3 Advantages of the Mobile Agent Paradigm

Mobile agents have many advantages over the traditional client/server model. In [URL2], various claims related to mobile agent are examined and its advantages are summarized as following:

- *Efficiency*: Mobile agents consume fewer network resources since they move the computation to the data rather than the data to the computation.

- *Reduction of network traffic*: Most communication protocols involve several interactions, especially when security measures are enabled. This causes a lot of network traffic. With mobile agents, one can package up a conversation and ship it to a destination host where the interactions can take place locally.

- *Asynchronous autonomous interaction*: Tasks can be encoded into mobile agents and then dispatched. The mobile agent can operate asynchronously and independent of the sending program. An example of this would be a mobile device dispatching an autonomous search agent onto the fixed network, disconnecting, then reconnecting some time later to collect the results of the search.

- *Interaction with real-time entities*: Real-time entities such as software controlling an ATM switch or a safety system in a nuclear installation require immediate responses to changes in their environment. Controlling these entities from across a potentially large network will incur significant latencies. For critical situations (nuclear system control), such latencies are intolerable. Mobile agents offer an alternative. They can be dispatched from a central system to control real-time entities at a local level and also process directives from the central controller.

- *Dynamic adaptation*: Related to the above topic, mobile agents have the ability to autonomously react to changes in their environment. However, such changes must be communicated to mobile agents from the mobile agent environment.

- *Dealing with vast volumes of data*: When vast volumes of data are stored at remote locations, as in weather information systems, the processing of this data should be performed local to the data, instead of transmitting it over a network.

- *Robustness and fault tolerance*: The ability of mobile agents to react dynamically to adverse situations makes it easier to build fault tolerant behavior, especially in a highly distributed system.

- *Support for heterogeneous environments*: Both the computers and networks on which a mobile agent system is built are heterogeneous in character. As mobile agent systems are generally computer and network independent, they support transparent operation.

- *Personalize server behavior*: In the intelligent networks, mobile agents are proposed as a way to personalize the behavior of network entities (e.g., routers) by dynamically supplying new behavior.

- *Support for electronic commerce*: Mobile agents can be used to build electronic markets. Here mobile agents embody the intentions, desires, and resources of the participants in such a market.

- *Convenient development paradigm*: The design and construction of distributed systems can be made easier by the use of mobile agents. Mobile agents are inherently distributed in nature and hence are a natural view of a distributed system.

Since mobile agents have many advantages over the traditional client/server model, we present a mobile-agent-based dynamic channel allocation algorithm. In the next chapter, we will describe our proposed algorithm in detail.

Chapter 4

Mobile-Agent-Based Dynamic Channel Allocation Algorithm

4.1 Motivation

In Chapter 2, we reviewed the different channel allocation schemes. We found the various techniques proposed in the literature made great effort on how to efficiently reuse channels in the cellular system so that to minimize the blocking probabilities for network traffic. However, those algorithms use the block-and-clear model, that is, if a call cannot allocate a channel, then this call is blocked and simply cleared immediately [Donald73][Ming89][Chih93]. But in fact, in some cases, a short delay before being connected could be acceptable. People would rather wait for a few seconds to get connected than redial later.

In our research, we specifically consider this case where the mobiles may not need to be connected immediately, mobile users might be able to tolerate a short delay before being connected. If the waiting time is chosen properly, the total blocking probability of the network can be reduced considerably [Reece96].

In our proposed new algorithm, we place the blocked calls in a waiting queue for a limited amount of time. During this short period, if there is a channel available for this call, then we can assign this channel to the call immediately. If there is still no channel available, then this call is cleared. This algorithm can give people the choice to wait

for a connection so that it improves the *quality of service* (QoS) in cellular network. QoS is a set of user-perceivable attributes of that which makes a service what it wants [Race82]. Specific QoS parameters take *subjective* or *objective* values, expressed in *user-understandable* language. Objective values are *customer-verifiable*, which are defined and measured across particular service parameters. Those subjective values represent the *provider's opinion* which are defined and estimated with respect to user surveys [Coch92].

In a customer-provider relation, QoS is defined by service parameters of the provider (called the system performance at the QoS provider interface), which satisfy customer requests (QoS customer interface) [Dini97]. Some major needs for QoS enhancements were summarized as follows:

- possibility to choose many values at a time between space values of parameters
- transparent verification and validation of preconditions between combined space values
- possibility to change dynamically the cardinality of value space
- improving the trader with new services in order to monitor relations

Our algorithm is based on the main idea of Local Packing algorithm [Chih93], because LP algorithm maintains a favorable performance over FCA under uniform traffic in the region of interest even when the network has a large number of channels. More importantly, it has a tremendous capability to alleviate congestion at traffic hot pots.

Up to now, most cellular network are using central control. It can produce near optimum channel allocation, but at the expense of a high centralization overhead [Nettleton89]. In our algorithm, we keep a channel status table for assigning and releasing channels in the Mobile Switch Center (MSC), each channel assignment procedure and channel release procedure require some choice to be made dynamically. These are arguably the hardest interactions to deal with from a computational

viewpoint and if using the traditional way, it will cause high network traffic [Mike95]. They are precisely the class of problems that our model is based on Mobile Agent addresses.

As we introduced before, Mobile Agent has great advantages on reducing network traffic, dealing with vast volumes of data and dynamic adaptation. Therefore, our study uses Mobile Agent to make dynamic decision and do the computation when it is in the remote destination (MSC). The Mobile Agent can select a proper channel for a call or monitor when a channel is available for a waiting call. It makes our algorithm more efficient and competitive with respect to the traditional methods without using the Mobile Agent.

In the following section, we will describe our algorithm in detail. First, we give a brief introduction to the main idea of LP algorithm.

4.2 Main Idea of LP Algorithm

In the LP algorithm, each base station assigns channels to new or hand-off calls using the Augmented Channel Occupancy Matrix (ACO), which contains the necessary and sufficient local information for each base station to select a channel.

Let M be the total number of available channels in the system and k_i be the number of neighboring cells to cell i within the co-channel interference distance. The ACO matrix, as shown in Figure 4.1, has $M+1$ columns and k_i+1 rows.

The first M columns correspond to the M channels. The first row indicates the channel occupancy in cell i and the remaining k_i rows indicate channel occupancy pattern in the neighborhood of i, as obtained from neighboring base stations. The last column of the matrix corresponds to the number of current available channels for each of the $k_i + 1$ co-channel cells.

	Channel									
cell	1	2	3	4	5	6	7	... ...	M	free channel
i		x			x					0
i_1	x			x			x			0
i_2			x							2
i_3	x						x	... ...		0
i_4			x			x				5
i_5				x						3
...	...	...	...	...	...	...	...		...	...
i_M										4

Figure 4.1 The Augmented Channel Occupancy Table

4.2.1 Channel Assignment Procedure Based on LP Algorithm

When a base station receives an access request, it searches for an empty column in its ACO table. If successful, it assigns that channel to the request. If the ACO table contains no empty column, the base station then looks for a column with a single check mark. If found, it identifies the corresponding cell and checks to see whether that cell has channels available (indicated by a nonzero entry in the last column). If that is the case, it sends a request to that cell to reassign the call currently using that channel to another channel and it assigns the found channel to its access request. The Figure 4.2 describes the procedure of channel assignment.

The content of the ACO table is updated by collecting channel occupancy information from all interfering cells through a simple procedure: each base station, when seizing or releasing a channel, sent this information to all interfering cells' ACO tables. A base station should send out update information also if its own entry in the augmented column has changed as a result of another cell seizing or releasing channels.

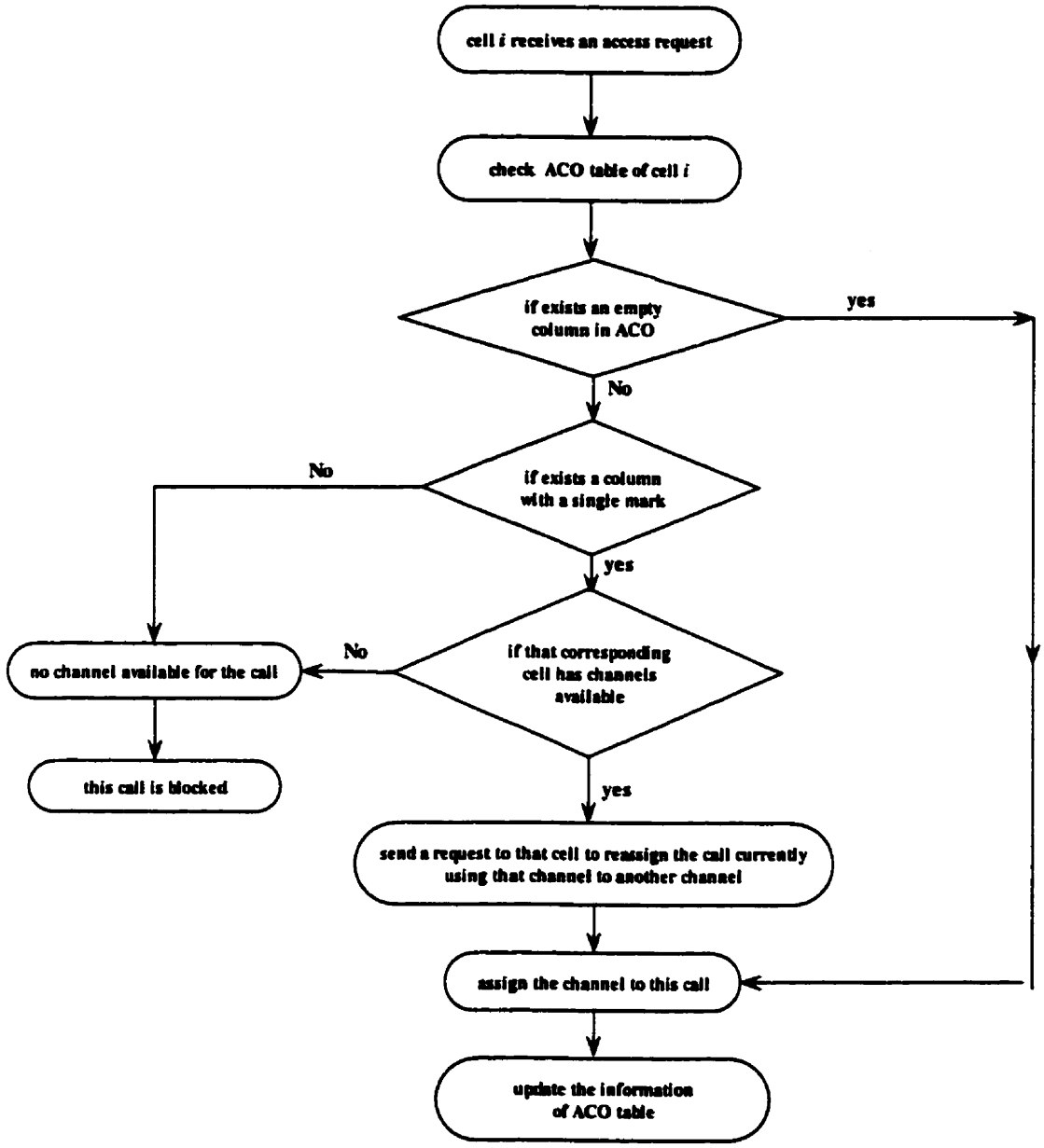

Figure 4.2 Channel Assignment Procedure Based on LP Algorithm

4.2.2 Channel Release Procedure Based on LP Algorithm

In the literature [Chih93], only the channel assignment procedure is considered. But in our proposed algorithm, we need to consider how to assign a released channel to a waiting call. According to the idea of the LP algorithm, we present the channel release

procedure based on the LP algorithm. Figure 4.3 describes the procedure of channel release:

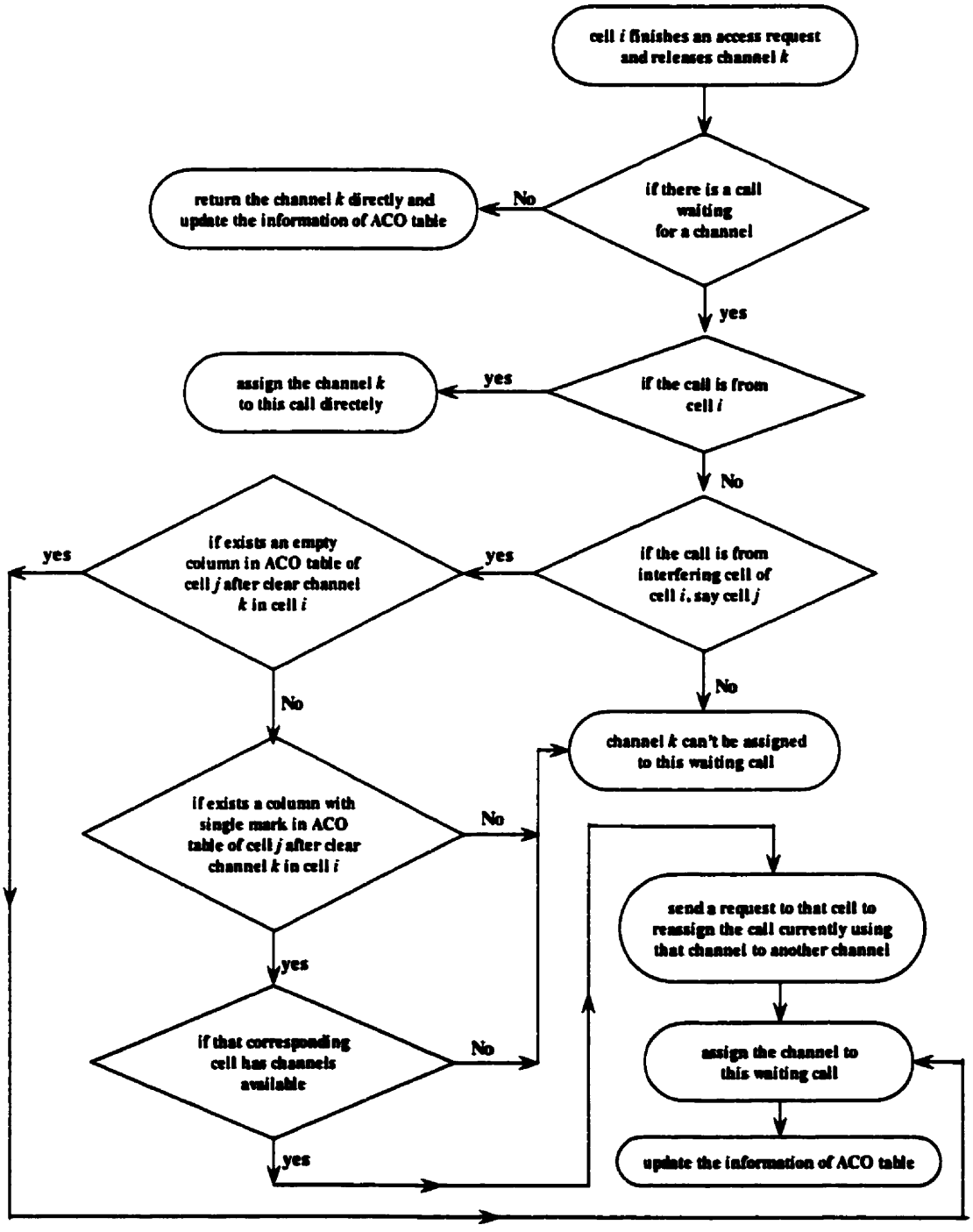

Figure 4.3 Channel Release Procedure Based on LP Algorithm

When a call on channel *k* of cell *i* terminates, if there is a call waiting in the queue, first check if the waiting call also comes from base station *i*, if yes, this channel can be directly assigned to this call and doesn't need update anything. If the waiting call is not from base station *i*, say it is from base station *j*, check if cell *j* is the interfering cell of cell *i*, if yes, check to see if this channel can be re-assigned to the waiting call. The following steps are executed:

1. First it searches for an empty column in cell *j*'s ACO table after clearing the check mark of the *k* channel in cell *i*. If successful, it assigns that channel to the waiting call.

2. If the ACO table contains no empty column, then looks for a column with a single check mark after clearing the check mark of the *k* channel in cell *i*. If found, it identifies the corresponding cell and checks to see whether that cell has channels available (indicated by a nonzero entry in the last column). If that is the case, it sends a request to that cell to reassign the call currently using that channel to another channel and it assigns the found channel to the waiting call. If no channel is available, then the channel *k* can't be assigned to this waiting call.

The content of the ACO table is updated by the same way as the one used in channel assignment procedure.

4.3 Design of Our Algorithm

In our algorithm, instead of implementing distributively at the base stations with a simple Augmented Channel Occupancy table as Chih-Lin and Pi-Hui Chao [Chih93] did, we propose to keep a table which contains the information of all base stations in the Mobile Switching Center (MSC).

Each base station is treated as an agency where a stationary agent *Base* is residing to provide services for the mobiles. If it receives a call request from a mobile, it will

create a *Start* agent and send it to the remote location MSC to select a channel for this call. If it receives a call finished message from a mobile, it will create a *Finish* agent and send it to MSC to release this channel. MSC is an agency where a stationary agent *Directory* is residing to provide information needed by each base station and to update the information in channel status table. Figure 4.4 shows this simple structure.

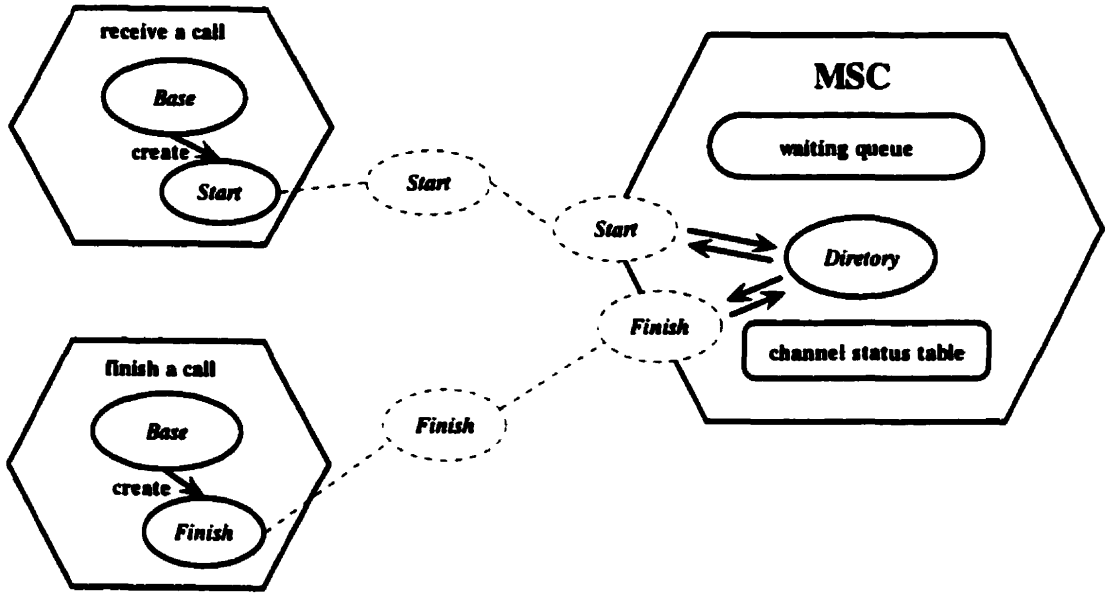

Figure 4.4 Structure of Mobile-Agent-Based Cellular System

4.3.1 Channel Assignment Procedure

When the *Base* agent in cell *i* receives an access request from a mobile, the following steps are executed (see Figure 4.5):

1. First, the *Base* agent creates a mobile agent *Start* with cell *i*'s authority and gives it the fixed amount of time it should wait for a channel.

2. Sends *Start* to the place where the stationary agent *Directory* is residing in MSC.

3. The *Start* agent meets the *Directory* agent and asks the service.

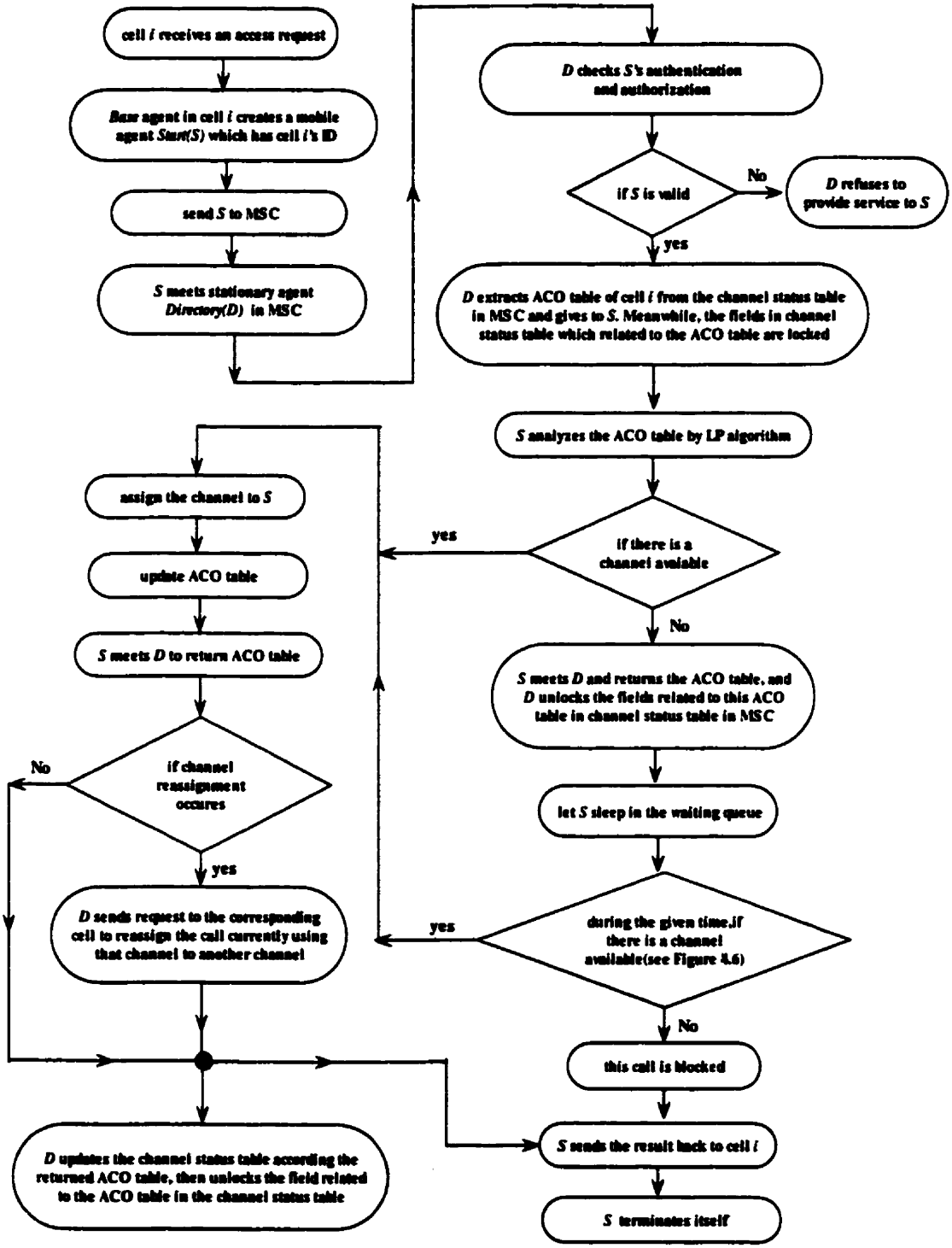

Figure 4.5 Channel Assignment Procedure

4. The *Directory* agent first checks the authentication and authorization of the *Start* agent, if they are not valid, then the *Directory* agent refuses to provide service. If the authentication and authorization are valid, then the *Directory* agent extracts the cell *i*'s ACO table and gives it to the *Start* agent. In the meantime, the fields in the total channel status table, which related to this ACO table are locked. That means they are not allowed to be accessed by other agent.

5. The *Start* agent uses the LP algorithm to analyze the ACO table.

    - If there is a channel available, then this channel is assigned to the *Start* agent. Meanwhile, the information in ACO table is updated. Then the *Start* agent meets the *Directory* agent and returns the updated ACO table. If the channel is available through channel-reassignment, the *Directory* agent sends a request to the corresponding cell to reassign the call currently using that channel to another channel. And the *Start* agent sends the result back and then terminates itself. In the meantime, the *Directory* agent updates the information in the total channel status table according to the updated ACO table, and then, the fields related to this ACO table are unlocked.

    - If there is no channel available, then the *Start* agent meets the *Directory* agent and returns the ACO table. The *Directory* agent unlocks the fields related to this ACO table and no update is needed. The *Start* agent goes into a waiting queue and sleeps there until invoked by the *Directory* agent. If in the given time, no channel is available, then the *Start* agent sends a signal back and terminates itself. In this case, the call is blocked.

4.3.2 Channel Release Procedure

When a call on channel *k* of cell *i* terminates, the following steps are executed (see Figure 4.6):

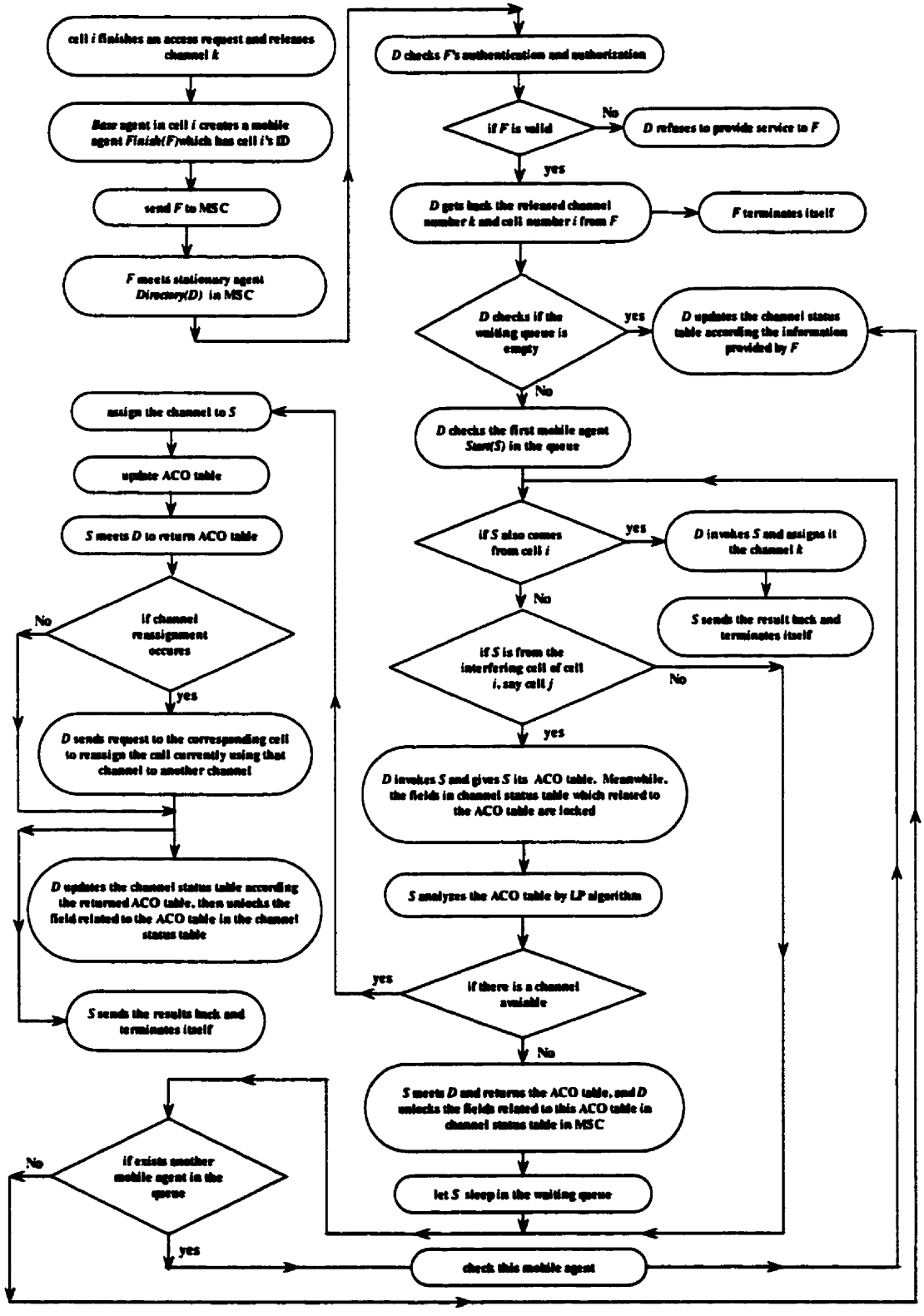

Figure 4.6 Channel Release Procedure

1. First, the *Base* agent in cell *i* creates a mobile agent *Finish* with cell *i*'s authority and gives it the channel's number *k*.

2. Sends the *Finish* agent to the place where the stationary agent *Directory* is residing in MSC.

3. The *Finish* agent meets the *Directory* agent and asks the service.

4. The *Directory* agent first checks the authentication and authorization of the *Finish* agent, if they are not valid, then the *Directory* agent refuses to provide service. If the authentication and authorization are valid, then the *Directory* agent gets the released channel number *k* and cell number *i* from the *Finish* agent. Then the *Finish* agent terminates itself.

5. After while, the *Directory* agent checks if the waiting queue is empty.

   a) If the queue is empty, then the *Directory* agent updates the channel status table according to the information the *Finish* agent gives.

   b) If the queue is not empty, then the *Directory* agent checks the first mobile agent *Start* in the queue.

      - If this *Start* agent also comes from cell *i*, Directory agent invokes it and assigns the channel *k* directly to it and doesn't need update anything. Then the *Start* agent sends the result back and terminates itself.

      - If this *Start* agent is not from cell *i*, check if it is from the interfering cell of cell *i*.

      If the *Start* agent is not from the interfering cell of cell *i*, then let the *Start* agent continue sleeping and checks the second mobile agent in the queue, then follows the same procedure as the first one.

If the *Start* agent is from the interfering cell of cell *i*, then the *Directory* agent invokes it and gives its ACO table. The fields in the channel status table which related to this ACO table are locked. With the ACO table, the *Start* agent uses LP algorithm to analyze if the channel *k* can be assigned to the call. If yes, then the *Start* agent updates the ACO table. Then the *Start* agent goes to meet the *Directory* agent and returns the ACO table. If the channel is available through channel-reassignment, the *Directory* agent sends a request to the corresponding cell to reassign the call currently using that channel to another channel. And the *Start* agent sends the result back and then terminates itself. In the meantime, the *Directory* agent updates the information in the total channel status table according to the updated ACO table and then the fields related to this ACO table are unlocked.

If no channel can be assigned to the *Start* agent, then the *Start* agent returns the ACO table to the *Directory* agent. And the fields related to this ACO table are unlocked. Let the *Start* agent sleep in the queue again. If there exists another mobile agent sleeping in the queue, *Directory* agent checks it and follows the same procedure as the first one.

- If the channel *k* cannot be assigned to any mobile agent in the waiting queue, then the *Directory* agent updates the channel status table according to the information the *Finish* agent gives.

Our implementation is based on the Grasshopper platform. In order to show how our implementation is possible, in the next chapter, we give a brief introduction to Mobile Agent System Interoperability Facilities Specification (MASIF), and the platform (Grasshopper) used in our implementation, which is compliant to MASIF standard.

Chapter 5

MASIF and Grasshopper

Mobile agents are a relatively new technology that is fueling a new industry. Up to now, there exist many agent systems, which differ widely in architecture and implementation. Our research is carried out in the Grasshopper mobile agent environment which is compliant to Mobile Agent System Interoperability Facilities Specification (MASIF). In this chapter, we give a brief introduction to MASIF standard and the Grasshopper platform.

5.1 MASIF

5.1.1 Background

An important goal in mobile agent technology is interoperability between various manufacturer's agent systems. The differences among mobile agent systems prevent interoperability and rapid proliferation of agent technology, and has probably impeded the growth of the industry [GMD97], thus it would be nice to have a single standard that is as universally accepted.

Just as CORBA defines a standard for distributed object interoperability, standards are needed for a universal agent platform that would allow any server to accept and execute an agent from any vendor. Interoperability becomes more achievable if actions such as agent transfer, class transfer, and agent management are standardized. Object Management Group (OMG) is currently working on an agent standard in the form of a CORBA common facility. The resulting standard will specify language-independent

interfaces for dealing with agents, but will probably not go as far as specifying any particular mobile code implementation.

One of the most promising candidates for mobile agent standard is GMD FOKUS's MASIF proposal. The MASIF standardization is a joint submission of GMD FOKUS, International Business Machines Corporation, Crystaliz, General Magic, and the Open Group in 1997. It is based on Java and built on top of the OMG Common Object Request Broker Architecture (CORBA), thus providing the integration of the traditional client/server paradigm and mobile agent technology. Our platform Grasshopper is the first intelligent mobile agent environment that is compliant to the MASIF standard. In the next two subsections, we show the advantages using Java in mobile agent implementation and the services provided by CORBA in MASIF standardization. After that, we describe two important interfaces contained in MAF module. Due to the methods of these two MASIF-compliant interfaces are not optimized for the Grasshopper environment, in Section 5.2.2.4, we will introduce two Grasshopper - specific interfaces.

5.1.2 Advantages Using Java in Mobile Agent Implementation

Java is an interpreted, object-oriented language and library set. Its main features are [URL11]:

- simple and familiar object-oriented language, which facilitates the definition of clean interfaces to promote the design of reusable software modules.
- architecture neutral, portable and robust system, which places emphasis on early error checking, and eliminates use of error prone programming features.
- interpreted and dynamic program execution, which permits application to adapt to an evolving environment by deferring binding of plug-in modules until runtime.
- a comprehensive security system, which enables construction of virus-free, tamper-free systems for network environments.

- multithreaded execution with synchronization between threads, which is useful in a multiprocessor system where threads run concurrently on separate processors, and improves program performance on single processor systems by permitting the overlap of input, output, or other slow operations with computational operations.
- support for distributed systems through the remote object invocation (RMI) and object serialization (OS) facilities, and offer extensive library of routines for coping with TCP/IP protocol.

A mobile agent is an active object that can move both data and functionality (code) to multiple places within a distributed system. A mobile agent should be able to execute in any machine within a network, regardless of the processor type or operating system. In addition, the agent code should not have to be installed on every machine that the agent could potentially visit. It should move with the agent's data automatically.

Therefore, it is desirable to implement agents on top of a mobile code system, such as the Java virtual machine (JVM). The Java Virtual Machine implements an abstraction layer to hide the underlying operating system and hardware architecture [URL11]. This abstraction is what insulates the built Java application from whatever system is hosting execution. This is unlike the tradition model where the application is built for a particular system. The built application is mapped directly to the particular system hosting execution. By shifting these system dependencies from the built application to the Virtual Machine, Java applications once built are inherently portable. It is the Virtual Machine, which must be ported to the host system, not each particular Java application. The dynamic nature of Java classes and objects, combined with advanced networking capabilities, make Java highly qualified for use as a mobile agent platform [Steven97].

Java and its run-time system produce a flexible and powerful programming system which supports distributed computing. An agent's classes are loaded at runtime over the network as it travels from one location to another. So we can see that Java is a

natural choice for implementing agent system because it is a mobile code platform with built-in support for networking.

5.1.3 CORBA Services

CORBA (Common Object Request Broker Architecture) is a specification of an architecture and interface that allows an application to make request of objects (servers) in a transparent, independent manner, regardless of platform, operating system, or local considerations. The CORBA ORB is an application framework that provides interoperability between objects, built in (possibly) different languages, running on (possibly) different machines in heterogeneous distributed environments [URL3].

The CORBA paradigm follows two existing methodologies: distributed client-server programming and object-oriented programming. The distributed computing is based on message-passing systems found in most UNIX systems. In CORBA, features of object-oriented programming, such as encapsulation and inheritance, are used.

CORBA can provide the following services which are related to mobile agent technology (Figure 5.1) [GMD97]:

- *Naming service*: CORBA naming service binds names to CORBA objects. The resulting name-to-object association is called a naming binding, which is always related to a naming context. A naming context is an object that contains a set of name bindings in which each name is unique.

- *Lifecycle service*: CORBA life cycle service defines services and conventions for creating, deleting, copying and moving CORBA objects.

- *Externalization service*: CORBA externalization service provides a standardized mechanism for recording an object's state onto a data stream, and for restoring an object's state from a data stream. An agent system uses this service when it needs to serialize and deserialize an agent's state.

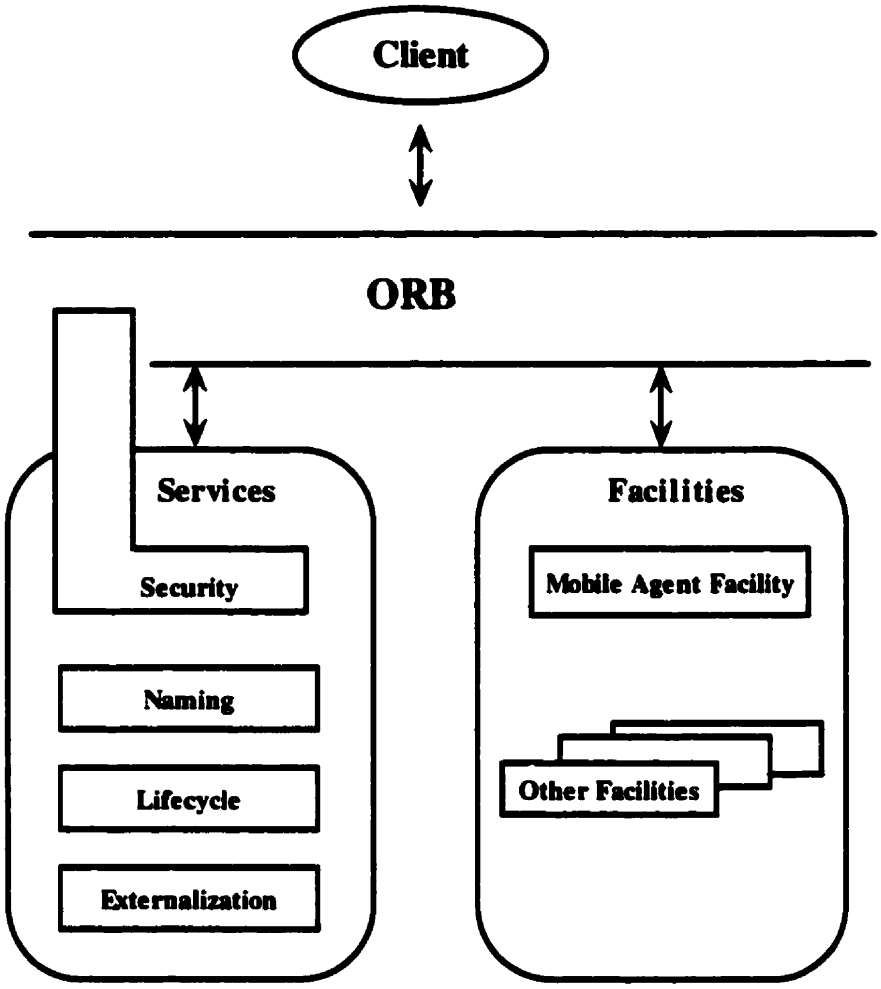

Figure 5.1 CORBA Services and Facilities

- *Security service*: although CORBA security does not currently meet all the needs of mobile agent technology, the mobile agent facility (MAF) implementation must use available CORBA security to satisfy its security needs. The security requirements for agents and agent systems in CORBA are:
    - Agent naming
    - Client authentication for remote agent creation
    - Mutual authentication of agent systems
    - Agent system access to authentication results and credentials

- Agent authentication and delegation
- Agent and agent system security policies
- Integrity, confidentiality, reply detection, and authentication

5.1.4 MAF Module

The Mobile Agent Facility (MAF) is a collection of definitions and interfaces that provide an interoperable interface for mobile agent systems. The MAF module contains two interfaces:

- MAFAgentSystem interface
- MAFFinder interface

The interfaces have been defined at the agent system level rather than at the agent level to address interoperability concerns.

5.1.4.1 MAFAgentSystem Interface

The MAFAgentSystem interface defines methods and objects that support agent management tasks such as fetching an agent system name and receiving an agent. These methods and objects provide a basic set of operations for agent transfer, including *receive*, *create*, *suspend*, and *terminate*.

5.1.4.2 MAFFinder Interface

The MAFFinder interface is a naming service. It provides methods for maintaining a dynamic name and location database of agents, places, and agent systems. The interface does not dictate what method a client uses to find an agent. Instead, it provides ways to locate agents, agent systems, and places that support a wide range of location technique. It defines operations for registering, unregistering, and locating agents, places, and agent system.

5.2 Grasshopper

Grasshopper is the first intelligent mobile agent environment, which is compliant to the MASIF standard. The standardization ensures that user's agent applications will be opened towards other agent environments and save the investments for the future. Grasshopper allows user to build agent-enabled distributed applications, which take advantage of local high-speed communication and local high-speed data access. Thus we chose to implement our algorithm in the Grasshopper platform.

5.2.1 Grasshopper Agent Environment

The Grasshopper environment consists of several agencies and a region registry, remotely connected via an Object Request Broker (ORB). The advantage of Grasshopper as an ORB-based agent platform is the integration of the traditional client/server paradigm and mobile agent technology. Due to the fact that Grasshopper is built on top of an ORB, the platform capabilities can easily be enhanced by simply accessing other (agent-based or non agent-based) ORB applications. For instance, a CORBA trading service can be integrated for finding agencies, places, or agents providing specific capabilities, or an event service can be used to enhance the low-level communication capabilities.

Several interfaces are specified to enable remote interactions between the distinguished distributed components. Apart from Grasshopper-specific interfaces, the MASIF-compliant interfaces (MAFAgentSystem and MAFFinder) are provided to enable interoperability between the Grasshopper platform and (MASIF-compliant) agent systems of different vendors. Figure 5.2 shows the Grasshopper environment [URL4].

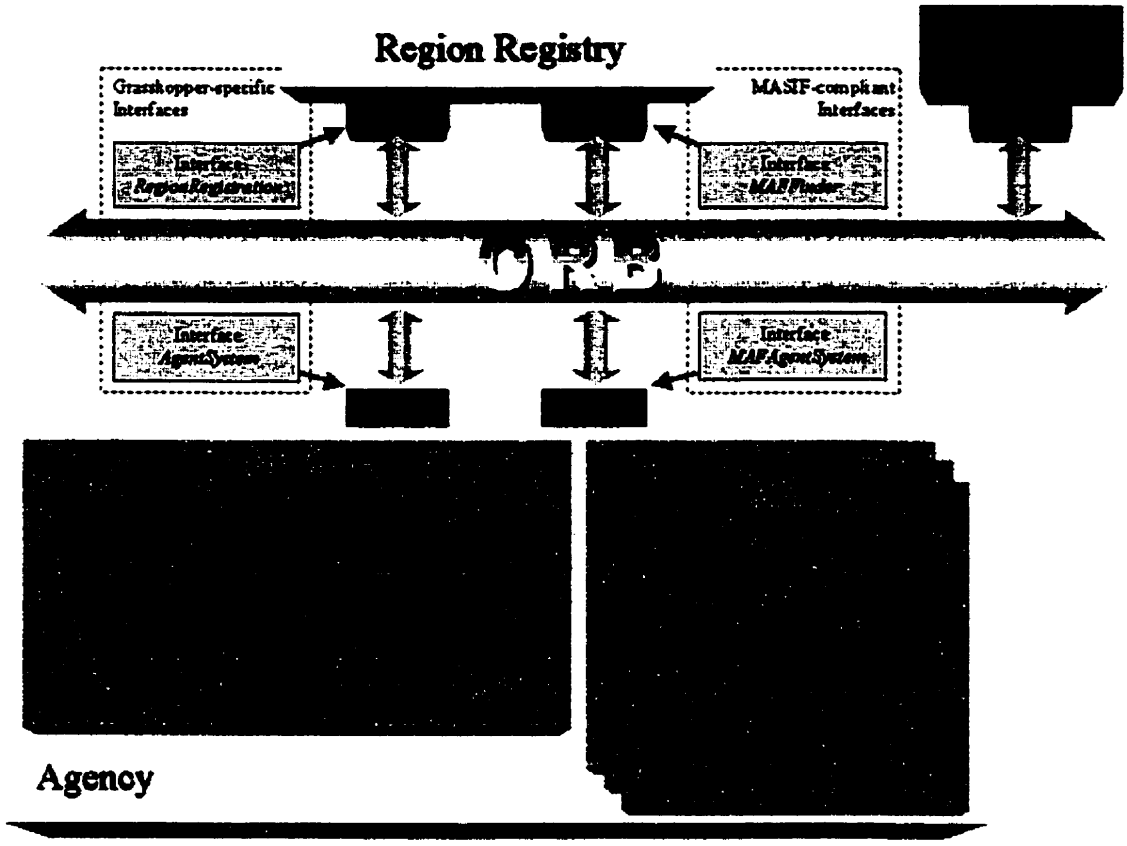

Figure 5.2 Grasshopper Environment [URL4]

5.2.2 Basic Components of Grasshopper Environment

5.2.2.1 Agency

Agencies are the actual runtime environments for mobile agents, consisting of a core agency and one or more places. Each agency runs on its own Java virtual machine. External interfaces are provided in order to enable the remote access of an agency via an ORB. Interface MAFAgentSystem is provided in order to enable a MASIF-compliant access, and the Grasshopper-specific interface AgentSystem offers sophisticated access to a Grasshopper agency.

The core agency provides only those capabilities that are inevitably required for the execution of agents. Agents access this functionality in order to retrieve information

about other agents, agencies or places, or in order to move to another location. Human users are able to monitor and control all activities within an agency by accessing the core services. Optionally, a Graphical User Interface (GUI) can be activated to facilitate user interactions. The entire content of an agency, i.e. the places as well as the agents residing in each place, can be monitored and controlled. The important core services are described as following:

Communication Service

This service enables asynchronous messaging and method invocation, and thus supports the communication between agents. Agents can contact each other in a location-transparent way. By contacting the region registry, the communication service is able to locate the agent to which a connection shall be established. If agents move away during communication, the new location is automatically updated.

Registration Service

Each agency must be able to know about all currently hosted agents and places for internal/external management purposes. The registration service is developed in order to deliver information about registered entities to hosted agents. Besides, the registration service of each agency is connected to the region registry which maintains information of agents, agencies and places in the scope of a whole region.

Management Services

Management services are developed to monitor and control agents and places of an agency by external (human) users. The following functionality is supported:

- create, remove, suspend, resume agents and places
- get information about specific agents and services
- list all agents residing in a specific place
- list all places of an agency

Apart from this, configuration management enables human users to specify system, trace, security, and communication properties.

Transport Service

This service supports the migration of agents from one agency to another. At the destination agency, the agent continues its task processing at that point where it has been interrupted before the migration. The transport service handles the externalization and internalization of agents, the localization of the destination agency, the connection establishment, and the transfer procedure itself.

Security Service

The Security Service provides authentication, privacy and integrity of inter-agent communication using Security Socket Layer (SSL) with Remote Method Invocation (RMI) or SSL with plain sockets. SSL is a strong and secure cryptographic method and state-of-the-art in many security-relevant applications, e.g. Web Browsers.

While the core services are tightly and statically integrated into the agency, additional services may be realized either by static components or by special agents, called *service agents*. In this way, the capabilities of the Grasshopper platform can be enhanced in a very flexible and comfortable way. The functionality realized by a service agent can be offered to other agents, applications, or human users. Also *mobile agents* offer functionality to other agents, services or human users. However, in contrast to service agents, mobile agents may move through the network in order to perform their task. For instance, a service agent or human user wants to collect information that is distributed throughout the network. In this case, a mobile agent can be created, which migrates from agency to agency, collects the desired information, and returns back to the initiator.

5.2.2.2 Region Registry

The region concept facilitates the management of the distinguished components in the Grasshopper environment. Agencies, as well as their places, can be associated to a specific region. Several agencies can be grouped to one region represented by one region registry. Each agency automatically registers each currently hosted service agent and mobile agent within the region registry. If an agent moves to another location, the corresponding registry information is automatically updated.

The region registry is realized by means of a Java program running on its own Virtual Machine. Agents may contact the region registry in order to locate other agents, services, places, or agencies. On the other hand, human users (administrators) are able to track their agents in the scope of the whole distributive environment by contacting the region registry.

As an agency, also a region registry provides interfaces that enable remote access to the offered functionality. The MASIF-compliant interface MAFFinder is provided in order to enable MASIF-compliant access, and the Grasshopper-specific interface RegionRegistration also can be used to offer a sophisticated access.

5.2.2.3 MASIF -- Compliant Interfaces

As we introduced in Section 5.1.4, MASIF contains two interfaces: interface MAFAgentSystem and interface MAFFinder. The interface MAFAgentSystem is associated to single agencies, and provides the following methods:

- create/suspend/resume/remove agent
- receive agent
- list hosted agents
- list available services
- get agent state
- get MAFFinder reference

The interface MAFFinder allows the registration and de-registration of agencies, places, and agents in the scope of a region. Additionally, methods are provided to search for specific agencies, places, and agents.

Because the methods of these two MASIF-compliant interfaces are not optimized for the Grasshopper environment, a Grasshopper agency also can be accessed via the Grasshopper-specific interfaces which have been designed especially for this platform, and therefore allows to access the offered functionality in the most efficient way.

5.2.2.4 Grasshopper – Specific Interfaces

The Grasshopper-specific interfaces include interface AgentSystem and interface RegionRegistration. In contrast to the interface MAFFinder and MAFAgentSystem, these two interfaces have been especially designed for the Grasshopper platform.

Apart from the functionality offered by MAFAgentSystem, the AgentSystem interface provides methods for sophisticated place management, i.e. the creation, suspension, resumption, and removal of places within an agency.

Apart from the functionality offered by MAFFinder, the RegionRegistration interface provides mechanisms for "freezing" an agent within a specific agency. Additionally, sophisticated search operations are comprised.

5.2.3 Agent Programming Guide On Grasshopper

The following two subsection comprise some information that is need for the implementation of a Grasshopper-compliant mobile agent.

5.2.3.1 Agent Methods

Grasshopper-compliant mobile agents are entirely implemented in Java, realized by means of at least one Java class. Each concrete agent has to be derived from the abstract class Agent which is part of the platform class library, and which has to be provided by each agency during its runtime. This abstract class comprises various

methods that realize the only "bridge" between an agent and its environment. These methods can be subdivided into two groups [URL4]:

- Modifiable methods: Several methods of the abstract class Agent must or may be re-implemented by the agent programmer. For instance, the method `live` has to be re-implemented since it specifies the actual task of the agent, whereas the method `createDescription` can optionally be overridden in order to specify a textual description of the agent's task.

- Non-modifiable methods: During its execution, an agent must be able to access the capabilities of the core agency. Thus, the class `de.ikv.grasshopper.agency.Agent` provides various methods. That means, a concrete agent does not get a reference of any component of the core agency, but instead invokes methods of its own superclass. These methods are declared *final* which means that they cannot be modified or re-implemented by agent programmers.

Some modifiable and non-modifiable agent methods that will be re-implemented or used in our algorithm will be introduced in the Section 6.1.

5.2.3.2 Inter-Agent Communication

Agents must be able to communicate with each other in order to exchange information. Especially, mobile agents must be able to access the offered functionality of service agents. The mechanism for inter-agent communication is provided by the communication service.

An agent that likes to make use of the communication service first has to create a local proxy object associated with the agent that should be contacted. All communication (method calls on the contacted agent) is done via that proxy object thus achieving location transparency. Once the proxy object is created, the agent can use all public methods of the associated agent. The communication service takes care of contacting the desired agent itself and determines the underlying communication protocol. The Figure 5.3 illustrates this scenario where Agent A is contacting Agent B.

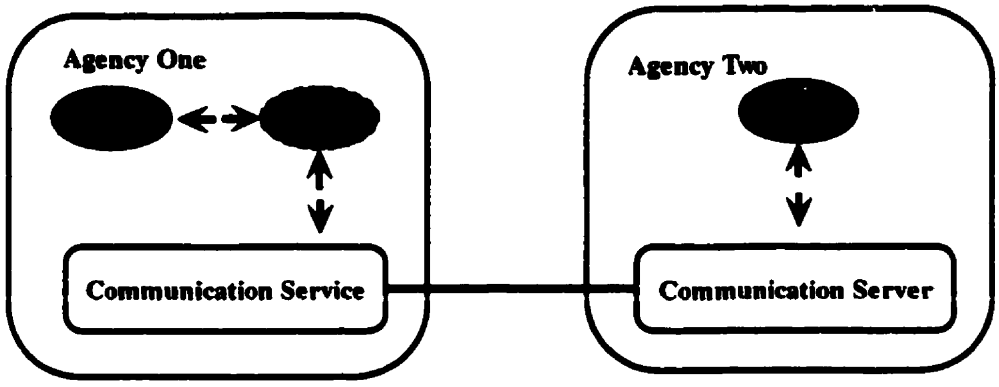

Figure 5.3 The Usage of Proxy Objects for Dynamic Method Invocation

In this chapter, we have introduced our platform - Grasshopper. We will show how to realize our algorithm on Grasshopper in the next chapter. And further more, we will analyze the performance of our proposed model through a simulator written by Java.

Chapter 6

Implementation and Performance

In this chapter, the implement detail on Grasshopper will be given in Section 6.1. And in Section 6.2, we will analyze the performance of our algorithm through a simulated cellular network consisting of 144 hexagonal cells with equal size arranged in a 12x12 grid.

6.1 Implementation of Our Algorithm on Grasshopper

In our algorithm, we have two kinds of service agents named the *Base* agent and the *Directory* agent. We also have two kinds of mobile agents named the *Start* agent and the *Finish* agent. Their tasks have been described in Section 4.3. In order to realize these tasks, we need implement four agents class: `BaseAgent`, `DirectoryAgent`, `StartAgent` and `FinishAgent`. Each agent has to be a subclass of the common `Agent` class. For instance, with the `BaseAgent` class, we have:

public class BaseAgent extends de.ikv.grasshopper.agency.Agent

And as we introduced in Section 5.2.3, several methods of the abstract class `Agent` must or may be re-implemented. In our implementation, we will re-implement `live()`, `init()`, `createName()`, `isMobileAgent()`(only for service agents) methods.

Figure 6.1 shows the diagram of our agent classes relationship according to Unified Modeling Language (UML). UML is a general-purpose notational language for specifying and visualizing complex software, especially large, object-oriented projects

[URL13]. A class is drawn as a solid-outline rectangle with three compartments separated by horizontal lines [URL14]. The behavior of a class is represented by its operations. The structure of a class is represented by its attributes. Relationships provide a pathway for communication between objects.

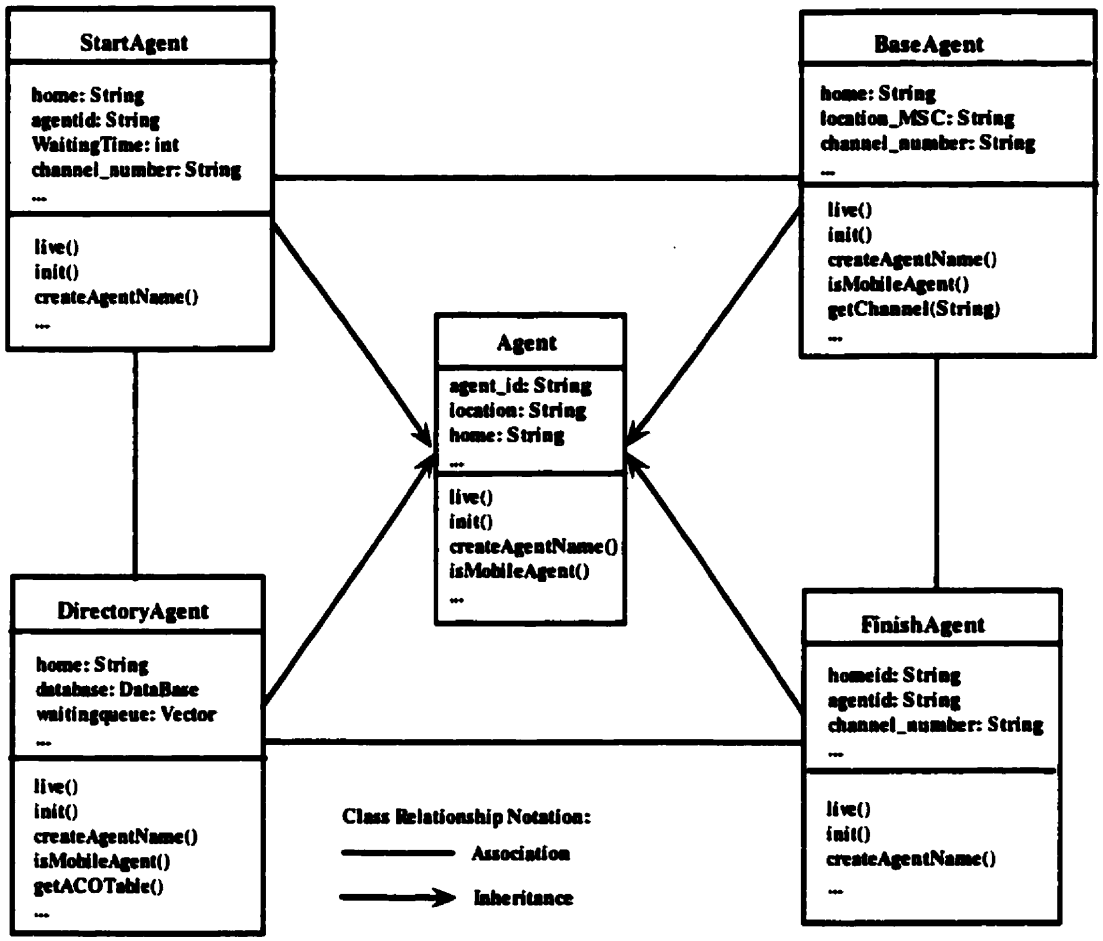

Figure 6.1 Diagram of Our Agent Classes Relationship in UML

In the above figure, there exist two types of relationships between classes: *inheritance* and *association*. Inheritance is a relationship between a superclass and its subclasses. Association is a bi-directional connection between classes which is shown as a line connecting the related classes [URL14]. In order to show how these agent classes communicate with each other, in the following subsection, we will introduce each agent class in detail.

6.1.1 BaseAgent Class

`BaseAgent` class is a subclass of common Agent. It declares the following objects:

```
private String home; //this base station's location
private String homeid;  //the identification of this base station
private String place;  //the current place of the BaseAgent
private String channel_number; //contains the number of the channel
private String agentid;  //the identification of created mobile agent
private String location_MSC; //the location of MSC
private String host_MSC; //the host name which MSC is residing
private String place_MSC; //the place name which MSC is residing
```

It includes the following methods:

- `createAgentName()`: This method defines an individual name for *Base* agent.

```
public String createAgentName(){
        return new String("BaseAgent");
}
```

- `isMobileAgent()`: This method indicates whether an agent is a mobile agent or a service agent. By default, agents are indicated as mobile agents (return value true). Thus, this method has to be overridden for each service agent.

```
public boolean is MobileAgent(){
        return false;
 }
```

- `init()`: This method allows to initialize the *Base* agent before the actual task processing is started.

```
public void init(){
```

```
    //initialize the channel_number to null at first
     channel_number=null;
   /*get this base station's location. getServiceinfo() method
     returns information about the service represented by the agent */
     home=getServiceinfo().getServiceRuntimeRelated().serviceLocation;
   //get the id of this base station
     homeid=getAgencyId();
   //get this BaseAgent current place
     place=de.ikv.grasshopper.util.LocationAssistant.getPlace(home);
   //get the MSC's location
     location_MSC=de.ikv.grasshopper.util.LocationAssistant.
                          createLocation(host_MSC, "MSC", place_MSC);
}
```

- `live()`: This method is the most fundamental method of each agent, since it specifies the agent's behavior.

```
public void live(){
   //initialize the information of mobile agent which will be created
    AgentInfo  info=null;
   /*if BaseAgent receives a signal from a mobile, getSignal(mobile)
    method analyzes the signal  and return the message.*/
    String message=getSignal(mobile);
   //If message is "CallArrived", creates Start agent  and send to MSC
    if(message=="CallArrived") {
      try{
       /*create a StartAgent in the current place. createAgent method
         enables an agent to create another agent in the same or another
         place of the current agency.*/
         info=this.createAgent("crim.ca.ymzhang.StartAgent",
                                        "file://", place, null);
```

```
        }
        catch(Exception e){};
      //get the id of the created agent
        agentid=info.getServiceID().toString;
      /*create a proxy of the StartAgent so that any public methods in
        StartAgent class can be accessed by BaseAgent class.*/
        StartAgentP  startagent=new StartAgentP(agentid, homeid);
      //invoke its move() method, startagent will migrate to the MSC
        try{
            startagent.move()(location_MSC);
          }
        catch(Exception e){};
      } //end if(message="CallArrived")
  //if message is "CallFinished", creates Finish agent  and send to MSC
   else if (message=="CallFinished")
      try{
        //create a FinishAgent in the current
          info=this.createAgent("crim.ca.ymzhang.FinishAgent",
                                                  "file://", place, null);
        }
      catch(Exception e){};
    //get the id of the created agent
      agentid=info.getServiceID().toString;
    /*get the returned channel number according the mobile ID. Here
      Mobile[mobileid] is an Object which containing the channel
      number it is currently using.*/
     String release_channel=Mobile[mobileid].channel_number;
    /*create a proxy of the FinishAgent so that BaseAgent class can access
      any public methods in FinishAgent class.*/
     FinishAgentP  finishagent=new FinishAgentP
```

```
                                        (agentid,homeid,release_channel);
        //invoke its move() method, finishagent will migrate to MSC
         try{
            finishagent.move()(location_MSC);
          }
         catch(Exception e){};
        } //end else if(message=="CallFinished")
      else
        System.out.println("invalid message!");
    }
```

- `getChannel(String channel_number)`: This method is used to receive the result from StartAgent in the remote place (MSC).

```
    public void getChannel(String channel_number){
      //if the channel_number is null, the call is blocked
       if(channel_number==null)
           System.out.println("This call is blocked");
      //else,, let this call use the channel indicated by channel_number
       else
          Mobile[mobileid].channel_number=channel_number;
      }
```

6.1.2 DirectoryAgent Class

`DirectoryAgent` class is a subclass of common Agent. It declares the following object:

```
    private String home;  //the home location
    DataBase  database; /*a reference of DataBase object. DataBase is an object
                          which contains all the channels' occupation information.
                          It only can be accessed by Directory agent.*/
```

```
ACOTable acotable; /*a reference of ACOTable object. ACOTable is an object
                     which contains the channel occupation information of
                     specific interference cells.*/
Vector waitingqueue; //the list for waiting queue
```

The `DirectoryAgent` class includes many methods. Among them, some methods are used to deal with DataBase which is not a main issue in our research. Thus, we will not show the implementation details of these methods here. The implementation of these methods can refer to our simulator (Appendix B). We only describe the following important methods:

- `createAgentName()`: This method defines an individual name for *Directory* agent.

```
public String createAgentName(){
    return new String("DirectoryAgent");
}
```

- `isMobileAgent()`: This method indicates whether an agent is a mobile agent or a service agent.

```
public boolean is MobileAgent(){
    return false;
}
```

- `init()`: This method allows to initialize the *Base* agent before the actual task processing is started.

```
public void init(){

  /*initialize channel status Database using intial_DataBase()
    database=new DataBase();
   initial_DataBase();
     //get the home location
```

```
        home=getServiceinfo().
                    getServiceRuntimeRelated().serviceLocation;
    }
```

- `live()`: This method specifies the *Directory* agent's behavior.

```
    public void live(){
        //do nothing in this method
    }
```

- `authentication(String agentid)`: This method is used to check if the agents valid or not.

```
    public boolean authentication(String agentid){
      /*We suppose in MSC, there exists a database containing all valid agent
        ID. Checking agentid in ID database by using method check_ID_
        DataBase(String), if found, return true, else return false.*/
        boolean found=check_ID_DataBase(agentid);
        return found;
      }
```

- `getACOTable(String agentid, String homeid)`: This method is used to provide the related ACO table for a specific *Start* agent.

```
    public ACOTable getACOTable(String agentid, String homeid){
      /*first check if the agent is valid by invoking authentication method. If
        the agent is valid, then providing the service to this agent.*/
        if(authentication(agentid)){
          /*check DataBase if the field relating to its ACO table is locked. Wait
            until the related fields are unlocked.*/
            while(isLocked_DataBase(homeid){}
          /*Extract ACO table from DataBase for this agent according to the
            homeid by using method getTable_DataBase(String).*/
```

```
        acotable=getTable_DataBase(homeid);
      /*Then lock the fields which related to this ACO table by using
        lock_DataBase(ACOTable), and return acotable.*/
        lock_DataBase(acotable);
        return acotable;
      }
    else
      //if the agent is not valid, then refuse to provide service
      System.out.println("This agent is not valid");
  }
```

- `returnACOTable(ACOTable acotable)`: This method is used to modify the channel status database according to the returned acotable and unlock the fields which related to this acotable.

```
public void returnACOTable(ACOTable acotable){
    //if the ACO table is changed, then change the DataBase accordingly
     if(ismodifed(acotable))
          modify_DataBase(acotable);
   //unlock the fields in DataBase which related to this ACO table
     unlock_DataBase(acotable);
  }
```

- `getReleasedChannel(String homeid, String agentid, String channel_number)`: This method is used to get released channel from *Finish* agent. And process our channel release procedure.

```
public void getReleasedChannel
          (String homeid, String agentid, String channel_number){
   /*first check if the agent is valid by invoking the authentication method.
     if the agent is valid, then providing the service to this agent.*/
     if(authentication(agentid)){
```

```
/*check if the waiting queue is empty. If it is not empty, check if this
  channel can be assigned to the waiting call.*/
  if(waitingqueue.isEmpty()==false){
    //check the elements in the waiting queue one by one
     Enumeration waitingitem=waitingqueue.elements();
     while(waitingitem.hasMoreElements()){
       //get the first proxy of Start agent
        StartAgentP  item=(StartAgentP)waitingitem.nextElement();
      /*if the item is from the same cell as FinishAgent, then awake
       the item, set the flag in item equal to 1 so that this Start agent
       can return the result. Then stop checking the waiting queue.*/
       if(item.homeid==homeid){
          notify(item);
        //inform the item which channel is released
          String  item.releasedchannel=channel_number;
          item.flag=1;
          break;
         }
      /*if the item is from the interference cell of  FinishAgent, then
       awake it, and set the flag in this item to 2 so that this Start
       agent can proceed the computation.*/
       if(isNeighber(item.homeid, homeid)){
          notify(item);
        //inform the item which channel is released
          String  item.releasedchannel=channel_number;
          String  item.releasedhomeid=homeid;
          item.flag=2;
        //wait until item finish its computation
          while(item.finished==false){};
       /*if the channel can be assigned to the item, break; Otherwise,
```

```
                    continue to check next element in the queue.
                    if(item.channel_number)
                        break;
                   } //end if(isNeighber(item.homeid, homeid))
                  } //end checking the waiting queue
                 } //end if(waitingqueue.isEmpty()==false)
               /*if the waiting queue is empty at first or this channel can not be
                 assigned to any waiting call, return this channel to Database.*/
                 returnChannel_DataBase(channel_number,homeid);
               } //end if(authentication(agentid))
             else
                System.out.println("This agent is not valid");
          }
```

6.1.3 StartAgent Class

`StartAgent` class is a subclass of common Agent. We declare the following variables in this class.

```
private String home;  //the location of the cell which this start agent comes from
private String  homeid;  //identification of the cell where the Start agent comes
private String  agentid:  //identification of the Start agent
private String  channel_number;  //the returned channel number
private ACOTable  acotable;  //the ACO table which the start agent will analyze
private int WaitingTime;  //the time for a call waiting in the waiting queue
```

Its constructor defines as:

```
public StartAgent(String agentid, String homeid){
        this.homeid=homeid;
        this.agentid=agentid;
    }
```

The class includes the following methods:

- `createAgentName()`: This method defines an individual name for *Start* agent.

```
public String createAgentName(){
    return new String("StartAgent");
}
```

- `init()`: This method allows to initialize the *Start* agent before the actual task processing is started.

```
public void init(){
    //get home's location
     home=getServiceinfo().getServiceRuntimeRelated().serviceLocation;
    //set 3 seconds for a Start agent waiting in the waiting queue
     WaitingTime=3000;
}
```

- `live()`: This method specifies *Start* agent's behavior.

```
public void live(){
//if this agent is Start agent, then get ACO table from Directory agent
 if(getName()="StartAgent"){
  /*create a proxy of Directory agent so that Start agent can access any
   public methods in Directory agent.*/
   DirectoryAgentP  directoryagent=new DirectoryAgentP();
   acotable=directoryagent.getACOTable(agentid, homeid);
  //analyze the ACO table according to our algorithm
   String channel_number=analyze_ACO(acotable);
 //return the ACO table to Directory Agent
  directoryagent.returnACOTable(acotable);
/*if no channel available, sleep in the waiting queue until invoked by
```

```
Directory agent. If waiting time is expired, then terminates itself.
The call is finally blocked.*/
if(channel_number==null){
 //add the proxy of this Start agent to waiting queue in Directory agent
  directoryagent.waitingqueue.addElement(new StartAgentP());
  String  releasedchannel; //the channel number which will be released
  String  releasedhomeid; //the cell ID which will release a channel
  while(time<=WaitingTime){
    try{
       java.lang.Thread.sleep();
       }
  /*invoked by Directory agent when a released channel is from the
    same or interference cell with Finish agent.*/
    catch(Exception e){
     /*set a flag to decide if Start agent need to analyze ACO table.
       First set flag 0. If flag is changed to 1 by Directory agent, that
       means the released channel is from the same cell as this Start
       agent. Directory agent can assign this channel directly to this
       Start agent, no need to modify DataBase. Then the Start agent
       sends the results back and terminates itself. If flag is changed to
       2 by Directory agent, that means the channel is from the inter-
       ference cell, need to analyze the ACO table to decide if this
       channel can be used.*/
       int  flag=0;
       boolean finished=false;
      while(flag==0){};
     //if flag=1, assign the released channel to this Start agent
          if(flag==1){
        channel_number=releasedchannel;
        break;
```

```
                }
                //if flag=2, then get ACO table from Directory agent
                acotable=directoryagent.getACOTable(agentid, homeid);
                /*analyze the ACO table according to our algorithm. The method
                reanalyze_ACO is used to analyze ACO table after clearing
                the released channel in the base station with releasedhomeid.*/
                channel_number=reanalyze_ACO
                                (acotable,releasedchannel,releasedhomeid);
                //inform Directory agent the analyzation is finished
                finished=true;
                //return the ACO table to Directory Agent
                directoryagent.returnACOTable(acotable);
                //if there is channel available, break; Otherwise, continue to sleep
                if(channel_number!=null)
                    break;
                } //end catch
            } //end while. Waiting time is expired.
        } //end if(channel_number==null)
        /*if the time is expired or there exists channel available, send the result
        back and terminates itself.*/
        /*create a proxy of BaseAgent so that Start agent can access any public
        methods in BaseAgent class.*/
        BaseAgentP baseagent=new BaseAgentP();
        baseagent.getChannel(channel_number); //send the result back to home
        remove(); //this method removes the agent from agency
        } //end if(getName=="StartAgent")
        else
        System.out.println("This is not Start Agent");
}
```

6.1.4 FinishAgent Class

`FinishAgent` class is a subclass of common Agent. It declares the following objects:

```
private String channel_number; //the released channel number
private String homeid; //identification of the cell where the Finish agent comes
private String agentid: //identification of the Finish agent
```

Its constructor is defined as:

```
public FinishAgent(String agentid, String homeid, String channel_number){
        this.channel_number=channel_number;
        this.homeid=homeid;
        this.agentid=agentid;
 }
```

`FinishAgent` class includes the following methods:

- `createAgentName()`: This method defines an individual name for *Finish* agent.

```
public String createAgentName(){
        return new String("FinishAgent");
 }
```

- `init()`: This method allows to initialize the *Finish* agent before the actual task processing is started.

```
public void init(){
    //get home's location
     String home=getServiceinfo().
                        getServiceRuntimeRelated().serviceLocation;
```

```
    }
```

- `live()`: This method specifies *Finish* agent's tasks.

```
public void live(){
  /*if this agent is Finish agent, then return this channel to Directory
    agent, then Finish agent terminates itself.*/
    if(getName()="FinishAgent"){
      /*create a proxy of Directory agent so that Finish agent can access
        any public methods of Directory agent.*/
          DirectoryAgentP  directoryagent=new DirectoryAgentP();
        //return the channel to Directory agent
          directoryagent.getReleaseChannel
                                  (homeid, agentid, channel_number);
          remove(); //remove this agent from the agency
    }
    else
          System.out.println("This is not Finish Agent");
}
```

6.2 Performance of Our Algorithm

In this section, a simulation program is written to evaluate the call blocking performance of the proposed algorithm with different waiting time. Specifically, in addition to its performance under uniform traffic, we examine its ability to alleviate congestion in the hot spots of a cellular system.

As reference cases, we use the performances of *Simple FCA Scheme*, *First Available DCA Scheme* and *Locking Packing (LP) Scheme* in the same situation. As we introduced in the Chapter 2, the *Simple FCA Scheme* allocates the same number of nominal channels to each cell. And the *First Available DCA Schemes* assigns the first available channel within the reuse distance encountered during a channel search to the call. In the Section 4.2, we have described the main idea of LP algorithm in detail.

6.2.1 Simulation

Our study is based on the following assumptions [David93][Ming89]:

1. All base-station transmitter power levels are the same in the absence of power control.

2. The radio link is assumed to be free from noise and fading. So there is only power loss in the radio signal due to propagation.

3. All portables and base stations have ideal homogeneous omni-directional antennas.

4. The channel assignment is made only for the radio links from mobiles to base stations. Also the channel assignment is made for *snapshots* of the system, where a *snapshot* is the set of mobiles in the system frozen in their positions, at some instant of time.

5. In each cell, calls originate at a random position. The simulation assigns the base station nearest to the mobile making a call attempt to provide service using a channel available at the base station.

6. The call arrives according to the Poisson distribution [Appendix A], and the call duration is exponentially distributed.

For the comparison purpose, our simulated cellular network is the same as the one in [Chih93], which consists of 144 hexagonal cells with equal size arranged in a 12x12 grid. In order to avoid the boundary effect, the 144 cells in our simulation are organized as a 12x12 array with wrap-around in both dimensions. Thus, the results are representative of an infinite system, and therefore apply to typical cells in a large network [Chih93]. Figure 6.2 shows the layout of the simulated 144-cell cellular network.

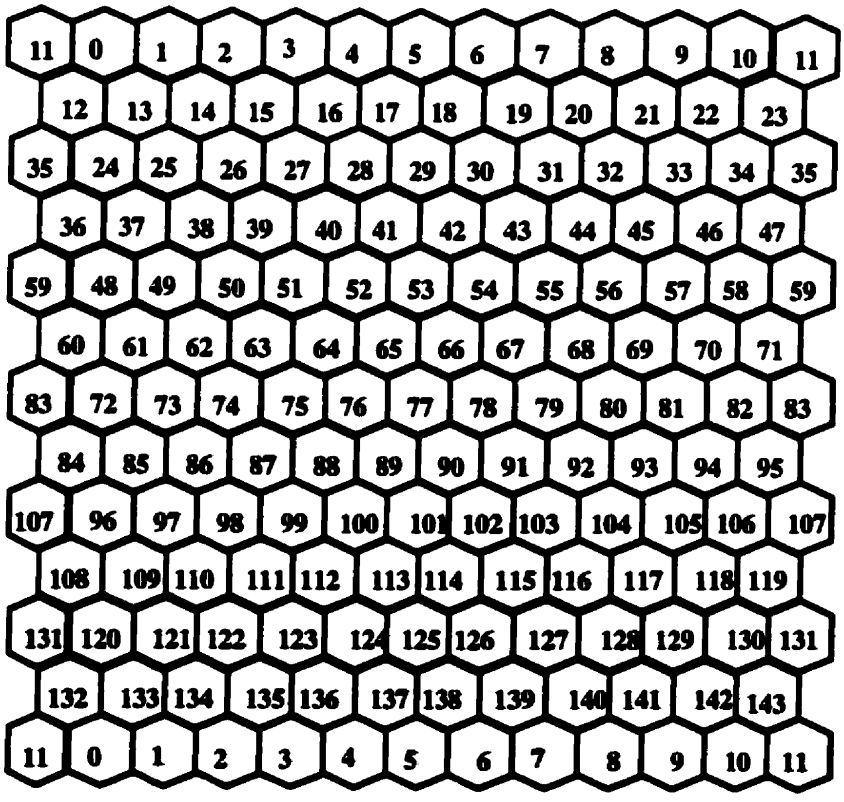

Figure 6.2 The Simulated 144-Cell Cellular Network Layout

Similar to current cellular system, the reuse constraint takes the form of two-cell buffering [Chih93]. That is, cells that use the same channel can not be either immediate or second-layer neighbors. Under this constraint, the reuse factor would be 7 when FCA is deployed.

In general, the total number of distinct channels available in the cellular system is typically in the range of 300-400 [Chih93]. In our study, we carry out the simulation of our algorithm, *Simple FCA* algorithm, *First Available DCA* algorithm and *LP* algorithm with 350 distinct channels in the cellular system, that is, for the *Simple FCA* system, there are 350/7=50 channels per cell.

The simulation was started initially with no calls in the system. The time required for stability was about 10 minutes. That means after 10 minutes, blocking probability in the first half hour (from 10 minutes to 40 minutes) almost equals the one in the second half hour (from 40 minutes to 70 minutes). Data were collected after stabilization for about 30 minutes.

Blocking is defined as the ratio of new call attempts blocked to new call attempts and does not include channel changes or forced call terminations at cell boundaries.

6.2.2 Uniform Traffic

In this section, we consider the case when all cells in the network have the same arrival rate. We simulate the performances under different traffic load, different waiting time for a call in the waiting queue and the different average duration time of calls. In the following subsection, we will analyze the effects resulted by different factors.

6.2.2.1 Performance under Different Traffic Load

The Figure 6.3 shows the average cell blocking rate of *Simple FCA Scheme*, *First Available DCA Scheme*, *LP Scheme* and our algorithm with 3 seconds waiting time in the queue as a function of traffic load per cell (calls/hour) when average duration time of calls is 1 minute.

Traffic load(calls/hour)	5000	5500	6000	6500	7000	7500	8000
Simple FCA	41.65%	48.08%	51.17%	55.58%	58.79%	61.08%	63.40%
First Available DCA	0.11%	3.59%	8.01%	13.87%	18.72	22.40%	26.82%
LP algorithm	0.00%	0.055%	0.37%	3.36%	7.57%	11.94%	16.59%
Our algorithm	0.00%	0.00%	0.04%	1.37%	4.81%	8.75%	14.25%

Figure 6.3 Blocking Comparison: Uniform Traffic

From the Figure 6.3, we can see that, our algorithm gives the lowest blocking probability under different traffic load, followed by *LP Scheme*, *First Available DCA Scheme* and the *Simple FCA Scheme*. Under moderate traffic conditions (6500 calls/hour), the blocking probabilities are 1.37%, 3.36%, 13.87% and 55.58% respectively. That means our algorithm with 3 seconds waiting time can decrease about 40 times, 10 times and 2.5 times blocking rate with respect to *Simple FCA Scheme*, *First Available DCA Scheme* and *LP Scheme*. From the Figure 6.3, we also can see that, under light and moderate traffic load, our algorithm can decrease much more blocking probability with respect to another three schemes, and under heavy traffic, our algorithm only outperforms *LP Scheme* with a small decrease in blocking probability, but it still performs much more better than *Simple FCA Scheme* and *First Available DCA Scheme*.

6.2.2.2 Performance under Different Waiting Time

In order to show how different waiting time for a call in the waiting queue affects the performance of our algorithm, we made simulations with different waiting time from 0 second to 10 seconds when the traffic load per cell is 6500 calls/hour when average duration time is 1 minute. Figure 6.4 shows the simulated result.

Waiting time(second)	0	1	2	3	4	5	6	7	8	9	10
Blocking probability(%)	3.36	2.60	1.81	1.37	0.95	0.65	0.46	0.32	0.20	0.13	0.09

Figure 6.4 Blocking Comparison with Different Waiting Time

From the Figure 6.4, we can see that the longer the waiting time, the less blocking probability is. From 0 second to 10 seconds, the blocking probability decreases from 3.36 to 0.09, that means, with 10 seconds waiting time, the blocking rate can decrease about 38 times with respect to the same algorithm without waiting queue. Regarding to the tolerance a mobile user can have, in the Section 6.3, we fix the waiting time to be 3 seconds when the average duration time is 1 minute.

6.2.2.3 Performance under Different Average Duration Time

Traffic (in Erlang) is a measurement of how "busy" a line is during a period of measurement. Traffic is calculated using the following formula [URL12]:

Traffic(in Erlangs) = Number of calls/hour x *Average duration time*

Thus, we can know that a large number of calls with a short average duration time will produce the same result as a small number of calls with a long average duration time.

The Figure 6.5 shows the average blocking probabilities of different algorithm under different traffic load per cell (calls/hour) with three seconds waiting time for our algorithm when average duration time of calls is 3 minutes.

Traffic load(calls/hour)	1800	2000	2200	2400	2600
Simple FCA	45.88%	50.98%	55.81%	59.38%	61.70%

First Available DCA	2.91%	8.90%	15.88%	21.68%	25.69%
LP algorithm	0.01%	0.56%	3.60%	8.78%	13.27%
Our algorithm	0.00%	0.14%	2.61%	7.65%	12.14%

Figure 6.5 Effect of Average Duration Time: Uniform Traffic

Comparing the Figure 6.3 to Figure 6.5, we can see that with the same blocking probabilities, all algorithms can carry about 3 times more calls per hour with 3 minutes average duration time than with 1 minute average duration time.

In order to show the effects of average duration time on the waiting time for a call in the waiting queue, we also simulate the performance of our algorithm with 3 minutes average duration time under different waiting time from 0 second to 10 seconds when the traffic load per cell is 2200 calls/hour. The Figure 6.6 shows the result:

Waiting time(second)	0	1	2	3	4	5	6	7	8	9	10
Blocking probability(%)	3.60	3.45	3.01	2.68	2.46	2.16	2.09	1.88	1.75	1.51	1.43

Figure 6.6 Effect of Average Duration Time: Different Waiting Time

From the Figure 6.6, we can see that the blocking probability decreases from 3.60 to 1.43 when the waiting time for a call in the waiting queue is changed from 0 second to 10 seconds, that means, with 10 seconds waiting time, the blocking rate can decrease about 2.5 times with respect to the same algorithm without waiting queue. Compared to the Figure 6.4, in which the blocking rate can decrease about 2.5 times with only 3 seconds waiting time with respect to the same algorithm without waiting queue, we can see that with the same blocking probability, the longer the average duration time is, the longer the waiting time for a call in the waiting call should have.

6.2.3 Traffic Hot Spots

Usually there are temporal and spatial variations in local traffic demands. The DCA Algorithms have great advantage in these situations due to flexibility in their channel assignment. Various patterns of traffic hot spots may be of practical interest [Chih93]. We particularly consider the isolated hot spots such as the Giant Stadium after a ball game (see Figure 6.7), the diagonal highway (see Figure 6.8) and the expressway around a metropolitan area during rush hour (see Figure 6.9).

We simulate the performances of these three layout by using *Simple FCA Scheme, First Available DCA Scheme, LP Scheme* and our algorithm with 1 minute average duration time and 3 seconds waiting time in the queue as a function of traffic load per shadow cell (calls/hour) while the traffic load in the other cells is 6200 calls/hour. The Figure 6.10, Figure 6.11 and Figure 6.12 show the simulated results.

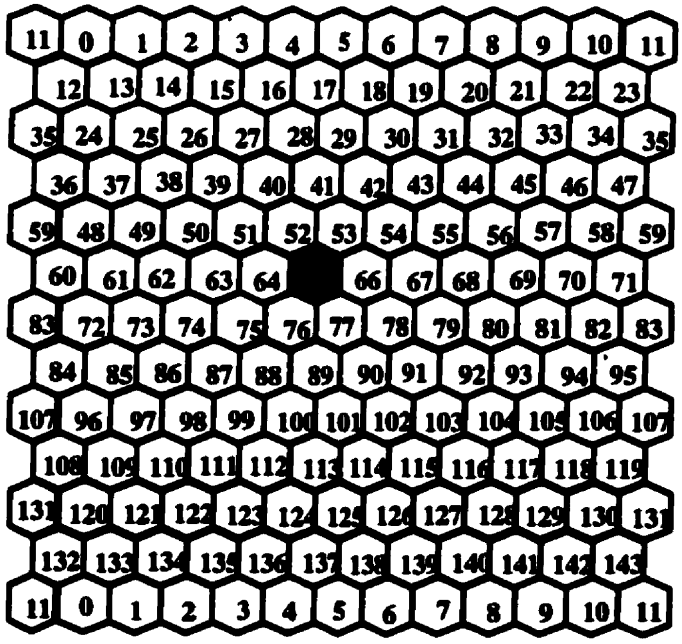

Figure 6.7 Traffic Hot Spots: Giant Stadium

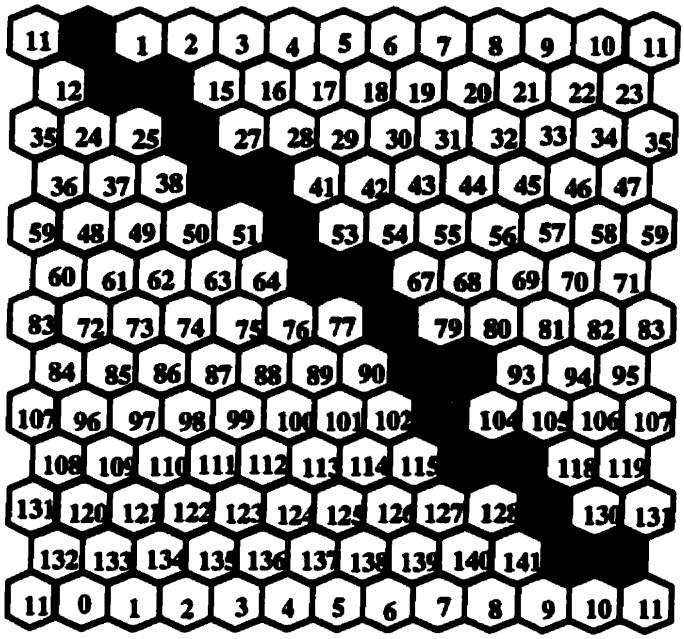

Figure 6.8 Traffic Hot Spots: Diagonal Highway

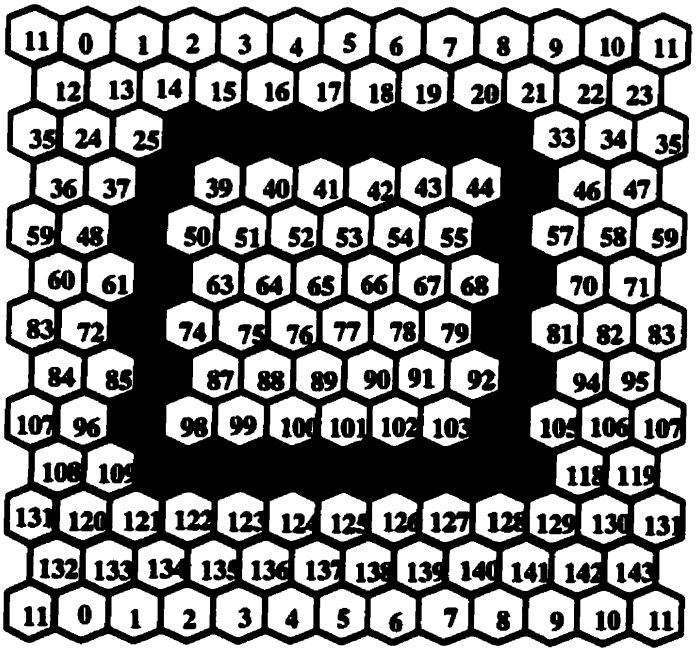

Figure 6.9 Traffic Hot Spots: City Beltway

Traffic load(calls/hour)	6400	6600	7000	7800	8600	9400	11000
Simple FCA	53.01%	53.22%	53.84%	55.08%	55.66%	56.21%	58.29%
First Available DCA	9.43%	10.13%	11.12%	13.07%	14.44%	15.16%	19.17%
LP algorithm	0.93%	1.27%	1.82%	3.41%	5.31%	6.91%	10.63%
Our algorithm	0.17%	0.41%	1.31%	3.05%	4.77%	6.43%	10.42%

Figure 6.10 Blocking Probability for Traffic Hot Spot: Giant Stadium

Traffic load(calls/hour)	6600	7400	8600	9800	11000	12600
Simple FCA	53.24%	54.03%	55.33%	56.42%	56.70%	58.69%
First Available DCA	10.07%	11.59%	13.32%	15.40%	16.61%	19.28%
LP algorithm	1.36%	2.51%	4.25%	6.75%	8.51%	11.22%
Our algorithm	0.35%	1.24%	2.82%	5.41%	7.53%	10.59%

Figure 6.11 Blocking Probability for Traffic Hot Spot: Diagonal Highway

Traffic load(calls/hour)	6600	7400	8200	9000	9800	10800
Simple FCA	53.60%	54.40%	55.49%	56.27%	57.44%	58.00%
First Available DCA	10.67%	12.30%	13.78%	15.15%	17.35%	18.99%
LP algorithm	1.46%	2.90%	4.60%	6.76%	9.06%	11.21%
Our algorithm	0.34%	1.63%	2.92%	5.34%	7.82%	10.61%

Figure 6.12 Blocking Probability for Traffic Hot Spot: City Beltway

From Figure 6.10, Figure 6.11 and Figure 6.12, we can see that our algorithm gives the lowest blocking probability under various traffic hot spots layout, followed by *LP Scheme*, *First Available Scheme* and *Simple FCA Scheme*. For example, in Diagonal Highway, when the traffic load in hot spots is increased 38% (8600 calls/hour), the blocking probabilities are 2.82%, 4.25%, 13.32% and 55.33% respectively. For Giant Stadium (Figure 6.10), Diagonal Highway (Figure 6.11) and City Beltway (Figure 6.12) layout, our algorithm can increase 78%, 103% and 74% traffic load in the hot spots respectively with only about 10% blocking probability. We can see from the given data that, with the light and moderate increase of the traffic load in the hot spots, our algorithm behaves much more better than another three algorithm. Under the heavy increase of traffic load in the hot spots, our algorithm only outperforms *LP Scheme* with a small decrease in blocking probability, but it still performs much better than *Simple FCA Scheme* and *First Available DCA Scheme*.

In this chapter, we have talked about the implementation and performance of our algorithm. In next chapter, we will summary our work and give the recommendation for future work.

Chapter 7

Conclusion

7.1 Contribution

In this research, a mobile-agent-based dynamic channel allocation with waiting queue algorithm was developed. This algorithm can give mobile users the choice to wait for a connection so that it improves the quality of service (QoS) in cellular network. That is, a user can accept some variations with respect to required QoS parameters.

From the viewpoint of computation and network resource, our model is based on Mobile Agent Paradigm that has great advantages on reducing network traffic, dealing with vast volumes of data and dynamic adaptation. Our study uses mobile agent to make dynamic decision and do the computation when it is in the remote destination. Thus, it makes our algorithm more efficient and competitive with respect to the traditional client/sever model.

We considered the Grasshopper mobile agent environment as our platform because it allows user to build agent-enabled distributed applications, which take advantage of local high-speed communication and local high-speed data access. Grasshopper is compliant to Mobile Agent System Interoperability Facilities Specification (MASIF) which is based on Java and built on top of the CORBA.

A simulation program also has been written to evaluate the performance of our proposed algorithm. Our simulator consists 144 hexagonal cells with 350 distinct channels which is typical channel number in the cellular system. We simulated the

performances under different traffic load, different waiting time for a call in the waiting queue and the different average duration time of calls. As reference cases, we also simulated the performances of *Simple FCA Scheme*, *First Available DCA Scheme* and *Local Packing Scheme* in the same situation.

The simulations showed that even with a few seconds waiting time for a call in the waiting queue, it maintains a favorable performance over *Simple FCA Scheme*, *First Available DCA Scheme* and *Local Packing Scheme* under uniform traffic load. The longer the waiting time for a call in the waiting queue, the less the blocking probability our algorithm has. The simulations also showed that with the same blocking probability under the same traffic load, the longer the average duration time is, the longer the waiting time should have.

From the viewpoint of practical interest, three typical patterns of traffic hot spots such as Giant stadium, Diagonal highway and City beltway were considered. Through the simulation, we can see that our algorithm also has lowest blocking rate with respect to the other three algorithms especially when the system has light or moderate traffic load. Under heavy traffic, our algorithm only outperforms LP algorithm with a small decrease in blocking probability, but it still performs much more better than *Simple FCA Scheme* and *First Available DCA Scheme*.

7.2 Future Work

The following recommendations are suggested for future improvements and further developments of our proposed algorithm:

- The algorithm could consider the priority of incoming calls. Each call has a priority factor, the call with higher priority should be served first if two calls come simultaneously. In the waiting queue, we could sort the calls by their priority factors and put the call with highest priority in the head of queue. This strategy can prevent losing some important calls and can let some urgent calls get connected as soon as possible.

- The algorithm could set dynamic waiting time instead of fixed waiting time for calls in the waiting queue. In this way, every mobile user can decide how long he can wait to get connected. This improvement can offer the mobile users more chance to state their requirement.

- In our proposed algorithm, we make assumption that the set of mobiles in the system is frozen in their positions at some instant of time. But practically, some mobiles can move from one cell to another and any active call needs to be allocated a channel in the destination cell. This event, termed the *handover* or *handoff*, deserves further consideration.

Appendix A

Poisson Distribution

The Poisson distribution governs the occurrence of random events in space or time. It assumes the distribution of the intervals elapsing between two consecutive requests is exponential. Usually the process is said to consist of discrete events occurring (with an exponential distribution) at a constant rate of L events/time. Then the probability of exactly N events occurring within a time interval t is [ULR7]:

$$\frac{(Lt)^N e^{(-Lt)}}{N!}$$

Waiting time distributions are based on the exponential distribution, which in turn is derived from the Poisson distribution.

Consider a problem such as the call request in a cellular system. The number of calls during any particular unit of time is governed by the Poisson distribution with mean L per unit time. We will consider the distribution of time intervals between successive call requests and find the probability that there is a time-interval of length t between successive call requests [URL8].

We divide t into increments dt in length such that there is a small probability p of the occurrence of a call request during dt, q is the probability of no occurrence of a call request in that interval, where $p+q = 1$. dt is assumed small enough to make the probability of more than one call request in dt negligible.

The probability, denoted by *dP*, that there are *n* incremental intervals between successive call requests is given by:

$$dP = pq^n$$

That is *n* intervals with no call requests, and in the (*n+1*)th interval we have a call request. So we have:

$$dP = p(1-p)^n$$

Now if there are *L* call requests per unit time, the mean number of call request in time *dt* is *Ldt*. So using the formula, *mean = number of trials* x *the probability of success at any trial*, we get:

$$Ldt = 1 \times p$$

This gives us $dt = p/L$. Because we can write $dt = t/n$. Equating expressions for *dt* we have $p/L = t/n$, that is $p = Lt/n$. Then we can get:

$$dP = L(1-Lt/n)^n dt$$

$$dP/dt = L(1-Lt/n)^n$$

As *n* -> infinity, the expression in brackets tends to *exp(-Lt)*. We can get:

$$dP/dt = Le^{(-Lt)}$$

The righthand side is the exponential distribution, and is the probability density function for the interval between successive events, i.e. it is the probability that the time interval lies between *t* and *t+dt*. The probability that a given interval is less than or equal to *t* is given by integrating. So we can get the cumulative distribution:

$$P(T<= t) = 1-e^{(-Lt)}$$

Appendix B

Source Codes

In this section, we list the important classes for simulators of Simple FCA Scheme, Fist Available DCA Scheme, LP Scheme and our proposed scheme.

1. Simple FCA Scheme

This program is written to calculate the blocking probability of Simple FCA Scheme. It includes class MSC, class BaseStation, class Call and class MyString. The program was tested using JDK 1.1.5 under Unix system.

```
/*************************************************************************/

package ca.mcgill.fix_channal;
import java.io.*;

/* ----------------------------------------------------------------------------------------------------------------------------

This is the controlling class used to calculate the blocking probability using Simple FCA algorithm. In this class, we only collect
the data between Start_time and End_time which given by the user.

 ----------------------------------------------------------------------------------------------------------------------------*/

class MSC{
  static int current_time;
  static BaseStation[][] basestation;
  static int block_num=0;   //counter for blocking calls
  static int total_call=0;      //counter for total calls
  static int Start_time,End_time;  //the start time and end time for data collecting

  //This is the constructor of class MSC, making some initialization
  public MSC(int total_chan_num){
      basestation=new BaseStation[12][12];
      for (int i=0;i<12;i++)
          for (int j=0;j<12;j++)
              basestation[i][j]=new BaseStation(i, j, total_chan_num/7);  //initialize the Base Stations
      current_time=0;
    }
```

```
//This is main class of whole package of Simple FCA algorithm.
public static void main(String argv[]){
    int Total_chan_num;
    String call_data="call.dat";
    if(argv.length<3) {
       System.out.println("Usage:  total_chan_num, start_time end_time");
       System.exit(1);}
    //read data from keyboard
    Total_chan_num=Integer.parseInt(argv[0]);
    Start_time=Integer.parseInt(argv[1])*60*100;
    End_time=Integer.parseInt(argv[2])*60*100;
    MSC msc=new MSC(Total_chan_num);
    try{
      FileReader reader=new FileReader(call_data);
      BufferedReader buf_reader= new BufferedReader(reader);
      String ln=null;
      while ((ln=buf_reader.readLine())!=null){
           //process the data read from call.data file
            String one_call[]=MyString.split(ln," ");
            current_time=Integer.parseInt(one_call[0]);
            if (one_call[4].charAt(0)=='c') {
             //only collect data between Start_time and End_time
                if(current_time>End_time) break;
                if(current_time>Start_time) total_call++;
                Call call=new Call(Integer.parseInt(one_call[1]));
                 basestation[Integer.parseInt(one_call[2])][Integer.parseInt(one_call[3])].MakeNewCall(call);}
           else {
                int callid=Integer.parseInt(one_call[1]);
                basestation[Integer.parseInt(one_call[2][Integer.parseInt(one_call[3])].ReleaseCall(callid);
                }
         } //end while
      buf_reader.close();
      } //end try
     catch(IOException e){
          System.err.println(e);
          System.exit(1);
     }
   System.out.println("The block probablity is"+(double)block_num/(double)total_call*100.0);
  }
}

/*-------------------------------------------------------------------------------------------------------------------------------------------

This class is used to calculate the blocking probability of Simple FCA Scheme. It includes MakeNewCall(Call),
AllocateNewChannel(), FindNewChannel(), TakeNewCall(Call,int) and ReleaseCall(int) methods.

-------------------------------------------------------------------------------------------------------------------------------------------*/

class BaseStation{
     int id,matrix_x, matrix_y; channal_num; //cell number and channel number
     Call[] channal_status;

     //constructor of class BaseStation to do some initialization
     public BaseStation(int x, int y, int channal_num){
          matrix_x=x;
          matrix_y=y;
          id= x*100+y;
          this.channal_num=channal_num;
          channal_status=new Call[channal_num];
   }

   //This method is used to allocate new channel to the base station.
   int AllocateNewChannal(){
           for(int i=0;i<channal_num;i++){
             if (channal_status[i] == null){
               return i;  // If there exist channel, return the channel number.
```

```
                }
            }
        return -1; //otherwise, return -1
    }

    //This method is used to calculate the blocking number for the calls.
    void MakeNewCall(Call call){
        int newchannal=FindNewChannal();
          if(newchannal==-1){
                if(MSC.current_time>MSC.Start_time)
                    MSC.block_num++;
                return;
            }
        TakeNewCall(call, newchannal);
      }


    //This method is used to decide if a empty channel can be found
    int FindNewChannal(){
          for (int i=0;i< channal_num;i++){
              if( channal_status[i]==null)
                  return i; //if found, resurn the channel number
              }
           return -1;  //otherwise, return -1
      }

    //This method shows how to take a new call
    void TakeNewCall(Call call, int newchannal){
            channal_status[newchannal]=call;
       }

    //This method shows the procedure how to release a call
    void ReleaseCall(int callid){
            for(int i=0;i<channal_num;i++){
                if(channal_status[i]!=null && channal_status[i].callid==callid)
                    channal_status[i]=null;
            }
     }
  }

/*----------------------------------------------------------------------------------------------------------------------------------

This class defines a object Call, which only contains one parameter callid.

----------------------------------------------------------------------------------------------------------------------------------*/
class Call{
  int callid;
  public Call(int callid){
     this.callid=callid;}
  }

/*----------------------------------------------------------------------------------------------------------------------------------

This class is used to do the string processing. It analyze the string read from file call.dat and extract the corresponding variable
such as Start_time, End_time, callid, cell number and call's property(released or new call).

----------------------------------------------------------------------------------------------------------------------------------*/

class MyString{

    //This method splits a string into an array of strings, and return it.
    public static String[] split(String string, String delimiar){
        int indexf=0;
        int indexb=0;
        string=string.trim();
        int count= split_num(string , delimiar);
        String[] out=new String[count];
```

```
        int len=delimiar.length();
        for(int i=0;i<count;i++){
            indexb=string.indexOf(delimiar,indexf);
            if(indexb==-1)
                indexb=string.length();
            out[i]=string.substring(indexf, indexb);
            indexf=indexb+len;
        }
        return out;
    }

    //This method is used to calculate the number of fields and return it.
    public static int split_num(String string, String delimiar){
        int indexf, indexb;
        int count=1;
        indexf=indexb=0;
        int len=delimiar.length();
        while((indexb=string.indexOf(delimiar,indexf))!=-1){
            count++;
            indexf=indexb+len;
        }
        return count++;
    }
}
```

2. First Available DCA Scheme

This program is written to calculate the blocking probability of First Available DCA Scheme. It includes class MSC, class BaseStation, class Call and class MyString. Among them, class Call and class MyString are the same as the ones in Simple FCA Scheme. Thus, we don't list these two class here. The program was tested using JDK 1.1.5 under Unix system.

```
/*************************************************************************/

package ca.mcgill.pure_dyn;
import java.io.*;

/* ----------------------------------------------------------------------------------------------------------------

This is the controlling class used to calculate the blocking probability using First Available DCA algorithm. In this class, we only
collect the data between Start_time and End_time which given by the user.

 ----------------------------------------------------------------------------------------------------------------*/

class MSC{
    static int current_time;
    static BaseStation[][] basestation;
    static int Start_time, End_time;  //set clock for the data collecting
    static int total_call=0, block_num=0;

   //This is the constructor of the MSC class, doing some initialization
    public MSC(int total_chan_num){
        basestation=new BaseStation[12][12];
        for (int i=0;i<12;i++)
          for (intj=0;j<12;j++)
            basestation[i][j]=new BaseStation(i, j, total_chan_num);
```

```
        for (int i=0;i<12;i++)
          for (int j=0;j<12;j++)
            basestation[i][j].Initial(); //initialize the Base Stations
        current_time=0;
      }

    //This is the main class of the whole package of First Available DCA algorithm
    public static void main(String argv[]){
        int Total_chan_num;
        String call_data="call.dat";
        if(argv.length<3) {
            System.out.println("Usage: total_chan_num start_time end_time");
            System.exit(1);
        }
        //read channel number, start time and end time from keyboard
        Total_chan_num=Integer.parseInt(argv[0]);
        Start_time=Integer.parseInt(argv[1])*60*100;
        End_time=Integer.parseInt(argv[2])*60*100;
        MSC msc=new MSC(Total_chan_num);
        try{
          FileReader reader=new FileReader(call_data);
          BufferedReader buf_reader= new BufferedReader(reader);
          String ln=null;
          while ((ln=buf_reader.readLine())!=null){
             String one_call[]=MyString.split(ln," ");
             current_time=Integer.parseInt(one_call[0]);
             if (one_call[4].charAt(0)=='c') {
                    if(current_time>End_time)   break;
                    if(current_time>Start_time)  total_call++;
                    Call call=new Call(Integer.parseInt(one_call[1]));
                    //according to the data to decide make a new call or release a call
                    basestation[Integer.parseInt(one_call[2])][Integer.parseInt(one_call[3])].MakeNewCall(call);
              }
             else {
                    int callid=Integer.parseInt(one_call[1]);
                    basestation[Integer.parseInt(one_call[2])][Integer.parseInt(one_call[3])].ReleaseCall(callid);
              }
            } //end while
          buf_reader.close();
         }
        catch(IOException e){
          System.err.println(e);
          System.exit(1);
        }
        System.out.println("The block probablity is "+(double)block_num/(double)total_call*100.0);
    }
}

/*-----------------------------------------------------------------------------------------------------------------------------------------

This class is used to calculate the blocking probability of First Available DCA Scheme. It includes Initial(), MakeNewCall(Call),
GetChannel(), FindNewChannel(), TakeNewCall(Call,int) and ReleaseCall(int) methods.

-----------------------------------------------------------------------------------------------------------------------------------------*/

class BaseStation{
    int neighber[][]=new int[6][2];
    BaseStation basestation[]=new BaseStation[6];
    int matrix_x, matrix_y;
    int channal_num;
    Call[] channal_status;

    //constructor of class BaseStation, make some initialization
    public BaseStation(int x, int y, int channal_num){
        matrix_x=x;
        matrix_y=y;
        this.channal_num=channal_num;
```

```
        channal_status=new Call[channal_num];
    }

    //This method decides the neighbors of cell i
    void Initial(){
        if((matrix_x%2)==1){
            basestation[0]= MSC.basestation[matrix_x][(matrix_y+12-1)%12];
            basestation[1]= MSC.basestation[matrix_x][(matrix_y+12+1)%12];
            basestation[2]= MSC.basestation[(matrix_x+12-1)%12][(matrix_y+12-1)%12];
            basestation[3]= MSC.basestation[(matrix_x+12-1)%12][matrix_y];
            basestation[4]= MSC.basestation[(matrix_x+12+1)%12][(matrix_y+12-1)%12];
            basestation[5]= MSC.basestation[(matrix_x+12+1)%12][matrix_y];
        }
        else{
            basestation[0]= MSC.basestation[matrix_x][(matrix_y+12-1)%12];
            basestation[1]= MSC.basestation[matrix_x][(matrix_y+12+1)%12];
            basestation[2]= MSC.basestation[(matrix_x+12-1)%12][(matrix_y+12+1)%12];
            basestation[3]= MSC.basestation[(matrix_x+12-1)%12][matrix_y];
            basestation[4]= MSC.basestation[(matrix_x+12+1)%12][(matrix_y+12+1)%12];
            basestation[5]= MSC.basestation[(matrix_x+12+1)%12][matrix_y];
        }
    }

    //This method is used to find a channel for the call by First Available DCA Algorithm
    int FindNewChannal(){
        Call neighber_channal[][]=new Call[6][channal_num];
        for(int i=0;i<6;i++)
            neighber_channal[i]=basestation[i].GetChannal();
        for(int i=0;i<channal_num;i++){
            if (channal_status[i] == null){
                int j;
                for(j=0;j<6;j++)
                    if(neighber_channal[j][i] != null) break;
                if (j==6) return i;      //if a channel is found, return the channel number
            }
        }
        return -1;     //otherwise, return -1
    }

    //This method is used to serve a new call and calculate the blocking number of calls
    void MakeNewCall(Call call){
        int newchannal=FindNewChannal();
        if(newchannal==-1){
            if(MSC.current_time>MSC.Start_time) MSC.block_num++;
            return;
        }
        TakeNewCall(call, newchannal);
    }

    //This method is used to update the ACO table after  taking a call.
    void TakeNewCall(Call call, int newchannal){
        channal_status[newchannal]=call;
    }

    //This method is used to do the procedure for releasing a call
    void ReleaseCall(int callid){
        for(int i=0;i<channal_num;i++){
            if(channal_status[i]!=null && channal_status[i].callid==callid)
                channal_status[i]=null;
        }
    }

    //This method is a object to return the channel status
    Call[] GetChannal(){
        return channal_status; }
}
}
```

3. Local Packing Scheme

This program is written to calculate the blocking probability of Local Packing Scheme. It includes class MSC, class BaseStation, class Call and class MyString. Among them, class Call and class MyString are the same as the ones in Simple FCA Scheme. Thus, we don't list these two classes here. The program was tested using JDK 1.1.5 under Unix system.

```
/*************************************************************************/

package ca.mcgill.dyn_with_reorg;
import java.io.*;

/* ---------------------------------------------------------------------------------------------------------------------

This is the controlling class used to calculate the blocking probability using LP algorithm. In this class, we only collect the data
between Start_time and End_time which given by the user.

 ---------------------------------------------------------------------------------------------------------------------*/

class MSC{
     static int current_time;
     static BaseStation[][] basestation;
      static int block_num=0;  //counter for blocking calls
      static int total_call=0;     //counter for total calls
      static int Start_time,End_time;  //set the start time and end time for data collecting

     //This is the constructor of class MSC, make some initialization
      public MSC(int total_chan_num){
            basestation=new BaseStation[12][12];
            for (int i=0;i<12;i++)
                 for (int j=0;j<12;j++)
                      basestation[i][j]=new BaseStation(i, j, total_chan_num);
            for (int i=0;i<12;i++) for (int j=0;j<12;j++)
                 basestation[i][j].Initial(); current_time=0;
       }

     //main class of the whole package of LP scheme
     public static void main(String argv[]){
           int Total_chan_num;
           String call_data="call.dat";
           if(argv.length<3) {
              System.out.println("Usage: java ca.mcgill.dyn_with_reorg.MSC total_chan_num start_time end_time");
              System.exit(1);
             }
           //read channel number , start time and end time from the keyboard
           Total_chan_num=Integer.parseInt(argv[0]);
           Start_time=Integer.parseInt(argv[1])*60*100;
           End_time=Integer.parseInt(argv[2])*60*100;

           MSC msc=new MSC(Total_chan_num);
           try{
             FileReader reader=new FileReader(call_data);
             BufferedReader buf_reader= new BufferedReader(reader);
             String ln=null;
             while ((ln=buf_reader.readLine())!=null){
                  String one_call[]=MyString.split(ln," ");
                  current_time=Integer.parseInt(one_call[0]);
```

```
                //read data from file call.dat to decide to make a new call or release a call
                if (one_call[4].charAt(0)=='c') {
                    if(current_time>End_time) break;
                    if(current_time>Start_time) total_call++;
                    Call call=new Call(Integer.parseInt(one_call[1]));
                    basestation[Integer.parseInt(one_call[2])]
                    [Integer.parseInt(one_call[3])].
                    MakeNewCall(call);
                 }
                else {
                    int callid=Integer.parseInt(one_call[1]);
                    basestation[Integer.parseInt(one_call[2])]
                    [Integer.parseInt(one_call[3])].
                    ReleaseCall(callid);
                 }
            } //end while
            buf_reader.close();
            }
        catch(IOException e){
            System.err.println(e);
            System.exit(1);
        }
        System.out.println("The block probablity is "+(double)block_num/(double)total_call*100.0);
    }
}

/*-----------------------------------------------------------------------------------------------------------------------------------------

This class is used to calculate the blocking probability of Local Packing Scheme. It includes Initial(), MakeNewCall(Call),
GetChannel(), FindNewChannel(), IsReorganizable(), Switch_channel(int,int), TakeNewCall(Call,int) and ReleaseCall(int)
methods.

-----------------------------------------------------------------------------------------------------------------------------------------*/

class BaseStation{
    BaseStation basestation[]=new BaseStation[6];
    Call neighber_channal[][]=new Call[6][];
    int matrix_x, matrix_y; int channal_num;
    Call[] channal_status;

    //This is the constructor of class BaseStation, making some initialization
    public BaseStation(int x, int y, int channal_num){
        matrix_x=x;
        matrix_y=y;
        this.channal_num=channal_num;
        channal_status=new Call[channal_num];
     }

    //This method is used to initialize the neighbor of cell i
    void Initial(){
        if((matrix_x%2)==1){
            basestation[0]= MSC.basestation[matrix_x][(matrix_y+12-1)%12];
            basestation[1]= MSC.basestation[matrix_x][(matrix_y+12+1)%12];
            basestation[2]= MSC.basestation[(matrix_x+12-1)%12][(matrix_y+12-1)%12];
            basestation[3]= MSC.basestation[(matrix_x+12-1)%12][matrix_y];
            basestation[4]= MSC.basestation[(matrix_x+12+1)%12][(matrix_y+12-1)%12];
            basestation[5]= MSC.basestation[(matrix_x+12+1)%12][matrix_y];
         }
        else{
            basestation[0]= MSC.basestation[matrix_x][(matrix_y+12-1)%12];
            basestation[1]= MSC.basestation[matrix_x][(matrix_y+12+1)%12];
            basestation[2]= MSC.basestation[(matrix_x+12-1)%12][(matrix_y+12+1)%12];
            basestation[3]= MSC.basestation[(matrix_x+12-1)%12][matrix_y];
            basestation[4]= MSC.basestation[(matrix_x+12+1)%12][(matrix_y+12+1)%12];
            basestation[5]= MSC.basestation[(matrix_x+12+1)%12][matrix_y];
         }
     }
```

```
//This method is used to decide if an empty channel can be found
int FindNewChannal(){
    for(int i=0;i<6;i++)
        neighber_channal[i]=basestation[i].GetChannal();
    for(int i=0;i<channal_num;i++){
        if (channal_status[i] == null){
            int j;
            for(j=0;j<6;j++)
                if(neighber_channal[j][i] != null) break;
            if (j==6) return i;  //if find an empty channel, return the channel number
        }
    }
    return -1;   //otherwise, return -1
}

//This method is used to decide if a channel can be found by reorganization using LP algorithm
int IsReorganizable(){
    for (int i=0;i<channal_num;i++){
        if(channal_status[i]==null){
            int j;
            int candadate_basestation=0;
            int k=0;
            //find the column which only has one check mark in the ACO table
            for(j=0;j<6;j++){
                if(neighber_channal[j][i]!=null) {
                    candadate_basestation=j;
                    k++;
                }
                if(k>1) break;
            }
            if(j==6 && k<2){
                int empty_channal=basestation[candadate_basestation].FindNewChannal();
                if(empty_channal!=-1){    //if the responding cell has empty channel, switch the channel
                    basestation[candadate_basestation].Switch_Channal(i,empty_channal);
                    return i;    //return the channel number i
                }
            }
        }
    }
    return -1; //if cannot reorganizable, return -1
}

//This method is used to do the procedure of call assignment and calculate the blocking number of calls
void MakeNewCall(Call call){
    int newchannal=FindNewChannal();
    if(newchannal==-1){
        int reorganz_channal=IsReorganizable();
        if(reorganz_channal!=-1){
            TakeNewCall(call, reorganz_channal);
            return;
        }
        if(MSC.current_time>MSC.Start_time)
            MSC.block_num++;
        return;
    }
    TakeNewCall(call, newchannal);
}

//This method is used to update the ACO table after take a new call
void TakeNewCall(Call call, int newchannal){
    channal_status[newchannal]=call;
}

//This method is used to switch a channel if the channel can be reorganized
void Switch_Channal(int from, int to){
    channal_status[to]=channal_status[from];
```

```
        channal_status[from]=null;
    }

    //This method is used to do the procedure of releasing a call
    void ReleaseCall(int callid){
          for(int i=0;i<channal_num;i++){
             if(channal_status[i]!=null && channal_status[i].callid==callid)
                channal_status[i]=null;
          }
       }

    //This method is used to return the status of channel
    Call[] GetChannal(){
      return channal_status;
    }
}
```

4. Our Proposed Scheme

This program is written to calculate the blocking probability of our proposed scheme. It includes class MSC, class BaseStation, class WaitingCall, class WaitingQueue, class Call and class MyString. Among them, class Call and class MyString are the same as the ones in Simple FCA Scheme. Thus, we don't list these two classes here. The program was tested using JDK 1.1.5 under Unix system.

```
/*************************************************************************/

package ca.mcgill.dyn_with_reorgandqueue;
import java.io.*;

/* ---------------------------------------------------------------------------------------------------------------------

This is the controlling class used to calculate the blocking probability using our proposed algorithm. In this class, we only collect
the data between Start_time and End_time which given by the user. And waiting time of a call in the waiting queue is also given
by user.

 ---------------------------------------------------------------------------------------------------------------------*/

class MSC{
     static int current_time;
     static BaseStation[][] basestation;
     static WaitingQueue waitingqueue; //the object of waiting queue
     static int block_num=0; //counter for blocking calls
     static int total_call=0;   //counter for total calls
     static int Start_time,End_time;   //set the start time and end time for data collecting
     static int ignore_num=0; //counter of calls whose duration time is less than waiting time
     static int waiting_time;  //waiting time for a call in the waiting queue

     //This is the constructor of class MSC, doing some initialization
     public MSC(int total_chan_num){
          basestation=new BaseStation[12][12];
          waitingqueue=new WaitingQueue();
               for (int i=0;i<12;i++)
                  for (int j=0;j<12;j++)
                       basestation[i][j]=new BaseStation(i, j, total_chan_num);
               for (int i=0;i<12;i++) for (int j=0;j<12;j++)
                  basestation[i][j].Initial(); current_time=0;
       }
```

```
    //This is the main class of the whole package of our proposed algorithm
    public static void main(String argv[]){
        int Total_chan_num;
        String call_data="call.dat";
        if(argv.length<4) {
        System.out.println("Usage: total_chan_num start_time end_time waiting_time");
        System.exit(1);
     }

     //read channel number, start time, end time and waiting time from the keyboard
     Total_chan_num=Integer.parseInt(argv[0]);
     Start_time=Integer.parseInt(argv[1])*60*100;
     End_time=Integer.parseInt(argv[2])*60*100;
     waiting_time=Integer.parseInt(argv[3])*100;
     MSC msc=new MSC(Total_chan_num);
     try{
        FileReader reader=new FileReader(call_data);
        BufferedReader buf_reader= new BufferedReader(reader);
        String ln=null;
        //read data from file call.dat, only processing data between start time and end ime
        while ((ln=buf_reader.readLine())!=null){
                String one_call[]=MyString.split(ln," ");
                current_time=Integer.parseInt(one_call[0]);
                //decide if it is a new call or a released call
                if (one_call[4].charAt(0)=='c') {
                        if(current_time>End_time) break;
                        if(current_time>Start_time) total_call++;
                        Call call=new Call(Integer.parseInt(one_call[1]));
                        basestation[Integer.parseInt(one_call[2])][Integer.parseInt(one_call[3])].MakeNewCall(call);
                }
                else {
                        int callid=Integer.parseInt(one_call[1]);
                        basestation[Integer.parseInt(one_call[2])][Integer.parseInt(one_call[3])].ReleaseCall(callid);
                 }
        } //end while
        buf_reader.close();
      } //end try
      catch(IOException e){
        System.err.println(e);
        System.exit(1);
      }
      System.out.println("The block probablity is "+(double)block_num/(double)total_call*100.0);
    }
}

/*-----------------------------------------------------------------------------------------------------------------------------------------

This class is used to calculate the blocking probability of our proposed scheme. It includes Initial(), MakeNewCall(Call),
GetChannel(), FindNewChannel(), TakeNewCall(Call,int) and ReleaseCall(int) methods.

-----------------------------------------------------------------------------------------------------------------------------------------*/

class BaseStation{
    BaseStation basestation[]=new BaseStation[6];
    Call neighber_channal[][]=new Call[6][];
    WaitingQueue waitingqueue;
    int matrix_x, matrix_y;
    int channal_num;
    Call[] channal_status;

    //This is the constructor of class BaseStation, making some initialization
    public BaseStation(int x, int y, int channal_num){
        matrix_x=x;
        matrix_y=y;
        this.channal_num=channal_num;
        channal_status=new Call[channal_num];
```

```
}

//This method is used to initial the neighbor of cell i
void Initial(){
    waitingqueue=MSC.waitingqueue;
    if((matrix_x%2)==1){
        basestation[0]= MSC.basestation[matrix_x][(matrix_y+12-1)%12];
        basestation[1]= MSC.basestation[matrix_x][(matrix_y+12+1)%12];
        basestation[2]= MSC.basestation[(matrix_x+12-1)%12][(matrix_y+12-1)%12];
        basestation[3]= MSC.basestation[(matrix_x+12-1)%12][matrix_y];
        basestation[4]= MSC.basestation[(matrix_x+12+1)%12][(matrix_y+12-1)%12];
        basestation[5]= MSC.basestation[(matrix_x+12+1)%12][matrix_y];
    }
    else{
        basestation[0]= MSC.basestation[matrix_x][(matrix_y+12-1)%12];
        basestation[1]= MSC.basestation[matrix_x][(matrix_y+12+1)%12];
        basestation[2]= MSC.basestation[(matrix_x+12-1)%12][(matrix_y+12+1)%12];
        basestation[3]= MSC.basestation[(matrix_x+12-1)%12][matrix_y];
        basestation[4]= MSC.basestation[(matrix_x+12+1)%12][(matrix_y+12+1)%12];
        basestation[5]= MSC.basestation[(matrix_x+12+1)%12][matrix_y];
    }
}

//This method is used to decide if an empty channel can be found
int FindNewChannal(){
  for(int i=0;i<6;i++)
    neighber_channal[i]=basestation[i].GetChannal();
  for(int i=0;i<channal_num;i++){
    if (channal_status[i] == null){
      int j;
      for(j=0;j<6;j++)
        if(neighber_channal[j][i] != null) break;
      if (j==6) return i;  //if find an empty channel, return the channel number
    }
  }
  return -1;   //otherwise, return -1
}

//This method is used to decide if a channel can be found by reorganization in ACO table
int IsReorganizable(){
  for (int i=0;i<channal_num;i++){
    if(channal_status[i]==null){
      int j;
      int candadate_basestation=0;
      int k=0;
      for(j=0;j<6;j++){
        if(neighber_channal[j][i]!=null) {
          candadate_basestation=j;
          k++;
        }
        if(k>1) break;
      }
      if(j==6 && k<2){
        int empty_channal=basestation[candadate_basestation].FindNewChannal();
        if(empty_channal!=-1){
          basestation[candadate_basestation].Switch_Channal(i,empty_channal);
          return i;      //if find channel, then return the channel number
        }
      }
    }
  } //end for
  return -1;   //otherwise, return -1
}

//This method is used to do the procedure of call assignment using our proposed algorithm
void MakeNewCall(Call call){
    int newchannal=FindNewChannal();
```

```
            if(newchannal==-1){
                int reorganz_channal=IsReorganizable();
                if(reorganz_channal!=-1){
                    TakeNewCall(call, reorganz_channal);
                    return;
                  }
                //if no channel available, then put it into waiting queue
                waitingqueue.Enqueue(new WaitingCall(matrix_x,matrix_y,MSC.current_time, call.callid));
                return;
             }
            TakeNewCall(call, newchannal);      //if there is an empty available, take the new call
         }

        //This method is used to update the ACO table after taking a new call
        void TakeNewCall(Call call, int newchannal){
            channal_status[newchannal]=call;
         }

        //This method is used to switch the channel if  reorganization occurs
        void Switch_Channal(int from, int to){
           channal_status[to]=channal_status[from];
           channal_status[from]=null;
        }

        //This method is used to do the call releasing procedure
        void ReleaseCall(int callid){
            if(waitingqueue.CheckReleaseCall(callid))
               return;
            int i;
            for(i=0;i<channal_num;i++){
               if(channal_status[i]!=null && channal_status[i].callid==callid){
                   channal_status[i]=null;
                   break;
                }
             }
            if(i<channal_num)
              waitingqueue.CheckWaitingCall(matrix_x,matrix_y,i);
         }

        //This is the object which contains the channel status
        Call[] GetChannal(){
            return channal_status;
         }
}

/*-----------------------------------------------------------------------------------------------------------------------------------------

This class is used to keep the identification of a waiting  call. It includes the cell number which the waiting call comes from, the
callid and the time which this call have already waited

-----------------------------------------------------------------------------------------------------------------------------------------*/

class WaitingCall{
      int source_x, source_y;
      int source_time;
      int source_callid;

      //This is the constructor of class WaitingCall, which doing some initialization
      WaitingCall(int x,int y,int time, int callid){
         source_x=x;
         source_y=y;
         source_time=time;
         source_callid=callid;
      }
 }
```

```
/*-----------------------------------------------------------------------------------------------------------------------

This class is used to describe the object Waiting Queue. It includes Enqueue(WaitingCall), CheckWaitingCall(int, int, int),
CheckRealseCall(int) and IsNeighber(int, int, int, int) methods.

-----------------------------------------------------------------------------------------------------------------------*/

import java.util.*;

class WaitingQueue{
      Vector waitingqueue;
      int waitingtime=MSC.waiting_time;

      //This is the constructor of class WaitingQueue, making some initialization
      public WaitingQueue(){
      waitingqueue=new Vector();
      }

     //This method is used to add the call without channel in the waiting queue
      void Enqueue(WaitingCall waitingcall){
           waitingqueue.addElement(waitingcall);
      }

     //This method is used to check if the channel released can be assigned to the call in the waiting queue.
      void CheckWaitingCall(int matrix_x, int matrix_y, int freechannal){
            if(waitingqueue.isEmpty()) return;    //if the queue is empty, then doonot need check
            Enumeration waitingitem = waitingqueue.elements();
           //use channel release procedure of our proposed algorithm to assign the channel
            while(waitingitem.hasMoreElements()){
                WaitingCall item= (WaitingCall) waitingitem.nextElement();
                int x=item.source_x;
                int y=item.source_y;
                if(matrix_x==x && matrix_y==y){   //if the waiting call also comes from cell I, assign the channel to it directly
                      MSC.basestation[x][y].TakeNewCall(new Call(item.source_callid), freechannal);
                      waitingqueue.removeElement(item);
                      return;
                 }
                //if waiting call is from the interfering cell of cell I, check if it can be reorganized
                if(IsNeighber(matrix_x, matrix_y, x, y)){
                      int channal=MSC.basestation[x][y].FindNewChannal();
                      if(channal!=-1){
                            MSC.basestation[x][y].TakeNewCall(new Call(item.source_callid),channal);
                            waitingqueue.removeElement(item);
                            return;
                        }
                      else {
                            int reorganz_channal=MSC.basestation[x][y].IsReorganizable();
                            if(reorganz_channal!=-1){
                                MSC.basestation[x][y].TakeNewCall(new Call(item.source_callid), reorganz_channal);
                                waitingqueue.removeElement(item);
                                return;
                             }
                       } //end else
                  } //end if
              } //end while
            return;
      }

     //This method is used to check the waiting queue when a call is released
      boolean CheckReleaseCall(int callid){
            boolean return_value=false;
            if(waitingqueue.isEmpty()) return false;
            Enumeration waitingitem = waitingqueue.elements();
            while(waitingitem.hasMoreElements()){
                WaitingCall item= (WaitingCall) waitingitem.nextElement();
               //if the waiting time is expired, then this call is blocked
                if((MSC.current_time-item.source_time)>waitingtime){
```

```
                waitingqueue.removeElement(item);
                if(MSC.current_time>MSC.Start_time)
                    MSC.block_num++;
                waitingitem=waitingqueue.elements();
                continue;
            }
          if(callid==item.source_callid){
                waitingqueue.removeElement(item);
                return_value=true;
                //if the call's duration time is less than waiting time, this call is ignored
                if(MSC.current_time>MSC.Start_time){
                    MSC.ignore_num++;
                    MSC.total_call--;
                }
          }
        } //end while
        return return_value;
    }

  //This method is used to decide if the waiting call is from the interfering cell of cell i
  boolean IsNeighber(int matrix_x, int matrix_y, int x, int y){
      if(matrix_x%2==1)
        if((matrix_x==x&&(matrix_y+12-1)%12==y)||(matrix_x==x&&(matrix_y+12+1)%12==y)||
            ((matrix_x+12-1)%12==x&&(matrix_y+12-1)%12==y)|| ((matrix_x+12-1)%12==x&&(matrix_y==y))||
            ((matrix_x+12+1)%12==x&&(matrix_y+12-1)%12==y)|| ((matrix_x+12+1)%12==x&&matrix_y==y))
                return true;
      else if((matrix_x==x&&(matrix_y+12-1)%12==y)||(matrix_x==x&&(matrix_y+12+1)%12==y)||
            ((matrix_x+12-1)%12==x&&(matrix_y+12+1)%12==y)|| ((matrix_x+12-1)%12==x&&(matrix_y==y))||
            ((matrix_x+12+1)%12==x&&(matrix_y+12+1)%12==y)|| ((matrix_x+12+1)%12==x&&matrix_y==y))
                return true;
      return false;
  }
}
```

Bibliography

[Anselm95] Anselm Lingnau and Oswald Drobnik, "An infrastructure for mobile agents: Requirements and Architecture", Proc. 13th DIS Workshop, Orlando, Florida, September 1995

[Chess95] D. Chess et al, "Itinerant Agents for Mobile Computing", IBM Research Report RC20010, IBM Research Division, 1995

[Chih93] Chih-Lin and Pi-Hui Chao, "Local Packing Distributed Dynamic Channel Allocation at Cellular Base Station", IEEE GLOBECOM, 1:293-301, 1993

[Coch92] Don Cochrane, "Quality of Service Mappings", The Management and Telecommunications Networks, eds: R. Smith, E. H. Mamdani, and J. G. Callagan, Ellis Horwood, 1992

[Cox72] D. C. Cox and D. O. Reudink, "Dynamic Channel Assignment in two dimension large-scale mobile radio systems", The Bell System Technical Journal, 51:1611-1628, 1972

[Dini97] Petre Dini, A. Hafid, "Towards Automatic Trading of QoS Parameters in Multimedia Distributed Applications", Open Distributed Processing and Distributed Platforms, 4: 166-179, May, 1997

[Donald72] Donald C. Cox and D. O. Reudink, "A Comparison of Some Channel Assignment Strategies in Large-Scale Mobile Communication System", IEEE Transaction on Communications, COM-20(2): 190-195, April, 1972

[Donald73] Donald C. Cox and Douglas O. Reudink, "Increasing Channel Occupancy in Large-Scale Mobile Radio Systems: Dynamic Channel Reassignment", IEEE Transactions on Vehicular Technology, VT-22(4): 218-222, November, 1973

[Elnoubi82] S. M. Elnoubi, R. Singh and S. C. Gupta, "A new frequency channel assignment algorithm in high capacity mobile communication systems", IEEE Trans. Vech. Technol, VT-31(3), August, 1982

[Engel73] J. S. Engel and M. Peritsky, "Statistically optimum dynamic sever assignment in systems with interfering severs", IEEE Trans. Vech. Technology, VT-22(4), November, 1973

[Fred96] Fred Halsall, "Data Communications, Computer Networks and Open Systems", Addison-Wekley Publishing Company, ISBN 0-201-42293-X, 1996

[Furuya91] Furuya. Y. and Yoshihiko Akaiwa, "Channel Segregation : A Distributed Channel Allocation Scheme for Mobile Communication Systems", IEICE Transactions, 74:1531-1537, 1991

[GMD97] GMD FOKUS, "Mobile Agent System Interoperability Facilities Specification", OMG TC Document orbos/97-10-05, November, 1997

[Hosoon96] Hosoon Ku and Gottfried W. R. Luderer, "An intelligent mobile agent framework for distributed network management", Network System Laboratory, Arizona State University, 1996

[Hua93] Hua Jiang and Stephen. S. Rappaport, "Prioritized Channel Borrowing Without Locking: A Channel Sharing Strategy for Cellular Communication", IEEE GLOBECOM, 1:276-280, 1993

[Jens93] Jens Zander and Hakan Eriksson, "Asymptotic Bounds on the Performance of a Class of Dynamic Channel Assignment Algorithms", IEEE Journal on Selected Areas in Communications, 11(6):926- 933, August, 1993

[Kahwa78] T. J. Kahwa and N. D. Georganas, "A hybrid channel assignment scheme in large scale", IEEE Trans. Communication, COM-26(4), April, 1978

[Katzela96] I. Katzela and M. Naghshineh, "Channel Assignment Schemes for Cellular Mobile Telecommunication Systems: A Comprehensive Survey", IEEE Personal Communications, pages 10-31, June, 1996

[Kazunori92] Kazunori Okada and Fumito Kubota, "On Dynamic Channel Assignment Strategies in Cellular Mobile Radio Systems", IEICE Transactions Fundamentals, 75(1634-1641), 1992

[Lewis73] Lewis. G. Anderson, "A Simulation Study of Some Dynamic Channel Assignment Algorithms in a High Capacity Mobile Telecommunications System", IEEE Transactions on Vehicular Technology, VT-22(4):210-217, November, 1973

[Michael95] Michael R. Genesereth and Steven. Ketchpel, "Software Agents", Computer Science Department, Stanford University, 1995

[Mike95] Mike Rizzo and Ian A. Utting, "An Agent-based Model for the provision of Advanced Telecommunications Services", Computing laboratory, University of Kent at Canterbury, U. K., 1995

[Ming89] Ming Zhang and Tak-Shing P. Yum, "Comparisons of Channel-Assignment Strategies in Cellular Mobile Telephone Systems", IEEE Transactions on Vehicular Technology, 38(4):211-215, November, 1989

[Ming91] Ming Zhang and Tak-Shing P. Yum, "The Nonuniform Compact Pattern Allocation Algorithm for Cellular Mobile System", IEEE Transactions on Vehicular Technology, 40(2):387-391, May, 1991

[Moo96] Moo Wan Kim, et al, "Dual Agent System to Integrate Service Control and Network Management", Fujitsu Laboratories Ltd., Japan, 1996

[Mutsumu93] Mutsumu Sericawa and David J. Goodman, "Instability and Deadlock of Distributed Dynamic Channel Allocation", Proceedings of the 43rd IEEE Vech. Technology Conference, pages 528-531, 1993

[Nettleton89] R. W. Nettleton, "A high capacity assignment method for cellular mobile telephone systems", 39th IEEE VTC, pages 359-367, 1989

[Partha97] Partha P. Bhattacharya and Leonidas Georgiadis, "Distributed Channel Allocation for PCN with Variable Rate Traffic", IEEE/ACM Transaction on Networking, 5(6):907-923, December, 1997

[Race82] Race Project QOSMIC Deliverable 1.3C: "QoS and Performance Relation", 82/KT/LM/DS/B/013/b1

[Reece96] C. S. Reece and A. Van De Liefvoort, "Performance Analysis of Heterogeneous Traffic on an Integrated Network Link with Finite Waiting Room and Anticipated-Release Researvation Policy", Computer Science Telecommunications, University of Missouri-Kansas City, February, 1996

[Songwu97] Songwu Lu and Vaduvur Bharghavan, "Adaptive Resource Management Algorithms for Indoor Mobile Computing Environments", Coordinated Sciences Laboratory, University of Illinois at Urbana-Champaign, 1997

[Steven97] Steven R. Farley, "Mobile agent system architecture", SIGS Publications, Inc. New York, NY, USA, 1997

[Sunghyun96] Sunghyun Choi and Kang G. Shin, "A Cellular Wireless Local Area Network with QoS Guarantees for Heterogeneous Traffic", Real-time Computing Laboratory, The University of Michigan, CSE-TR-300-96, August, 1996

[URL1] http://www.dse.doc.ic.ac.uk/~nd, 1998

[URL2] http://www.cs.tcd.ie/Brenda. Nangle/iag.html, 1997

[URL3] http://www.acl.lanl.gov/CORBA, 1998

[URL4] http://www.ikv.de/products/grasshopper.html, 1998

[URL5] http://ccnga.uwaterloo.ca/jscourias/GSM/gsmreport.html, 1998

[URL6] http://www.readiodesign.com/cellwrks.html, 1998

[URL7] http://www.seanet.com/~ksbrown/kmath026.htm, 1997

[URL8] http://forum.swarthmore.edu/dr.math, 1998

[URL9] http://www.bell-labs.com, 1997

[URL10] http://www.concentric.net/~tkvallil/snmp2.html, 1997

[URL11] http://iworks.interworks.org/conference/IWorks97/sessions, 1998

[URL12] http://www.albury.net.au/~rhodes/knowldge.html, 1997

[URL13] http://www.zdwebopedia.com/TERM/U/UML.html, 1998

[URL14] http://www.rational.com/uml/index.shtml, 1998

[URL15] http://homer.span.ch/~spaw2724/SNMP, 1998

[Vijay93] Vijay K. Jain and Bonchul Koo, "TDMA/FDMA PCN System: An Advanced Channel Borrowing Strategy", IEEE GLOBECOM, 1:271-274, 1993

[White95] White J., "Telescript technology: The foundation of the electronic market place", General Magic white paper, 1995

[Wuyi91] Wuyi Yue, "Analytical Methods to Calculate the Performance of a Cellular Mobile Radio Communication System with Hybrid Channel Assignment", IEEE Transactions on Vehicular Technology, 40(2): 453-460, May, 1991

[Yoshiyasu93] Yoshiyasu Nishibe, et al, "Distributed Channel Allocation in ATM Network", IEEE GLOBECOM, 1:417-423, 1993