A SIMULATION FRAMEWORK FOR PERFORMANCE EVALUATION AND SECURITY RESEARCH IN MULTI-INTERFACE MULTI-CHANNEL NETWORKS

THESIS

Presented to the Graduate Council
of Texas State University-San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

Heywoong Kim, B.S

San Marcos, Texas
December 2010

FAIR USE AND AUTHORS PERMISSION STATEMENT

Fair Use

Duplication Permission

ACKNOWLEDGMENTS

First of all, I would like to thank Dr. Qijun Gu, the supervisor professor of my thesis, for his long supervision and contribution. Without his guidance and thoughtful support, my research work could not have been completed. I owe a huge debt of gratitude to his kindness and patience. I would also like to thank Dr. Xiao Chen and Dr. Mina S. Guirguis for agreeing to be on my committee and for their insight and help.

In addition, my deepest gratitude goes to my family for their unflagging love and support throughout my life, this thesis is simply impossible without them. Also, I thank to friends of Chi Alpha, Christian Fellowship at Texas State University-San Marcos, whose presence helps to make my life abundant in San Marcos.

This thesis was supported by NSF under award number 0916469, 0916000, and 0915318.

This manuscript was submitted on November 1, 2010.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

A SIMULATION FRAMEWORK FOR PERFORMANCE EVALUATION AND SECURITY RESEARCH IN MULTI-INTERFACE MULTI-CHANNEL NETWORKS

by

Heywoong Kim

Texas State University-San Marcos

December 2010

SUPERVISING PROFESSOR: QIJUN GU

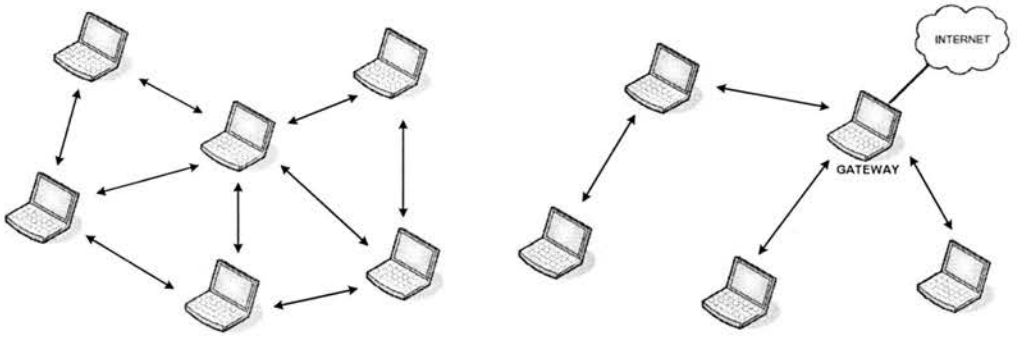
In wireless networks, devices can be equipped with multiple interfaces to utilize multiple channels and increase the overall throughput of a network. Various channel assignment protocols have been developed to better utilize multiple channels and interfaces. However, the research of channel assignment protocols is still lack of a good simulation tool that can content with a variety of requirements and specifications of channel assignment protocols. This thesis proposes MIMC-SIM, a generic simulation framework to study channel assignment protocols in multi-interface and multi-channel networks. The MIMC-SIM framework is built in OMNeT++ with INET and implements a new layer between the network layer and the MAC layer. The MIMC-SIM framework has a novel structure which supports

generic features and specific behaviors of channel assignment protocols It also provides a generic and flexible code structure for implementing channel assignment protocols

CHAPTER 1

INTRODUCTION

Wireless network is a type of network in which nodes communicate over a distance using radio signals instead of wires. Since computers became able to communicate via wireless networks, many efforts have been contributed to increase capacity and accessibility of wireless networks. Many wireless protocols have been developed, such as IEEE 802.11, Bluetooth, etc. With such wireless protocols, various wireless networks have been implemented, for instance, ad hoc network and mesh network. An ad hoc network is a type of wireless network in which nodes act as independent routers and forward packets for communication with other nodes. A mesh network is a type of an ad hoc network. In a mesh network, typically, one of the nodes connects to another network, such as the Internet, and behaves as a gateway. Most traffic in the mesh network is directed to/from a gateway [19]. Figure 1.1(a) and 1.1(b) show



(a) Ad Hoc Network (b) Mesh Network

Figure 1.1: Two Types of Wireless Networks

an ad hoc network and a mesh network respectively. Such wireless networks can be greatly extended by each node without such infrastructure as an access point.

However, the capacity of wireless networks is limited compared to wired networks. In a wireless network, when nodes are close enough to communicate with each other, it is said that they are in the communication range. In the communication range, only one transmission is allowed in a single channel at a moment. When multiple transmissions occur simultaneously in a single channel, the communications interfere with each other. Such interference incurred by adjoining nodes aggravates the capacity of a wireless network. In order to prevent such interference, utilizing multiple channels and multiple network interface cards (NIC) has been considered.

Many communication protocols, such as IEEE802.11, Bluetooth, and WiMAX, provide multiple orthogonal channels whose frequencies do not overlap with each other Utilizing multiple orthogonal channels allows nodes to communicate simultaneously on different channels without interference. Such simultaneous multiple communications can improve the total throughput of a network [20]. In addition, in order to utilize multiple channels efficiently, multiple interfaces are equipped in each node and assigned to different channels. Thereby, such a network is called multi-interfaces multi-channels (MIMC) network, in which nodes utilize multiple channels with multiple interfaces.

Many research have shown that MIMC networks provide much better performance than single channel wireless networks Ashish et al [19] showed that a MIMC network can achieve a factor of 6 to 7 throughput improvement compared to

a single channel wireless network Pradeep et al. [17] showed that MIMC networks have better performance even when the number of interfaces is smaller than the number of channels. Vartika et al [14] also demonstrated that even if frequently switching channels is limited, MIMC networks still achieve good throughput

A MIMC network can achieve such good performance by a carefully designed channel assignment protocol. A channel assignment (CA) protocol assigns the multiple channels to nodes so as to better utilize multiple channels and interfaces and maximize the overall throughput of a network. CA protocols allow nodes to exchange their channel and traffic information, collaborate on channel assignment negotiation, and assign channels to nodes to reduce interference in transmission The design of CA protocols has been studied in mesh network [19, 16] and ad hoc networks [26, 18]

However, no good and generic simulation tools are available for studying problems of channel assignment in MIMC networks. The simulation tools developed by existing research on CA protocols are too specific to the CA protocols and the network topologies [19, 16, 26, 18]. They are hard to be reused for studying various problems in MIMC networks and evaluating and comparing performance of proposed new schemes. Although quite a few emulation testbeds and simulation tools have been developed for studying wireless networks, they are still not sufficient yet to satisfy the needs of MIMC network research Several deployed wireless testbeds [1, 5, 11, 12] can be used to validate some wireless protocols However, nodes in these testbeds mostly have only one radio, even though they use multiple channels The testbeds can only emulate a network with a limited scale. The

topology of the nodes is hard to change, and node mobility can hardly be studied in these testbeds. Meanwhile, a few simulation tools have been developed [2, 3, 6, 9], which can address the problems in the wireless testbeds. They can support a large-scale simulation of various protocols in wireless and mobile networks. However, to the best of our knowledge, no general simulation framework has been actually developed for MIMC networks. Even though some simulation tools have partially added mechanisms for supporting multiple interfaces and multiple channels, they have not truly examined the needs of MIMC network simulation which will be discussed shortly in Section 4.

This thesis presents a generic simulation framework, named MIMC-SIM, for MIMC networks The MIMC-SIM framework is built in INET/OMNeT++. The main purpose of the MIMC-SIM framework is to include generic features of CA protocols and support a variety of CA protocols. To do so, the MIMC-SIM framework adds a new layer between the network layer and the MAC layer where CA protocols are adopted The new layer allows CA protocols to work compatibly with protocols at the network and the MAC layers In addition, the MIMC-SIM framework provides generic and flexible code structure for easy extension according to protocol specification The MIMC-SIM framework also adapts a variety of factors in simulation, such as network topology and traffic volume. In the MIMC-SIM framework, two CA protocols are implemented and experimented according to [19] and [26] Additionally, vulnerability of the two CA protocols is tested by placing an attacking node which manipulates the CA protocols in the network. The MIMC-SIM framework will contribute to the research and development of MIMC

networks.

The rest of this thesis is organized as follows:

Chapter 2 provides the background of CA protocols.

Chapter 3 discusses OMNeT++ and INET framework.

Chapter 4 discusses design issues of a MIMC network simulator and overviews the architecture of the MIMC-SIM framework

Chapter 5 and 6 present major modules in the MIMC-SIM framework in detail

Chapter 7 presents adjunct modules in the MIMC-SIM framework in detail and discusses modification in INET.

Chapter 8 presents a generic code structure to implement a state machine.

Chapter 9 shows the implementation of CA protocols in the MIMC-SIM framework

Chapter 10 shows evaluation of CA protocols in the MIMC-SIM framework.

Chpater 11 shows vulnerability of CA protocols in the MIMC-SIM framework

Finally, Chapter 12 provides the conclusion of this thesis.

CHAPTER 2

CA PROTOCOLS

In a MIMC network, CA protocols conduct nodes to assign channels so as to minimize interference among nodes and maximize the overall throughput of a network. To do so, CA protocols allow nodes to exchange their channel information and traffic information each other For example, CA protocols usually ask nodes to scan and listen local traffic when they just join a network in order to find neighbor nodes and available channels When a node obtains a channel, the node shall broadcast its channel and related information to let other neighbor nodes know CA protocols define how nodes exchange their information with neighbor nodes and assign channels based on the shared information

In a MIMC network, nodes can use multiple channels with multiple interfaces simultaneously. However, considering the cost and the small size of a node, normally the number of interfaces, m, of a node should be smaller than the number of channels, c It is shown [17] that the network capacity is affected by the ratio of c to m, rather than the number c or m When c/m is $O(log(n))$ in a random network, network capacity will not be degraded Because of $m < c$, CA protocols mostly focus on deploying channels to nodes to minimize interference and maximize throughput of a network.

In this chapter, a few CA protocols that aim to improve network capacity by

reducing channel interference are briefly summarized

2.1 CA Protocols

In [18], in order to increase network capacity, interfaces of a node are divided into two categories fixed interface and switchable interface A fixed interface is assigned to a particular channel and works on the channel for long time period. A fixed interface is used to receive packets from other nodes. A node randomly selects a channel in an initial level and assigns the channel to a fixed interface. Later, the node could change a channel of a fixed interface to a less used channel to reduce interference. A switchable interface is used to ensure connectivity with other nodes In other words, nodes frequently switch a channel of its switchable interface to its neighbor nodes' fixed channel for sending packets. The drawback of the protocol is that the channel assignment of a fixed interface takes time to converge. In addition, if the number of channels that nodes can use is large, the switching channel delay may be large when nodes need to switch back and forth to communicate with different neighbor nodes

In [26], CA algorithms based on s-disjunct superimposed code was proposed to mitigate co-channel interference of network capacity maximization For each node, all orthogonal channels are labeled as either 1 for primary or 0 for secondary via a binary channel codeword Then, a node, u, first searches a set of primary channels that are secondary to all interferers in two-hop communication range since these channels may not be used by the interferers. If the searching fails, u chooses the secondary channels that are not primary, but also secondary to any of interferers

since the interferers may not use them either If u cannot find such channel, it picks up the primary channel that is primary to the least number of interferences.

In [16, 19], CA protocols were proposed specifically for a wireless mesh network [19] considers the channel assignment problem as two sub problems: 1) an interface assignment problem where interfaces of a node are divided into two categories· UP-NICs used for communicating with its parent node, and DOWN-NICs used for communicating with its child nodes; 2) an interface-channel assignment problem where the channel assignment of a node's UP-NIC is determined by its parents A less loaded channel will be assigned to a DOWN-NIC to prevent the interference A node periodically reevaluates its current channel usage and switches a heavily loaded channel to a less loaded channel for its DOWN-NIC. The channel assignment of a node relies on its parents The parents always have higher priority than the children A node close to a gateway will pick a channel earlier than those farther away.

In [16], a distributed CA protocol is proposed for a dual radio mesh network. [16] considers that the interfaces using different orthogonal channels from the same frequency band might interfere with each other unless they are separated by a sufficient distance In order to solve the problem, they assume that the number of interfaces that nodes can equip is practically two, and nodes utilize channels in two different frequency bands on each of their interfaces to reduce interference. Thereby, each gateway in a mesh network associates a channel sequence presenting channels in different frequency bands alternatively with each of its interfaces The channel sequence is propagated along with routing information in periodic route announcement messages A node obtains channels in two different frequency bands

based on the channel sequence and the distance (hops) to the gateway. The nodes on the same hops from a gateway share one channel in common, then all paths to the gateway can operate on distinct channels to eliminate intra-path interference Compared with [19], the CA approach does not rely on the parent. However, if a gateway changes its channel sequence, the nodes connected to the gateway need to change channels accordingly.

CHAPTER 3

SIMULATORS

This chapter introduces OMNeT++ and INET in which the MIMC-SIM framework has been developed and compares OMNeT++ with other simulation tools on aspects including model design, performance, experiment design, and debugging

3.1 OMNET

OMNeT++ [8] is an open-source discrete event simulation environment. It is not a simulator of any particular system, but rather provides a generic and flexible architecture for writing simulation tools It has been used to model and simulate communication networks, operating systems, hardware architectures, distributed systems, and so on. Although OMNeT++ is not a network simulator itself, it has been widely utilized as a network simulation platform Moreover, OMNeT++ has been one of the alternative simulators against open-source research oriented simulator NS-2 [6] and the commercial software OPNET [9]

The most important feature of OMNeT++ is its object-oriented component architecture In OMNeT++, network components, such as network layers, network protocols, or network nodes, are composed hierarchically by modules Modules are classified into simple modules and compound modules. A simple module is the lowest level module which implements actual activities of the module. A compound

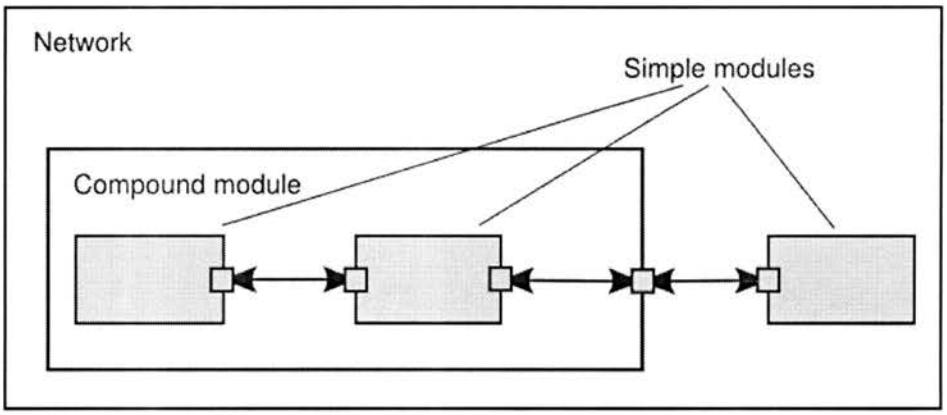


Figure 3.1: Module Structure in OMNeT++ [8]

module does not define actual activities, but combines simple modules to act like a network component. The compound modules can be combined into an even larger compound module. Figure 3.1 shows the hierarchy of simple modules and compound modules. Boxes represent modules, and small squares represent gates through which modules are connected. Arrows connecting boxes represent connections between modules. Using this architecture, the logical structure of an actual system can be efficiently described [21]. In OMNeT++, the structure of modules are described in the NED language, which is OMNeT++'s high-level language. NED is used to define simple modules, and combine them into compound modules. The modules defined in NED can be reused in any other compound modules. The actual activities of simple modules are written in C++, using the OMNeT++ simulation class library.

The fundamental ingredient of OMNeT++ making itself distinguished from other simulators is the message passing mechanism. In OMNeT++, modules do not call other modules' functions directly. Instead, modules communicate by exchanging

messages, where messages may carry arbitrary data structures, for instance, data packets for network communication Modules usually pass messages along predefined connections via gates, but it is also possible to directly send messages to destination modules without the predefined connections. Messages can be easily defined in msg files by using message definition function provided by OMNeT++

3.2 INET

The INET framework is an open-source communication network simulation package built in the OMNeT++ simulation environment [3] The INET framework contains models for various networking protocols, such as UDP, TCP, IP, IEEE802.11, and etc, and several application models The INET framework also supports wireless and mobile simulations as well. Protocols are represented as modules, and the modules are combined to construct hosts and network devices including a router, a switch, an access point, and so on Using INET with OMNeT++, various types of a network can be implemented and simulated In fact, various extensions have been already added into INET [3] INETMANET [4] is a project to model mobile ad hoc network protocols in the INET framework, and OverSim [10] is a project to model overlay and P2P network protocols The MIMC-SIM framework is also an extension in INET to model MIMC network protocols

Figure 3 2 shows the internal structure of a mobile host which composes a wireless network in INET The structure of the mobile host is founded to develop the MIMC-SIM framework. In the rest of this section, modules constructing the mobile host are briefly explained Then, interactions among modules are clarified.

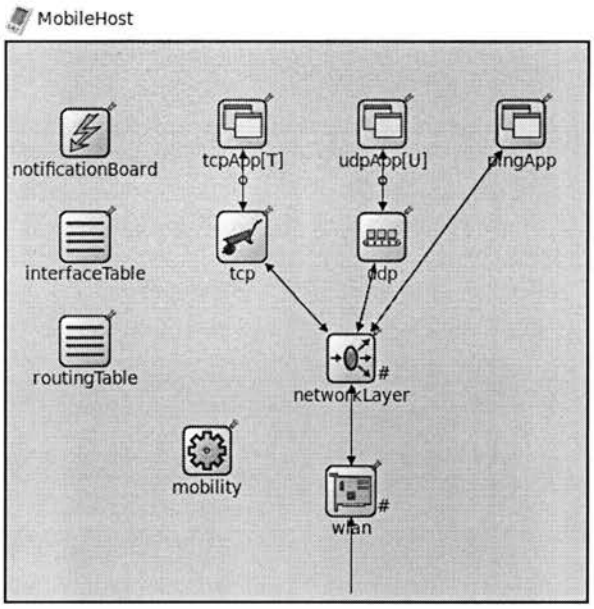
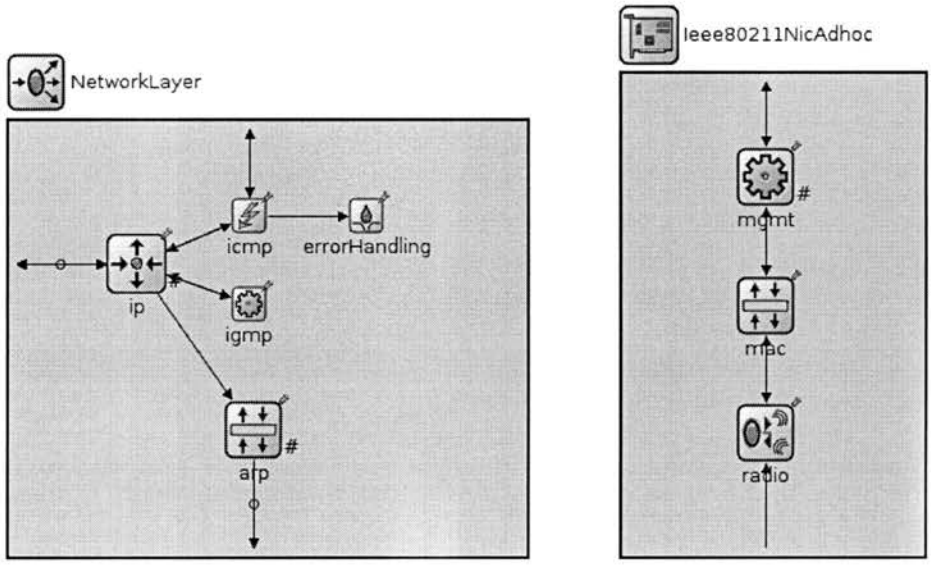


Figure 3.2: Mobile Host Sturcture in INET

3.2.1 Modules

Inside the host of Figure 3.2, some of the modules represent network protocols and are connected according to their associated layers. For example, the *tcpApp*, *udpApp*, *pingApp* modules at the top represent the application layer. The *tcp* and *udp* modules and the *networkLayer* module in the middle implement protocols at the transport layer and the network layer respectively. The *wlan* module at the bottom resembles a network interface card in the host and implements protocols at the link layer and the physical layer. Furthermore, the *networkLayer* and *wlan* modules are compound modules which are embodied in Figure 3.3(a) and Figure 3.3(b) respectively. In Figure 3.3(a), each module represents a protocol as named for itself. For example, the *ip* module implements IP protocol. In Figure 3.3(b), the *radio* module represents a physical radio, and the *mac* module implements the MAC



(a) Network Module (b) Wlan Module

Figure 3.3: Inside Architecture of Network Module and Wlan Module

protocol. And the *mgmt* module manages those two lower modules for ad-hoc mode.

In addition, in Figure 3.2, a host node includes additional modules which support other modules to collaborate together, hold data, or move a mobile host node around. These modules do not implement specific network protocols. For example, the *notificationBoard* module allows modules to notify each other about their events. When a module notifies of an event, the *notificationBoard* module disseminates the event to other modules. The *interfaceTable* module maintains such information as IP address, MAC address, MTU, etc, of network interfaces in a node. The *interfaceTable* module provides such information to other modules. The *routingTable* module maintains a routing table. The route of a outgoing packet is decided according to the routing table in the *routingTable* module. The mobility module deals with movement of a mobile node. This module constantly changes the

position of its host node in a network during simulation

3.2.2 Interaction

In INET, modules can interact with each other by three different mechanisms: message pass, direct access, and notification. The first mechanism, message pass, is provided in OMNeT++. Modules connected via gates usually pass messages to communicate with each other. This mechanism is best for the process of packet transmission. For example, in Figure 3.2, when the *tcpApp* module sends a message (which is referred to a packet in network transmission) to the tcp module, the tcp module deals with the message according to the TCP protocol and sends it to the *networkLayer* module. Then, the *networkLayer* module deals with the message according to a network layer protocol, such as the IP protocol, and sends it to the *wlan* module.

The second mechanism, direct access, is to interact with the modules not connected via gates, such as the *notificationBoard*, *interfaceTable*, and *routingTable* modules. Such modules are directly accessed by calling the access function built upon the *ModuleAccess* class in other modules. Then, all functionality of such modules can be utilized by other modules. For example, information of the routing table in the *routingTable* module can be retrieved through this mechanism.

The last mechanism, notification, is for modules to notify each other about their events, for instance, NIC configuration change, routing table change, mobile node position change, a state of a module change, communication failure, and so on. The

notification mechanism is handled by the *notificationBoard* module When a module wants to notify other modules of an event, the module accesses the *notificationBoard* module and let it diffuse the event to modules interested in learning about the event with additional information. Events that can be notified via the *notificationBoard* module are referred to notifications The notifications are identified by their categories which are maintained in the *notificationBoard* module according to kinds of events Using the notification mechanism, a module can interact with multiple modules at once.

3.3 Other Simulators

Besides OMNeT++, quite a few open source based simulators have been developed in the network research area Among all, NS-2 is the most widely used, and NS-3 is the successor of NS-2 with better features. Nevertheless, the MIMC-SIM framework is developed in the environment of OMNeT++ with INET. In this section, the network simulation tools, NS-2 and NS-3, are briefly introduced and compared to OMNeT++ with a focus on several views design structure, performance, and experimental environment in order to show that OMNeT++ has better features

3.3.1 NS-2

NS-2 is the most widely used network simulator in the network research area [23] NS-2 is a discrete event simulator that supports the simulation of TCP, routing, and multicast protocols over wired and wireless networks [6]. NS-2 uses C++ code for implementing the core part of a simulation, such as behavior of a system, and OTcl

scripts for configuring the system, such as a network topology This design structure saves resources from unnecessary recompilations if something has been changed in the simulation set-up However, the structure has drawbacks· the OTcl script makes the simulation slow down [24]

3.3.2 NS-3

NS-3 is also a discrete event simulator designed for the network research It is the next generation of NS-2 However, the architecture of NS-3 is much different from NS-2 [7] In order to abandon the problem caused by using OTcl scripts in NS-2, NS-3 relies entirely on C++ for implementing the simulation with optional Python bindings [25] Therefore, models in NS-2 cannot be reused in NS-3 without porting properly Even though many improvements have been made in NS3 in terms of performance and scalability, NS-3 is still under development Since NS-3 does not provide sufficient models to implement MIMC networks, only NS-2 is considered to compare with OMNeT++ in the next section

3.4 Comparison

3.4.1 Model management

OMNeT++ has a clear boundary between the simulation kernel and module implementation The OMNeT++ simulation kernel consists of a class library on which modules are implemented [23]. The OMNeT++ kernel generates modules as executable by compiling and linking them against the class library [21] In this

structure, the class library does not need to be modified to implement new modules Hence, OMNeT++ provides good features in terms of integrity and reusability In NS-2, in contrast, the boundary between the simulation kernel and modules is unclear [22] In NS-2, modules are usually generated by modifying the pure kernel a bit to adapt their activities Because of that, it is hard to maintain the kernel of NS-2 constantly In addition, after many modifications of the kernel, it will be difficult for other developers to reuse the kernel. This limits the reusability of NS-2

3.4.2 Programming Model

OMNeT++ separates clearly implementation of activities of modules and configuration of modules As mentioned in Section 3.1, OMNeT++ uses two different languages C++ and NED C++ is used to implement activities of modules, and NED is used to configure modules Since OMNeT++ manages the two languages in different roles clearly, the boundary between two languages is clear. NS-2 also provides the two different languages· C++ and OTcl. In NS-2, basically, C++ is used to implement activities of components, and OTcl is used to configure the network topology. However, NS-2 allows activities of components to be implemented in OTcl This blurs the boundary of the two languages Also, it is difficult for developers to track codes

3.4.3 Performance

Network simulators' ability to run huge scale networks are considered in terms of performance. According to [25], OMNeT++ can simulate huge scale networks up to

the limitation of the virtual memory capacity of a system, whereas NS-2 is not suitable to simulate the large network topologies. Figure 3.4 shows the simulation runtime measured at different network sizes for the compared simulators. It shows that OMNeT++ provides better performance than NS-2 for large size networks. This is because OMNeT++ maintains the set of future events in a binary heap [8], while NS-2 maintains it in a linked list.

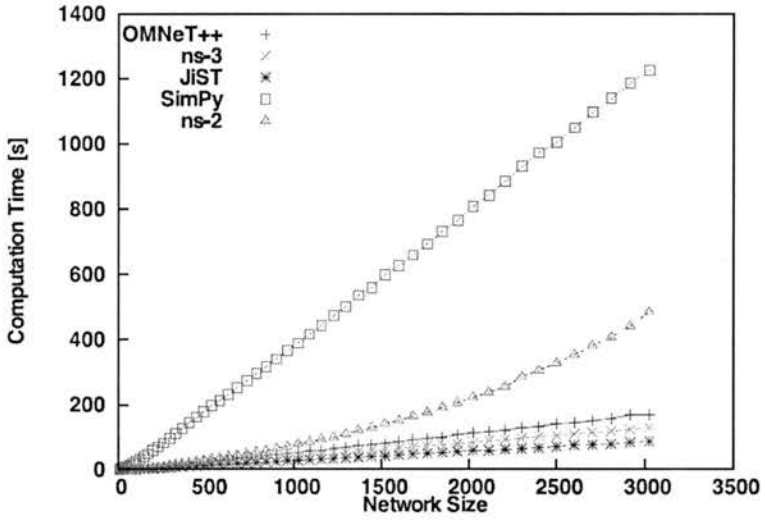


Figure 3.4: Simulation runtime [25]

3.4.4 Experiment Design

In order to experiment in various settings efficiently, parameters of experiments need to be separated from models. OMNeT++ separates experiments from models by using .ini files (text files) where parameters of a simulation experiment are written. In NS-2, in contrast, the experiment part mingles with models. For example, parameters of a simulation experiment are embedded in the OTcl scripts where the network topology is also defined. Therefore, the way to change the

parameters in NS-2 is not easy as in OMNeT++.

3.4.5 Debugging

Debugging in a network simulation is not only debugging code, but also tracing variation of a network simulation [21]. OMNeT++ provides very powerful GUI (Figure 3.5), showing packet transmissions and network status while a simulation is running. Using the GUI, OMNeT++ allows users to check the process of simulation of networks visually, and also have ability to control the network by changing parameters during simulation. In contrast, NS-2 also provides a GUI, called *nam*, to allow users to trace the process of a network simulation. However, the process of a network simulation can be visualized only after a network is completely simulated. Compared to OMNeT++, NS-2 does not provide functionality to debug during simulation.

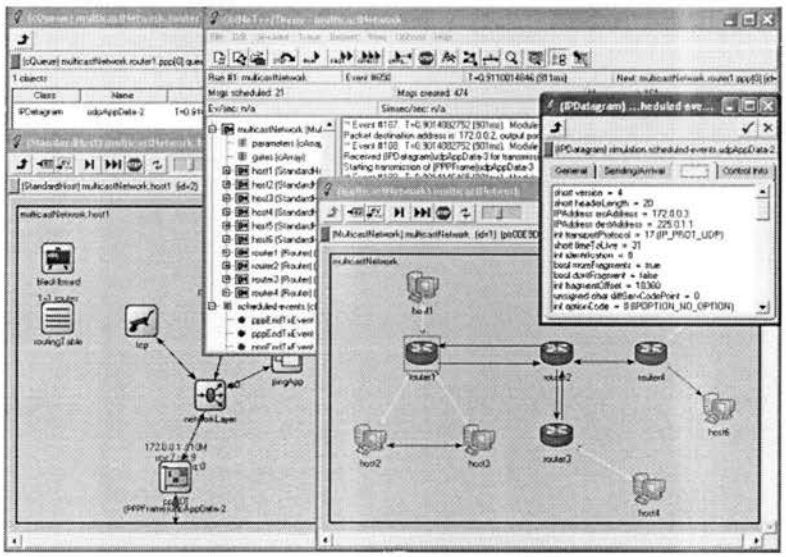


Figure 3.5: Graphical runtime interface in OMNeT++ [8]

CHAPTER 4

OVERVIEW OF MIMC-SIM

This chapter discusses assumptions used in the MIMC-SIM framework, the main challenges, and CA issues in designing the MIMC-SIM framework. In addition, this chapter presents the overall architecture of the MIMC-SIM framework.

4.1 Assumptions

The MIMC-SIM framework assumes that MIMC networks utilize multiple orthogonal channels In the current implementation of INET, this assumption is well supported by the signal propagation model adapted in the *radio* module A signal delivered in one channel does not contribute anything to another orthogonal channel In the future, the *radio* module can be modified to adapt a better signal model to capture the major characteristics of signals in overlapping channels.

The MIMC-SIM framework assumes all NICs are using the same communication protocol or compatible protocols in the same protocol family. For example, in a mesh network, a node can be equipped with two NICs One NIC may work on IEEE802 11b and the other may work on IEEE802.11g The assumption implies that a packet transmitted in a channel could be delivered to all NICs in that channel. If different communication protocols with overlapping channels are used, a signal that one protocol transmits a packet in one channel becomes a noise signal at

other protocols using the same channel The current *radio* module in INET does not support concurrent multiple communication protocols.

Even though INET allows nodes to assign multiple IP addresses with multiple NICs, the MIMC-SIM framework assumes that each node is identified by the unique IP address In simulation, it is assumed that all nodes in a MIMC network are in the same subnet network, which means all NICs of each node are in the same subnet network This assumption allows nodes to communicate with one IP address over multiple MAC addresses For example, although a node sends a ping echo packet out via one specific NIC, the node can receive the ping reply packet via another NIC whose MAC address is different from the first one. Mapping a single IP address to multiple MAC addresses in a node makes a routing algorithm easy to be implemented in the MIMC-SIM framework

The MIMC-SIM also assumes the number of channels is usually greater than the number of NICs in nodes Researchers [16, 13, 15] have shown that multiple NICs of a node should be separated by at least 18 inches so that their radio transmission does not interfere with each other even though they use different orthogonal channels. Hence, given the limit size of most mobile devices, a node could have only a few NICs (mostly two or three) Whereas, wireless networks often have more orthogonal channels For example, IEEE801.11b/g has 3 orthogonal channels, IEEE802.11a has 13, and IEEE802.15 4 has 16.

4.2 Challenges

Although INET can support partially multiple interfaces and multiple channels in network simulation, quite a few challenging issues remain unaddressed for MIMC network simulation due to two major reasons One is that the wireless framework in INET was basically designed for simulating wireless communication in one channel Even though it allows NICs to use multiple channels, it assumes that all NICs of the same host use the same channel and work on the same mechanism in simulation The other reason is that INET handles multiple NICs in wireless communication directly based on the model of wired network, which simply makes the NICs forward packets over separated communication links. In a MIMC network, such a model ignores the collaboration among the NICs, thus, it cannot be used to support MIMC simulation.

4.3 Issues of Channel Assignment

In order to develop a general framework that adopts various requirements of MIMC networks, the MIMC-SIM framework is designed for addressing four major issues of simulating CA protocols.

First, CA protocols assign channels to nodes in various ways and assign various roles to NICs accordingly. The MIMC-SIM framework is designed to support two major categories of CA protocols. One category is node-based channel allocation [18, 26]. It assigns a set of channels to each node, and nodes usually receive packets on their assigned channels. In this category, a receiving node guarantees that it

always receives packets on a particular channel, and a sending node tunes its channel to a receiver's channel to deliver packets The NIC used to receive packets on a particular channel is known as a receiving-NIC, and the NIC used to send packets is known as a sending-NIC. The other category is link-based channel allocation [16, 19]. It assigns channels to links, and nodes on a link use the channel assigned the link In these category, NICs in a node are classified into two different groups, for example, up-link NICs and down-link NICs, according to the routing topology of a network. In this type of network, a node assigns its up-link channel according to its parent node, and assigns its down-link channel according to a CA protocol. The framework should support nodes to manage their NICs with different roles.

Second, the MIMC-SIM framework needs to handle issues including the mapping between a MAC address and an assigned channel. In a MIMC network, a NIC is always uniquely identified by its MAC address, while a node could be identified by a single IP address. When a node sends a packet, the packet carries the IP addresses of the destination and the next hop. The sending node needs to resolve the MAC address of the next hop NIC with the next hop IP address and the associated channel information with the MAC address. As NICs could switch on different channels, the CA protocol needs to help nodes maintain channel information associated with their next hop NICs. Hence, the MIMC-SIM framework needs to properly maintain IP addresses of nodes and MAC addresses and channel information of their NICs to support CA protocols.

Third, a CA protocol needs to interact with other protocols, beyond simply making NICs forward packets. A CA protocol is placed between the network layer

and the MAC layer and works with various MAC protocols, IP protocols, routing protocols, and ARP protocols To achieve this, the framework needs to identify the components in CA protocols that are independent of other protocols Meanwhile, the framework should provide mechanisms for these protocols to interact so that a CA protocol can work with a specific MAC protocol or network protocol. Furthermore, CA protocols do not only interact with other protocols, but also integrate them such as ARP protocols and routing protocols to adopt its own algorithm.

Finally, a variety of CA protocols have been proposed in the past. The MIMC-SIM framework shall provide a coding structure that accommodates common features shared among these protocols and allows flexible extension to implement specific protocol behaviors as well Many CA protocols can be modeled by an operation plane and an algorithm plane The operation plane specifies the operations which are fundamental activities of CA protocols, such as tuning a channel to a radio, scanning a particular channel, and transmitting a data packet in an appropriate channel The algorithm plane manages the way to exchange channel information between nodes, and computes the channel allocation based on the channel information collected using the operation plane. Although a particular CA protocol always differs from other CA protocols in many details, they share some common procedures of executing operations and algorithms. Hence, the MIMC-SIM framework utilizes these observations to structure its architecutre.

4.4 Architecture of MIMC-SIM

To address the aforementioned issues of CA protocols in MIMC network simulation, the MIMC-SIM framework defines a new host structure as shown in Figure 4.1. Compared with the typical host in INET depicted in Figure 3.2, the MIMC-SIM framework adds the new module, *MIControl* (named after multi-interface control), where CA protocols are adopted. The *MIControl* module is placed as a new layer between the *networkLayer* module and the *wlan* modules, which represent the network layer and the MAC layer respectively.

The new structure allows CA protocols to work independently with various MAC protocols and IP protocols. Since the *MIControl* module is separate from the *networkLayer* module and *wlan* modules, the *MIControl* module does not

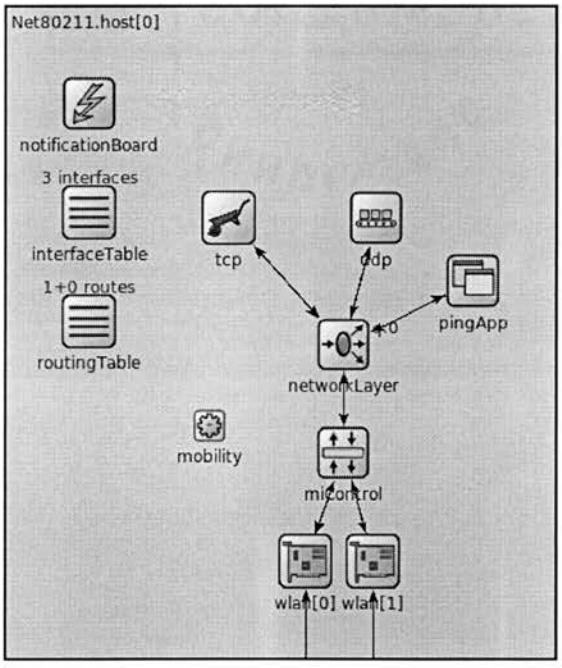


Figure 4.1: Mobile Host Sturcture in the MIMC-SIM Framework

participate in their process, but cooperates with them by exchanging messages to perform CA protocols Even if the instances of the *networkLayer* module and the *wlan* module are changed, the *MIControl* module can still perform its operations without modification. Moreover, the new structure allows CA protocols to deal with routing issues for sending packets. When the *MIControl* module receives packets from the *networkLayer* module, it can replace the routing information of packets decided in the *networkLayer* module with new routing information according to CA protocols

In addition, compared to the original host structure in INET, the new host can have multiple *wlan* modules and coordinate them. In the new host structure, CA protocols can easily coordinate multiple *wlan* modules by the *MIControl* module For example, the *MIControl* module can forward packets from the *networkLayer* module to a particular *wlan* module for sending, while *MIControl* uses another *wlan* module only for receiving

Figure 4 2 shows the inside structure of the *MIControl* module. The *MIControl* module is constituted with four kinds of modules: *mainControl*, *subControl*, *neighborTable*, and *nicTable*.

4.4.1 mainControl

The *mainControl* module implements the algorithm plane of CA protocols. It composes channel management packets, coordinates multiple *subControl* modules, collects neighbor nodes' information and analyzes them, updates routing information, and decides routes and a proper *wlan* module for outgoing packets.

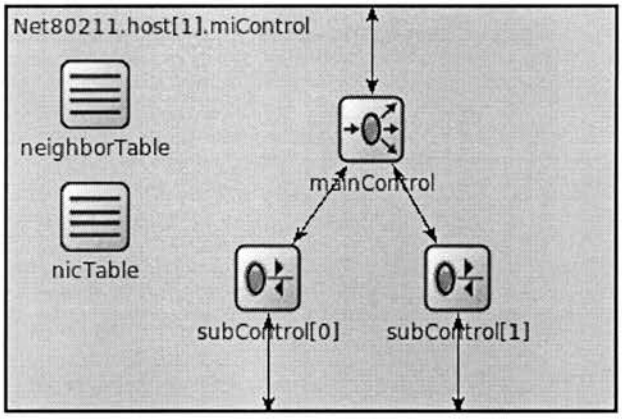


Figure 4.2: *MIControl* Module Sturcture

Moreover, the *mainControl* module manages *neigbhorTable* by updating neighbor nodes' information. A new CA protocol can be adopted into the simulation framework by implementing a new *mainControl* module. Researchers can simply extend the base class of the *mainControl* module to adopt their own CA protocols.

4.4.2 subControl

The *subControl* module implements the operation plane of CA protocols. Since the operation plane is independent to the algorithm plane of CA protocols, the *subControl* module is designed separately from the *mainControl* module. In addition, each *subControl* module corresponds to a specific *wlan* module because the process of CA operations is specific to an individual *wlan* module. The *subControl* module controls its corresponding *wlan* module to perfrom the CA operations. Thus, the same number of *subControl* modules are equipped as the number of *wlan* modules. The *subControl* module does not decide when to conduct CA operations. Instead, it receives commands from the *mainControl* module, and

performs CA operations based on the commands. The *subControl* module only decides the order of performing CA operations based on their given priority. In order to take CA operations from the *mainControl* module and control a *wlan* module, the *subControl* module is placed under the *mainControl* module and connected to an individual *wlan* module. The *subControl* module also guarantees that packets are transmitted on the correct channel by controlling its *wlan* module. Since CA operations the *subControl* module performs are independent to CA protocols, the *subControl* module is not required to be re-implemented for a new CA protocol.

4.4.3 neighborTable

In MIMC networks, no matter what CA protocol is used, nodes collect and maintain the information of their neighbor nodes. The *neighborTable* module is designed to maintain such information. This module is directly accessed by other modules using the direct access mechanism (Section 3.2.2). Both the *mainControl* module and the *subControl* module access the *neighborTable* module to update neighbors' information or retrieve the information. The *neighborTable* module does not participate in forwarding packets in the *MIControl* module. Since each CA protocol requires different information for neighbor nodes, this module shall be extended to store proper information according to a CA protocol

4.4.4 nicTable

The *nicTable* module is designed to maintain various roles and information of NICs. CA protocols allow nodes to utilize their NICs in different roles, for instance, upper

NIC and down NIC [19] Also, CA protocols allow nodes to assign specific channels to their NICs. The *nicTable* module maintains roles and channel information of NICs defined by a CA protocol. The *nicTable* module is directly accessed by other modules using the direct access mechanism (Section 3 2 2) Usually, the *mainControl* module accesses the *nicTable* module to retrieve the information, while the *subControl* module accesses it to update Similarly to the *neighborTable* module, also, the *nicTable* module does not participate in forwarding packets The *nicTable* module will be extended to define new roles and store proper information of NICs according to a CA protocol

4.5 Messages

In OMNeT++, modules connected via gates pass messages to communicate with each other The MIMC-SIM framework classifies those messages into command, channel management packet, and data packet.

In the MIMC-SIM framework, the *mainControl* module controls the *subControl* module to perform CA operations by sending special messages. Such special messages are referred to CA commands. The MIMC-SIM framework defines the *MICommand* class as the base class on which a specific CA command will be implemented In addition, the *subControl* module sends messages to its *wlan* module to configure the channel information of the *wlan* module. Such messages that one module sends to another module to use its service are considered as commands in the MIMC-SIM framework

In MIMC networks, nodes exchange their information such as IP address, MAC

addresses, channel information, traffic information, and so on. In the MIMC-SIM framework, channel management packets are the messages carrying such information and transmitted among nodes. Channel management packets are specified by CA protocols and generated in the *mainControl* module. In order to handle channel management packets, the MIMC-SIM framework defines the *MIPacket* class as the base class on which any kinds of channel management packets will be implemented.

Data packets are generated at or above the network layer In the MIMC-SIM framework, all the messages received in the *MIControl* module from the networkLayer module are regarded as data packets. Processing data packets is similar to the processing in INET, but the MIMC-SIM framework ensures that each data packet is transmitted on the correct channel.

Since the MIMC-SIM framework is built atop the MAC layer, both channel management packets and data packets are considered as data frame at the MAC layer For example, if the underlying MAC protocol is IEEE802.11, the two types of packets will be formatted as IEEE802 11 Data Frame. So, in the MIMC-SIM framework, channel management packets and data packets are simply referred to packets.

4.6 State Machine

As any other network protocols, CA protocols can also be modeled in a finite state machine (or state machine), which is used for computer programs In the MIMC-SIM framework, to adopt a CA protocol, the *mainControl* module is implemented based on a state machine. In addition, the *subControl* module is also

implemented based on its own state machine. A state machine can be described using a state diagram which abstractly describes a state machine. To implement a state machine as described in a state diagram, the MIMC-SIM framework provides a set of predefined macros, named FSME (named after Enhanced Finite State Machine). FSME is inspired by FSMA, which is also a set of predefined macros handling a state machine in INET. FSME provides a generic code style and flexible extension for implementing a state machine. Compared to FSMA, FSME allows each state in a state machine to be implemented separately. The details of FSME are explained in Chapter 8.

CHAPTER 5

MAINCONTROL

In this chapter, the design and basic operations of the *mainControl* module are presented based on the state machine of the *mainControl* module

The *mainControl* module is the core module implementing actual CA protocols in the MIMC-SIM framework. It is built upon the MIMainControlBase class, which implements a set of abstract functions that perform CA operations and a few basic INET functions that initialize the module and pass messages to proper functions. Moreover, the MIMainControlBase class defines a set of functions using FSME functions to deal with a finite state machine. The MIMainControlBase class is designed to support common features of CA protocols so that child classes can utilize them. As shown in Figure 5.1, the MIMainControlBase class is extended from the cSimpleModule class which provides basic features of a simple module and the INotifiable class which deals with notification function in INET. Then, the MIMainContrlBase class shall be extended to a child class to implement a specific CA protocol For example, the DrcaMainControl class and the ScodeMainControl class extends the MIMainControlBase class to implement the distributed routing/channel algorithm in a MIMC network [19] and the CA protocol based on superimpose code [26] respectively.

The process of the *mainControl* module is represented in a state machine.

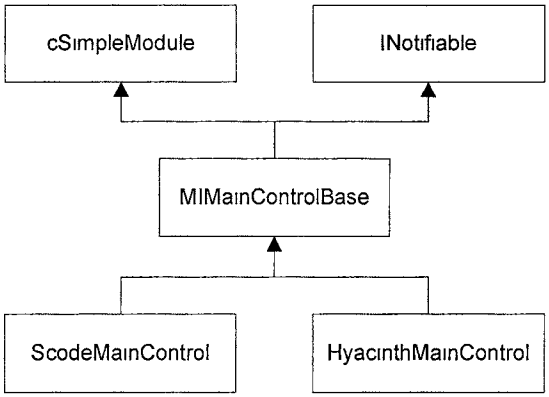


Figure 5.1 Class Hierarchy Of *mainControl* Module

According to a state machine, the *mainControl* module determines when and how to perform the following major operations for a CA protocol

- Commanding CA operations
- Handling channel management packets
- Computing channel and route
- Transmitting packets

5.1 State Machine

The *mainControl* module implements a state machine to adopt a CA protocol logically The process of a CA protocol can be represented in a state machine Since the process of each CA protocol is unique, implementation of a state machine is specific to a CA protocol. However, the state machines of CA protocols can be generalized into the five states: INIT, SCAN, ASSIGN, SETNIC, and NORM Figure 5 2 shows the state diagram of the five states in the *mainControl* module conceptually. Even though CA protocols have their own specific state diagrams,

their state diagrams can be generally described within the five common states. According to the five states, CA operations are performed in proper manner. The state diagram of a new CA protocol shall be depicted based on Figure 5.2 in the MIMC-SIM framework.

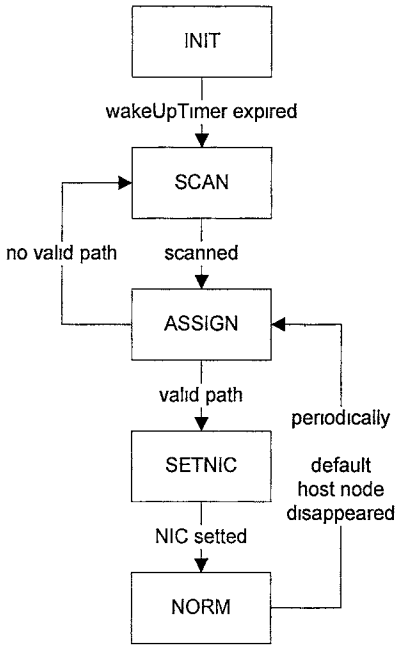


Figure 5.2: State Diagram Of The MainControl Module

In Figure 5 2, INIT is the beginning state when a node is initialized and substantiated itself When a node is in the INIT state, although the node has already initiated in simulation, the node is regarded as inactive. As long as a node stays in the INIT state, a node ignores all the incoming packets from other nodes and does not send any packets or perform any operations Thus, other nodes will not be able to find the inactive node The INIT state is designed for a simulation reason. When a network is initiated in simulation, all nodes are initiated at the same time However, nodes may join a network in arbitrary time point In the

MIMainControlBase class, the timer, *wakeUpTimer*, is defined and scheduled when a node is initiated. When *wakeUpTimer* is expired, the node is regarded as active when actually starts performing CA protocols in a network. However, this state may not be considered in real network environment.

The SCAN state comes after the INIT state. It is triggered when *wakeUpTimer* is expired in the INIT state. Basically, CA protocols allow nodes to listen to the medium to find potential neighbor nodes and gather information from them. The SCAN state is designed for that reason. In the SCAN state, the *mainControl* module commands the *subControl* modules to scan channels instead of scanning by itself. During scanning, the *mainControl* module receives channel management packets from neighbor nodes and updates them into the *neighborTable* module. When the *mainControl* module is notified of the completion of scanning by the *subControl* modules, the ASSIGN state is triggered.

When a node enters the ASSIGN state, a node assumes that sufficient channel information is collected in the SCAN state. In the ASSIGN state, based on the channel information gathered from neighbor nodes, a node computes a channel to assign and decides a default route according to a CA protocol's algorithm. With the computed results, a node tries to join a network. In some CA protocols, a node may ask its expected default host node (parent node) to join a network in this state. If a node confirms to join a network according to a CA protocol, the node enters the SETNIC state. Otherwise, the node either computes the channel and default route once again to find another parent node or goes back to the SCAN state to collect new channel information.

In the SETNIC state, a node sets its NICs into the channel computed in the ASSIGN state accordingly If a node has encountered this state before since the node started up, then the node may have a problem to set up its NICs because there might be an ongoing CA operation or a packet transmission in its NICs Forcing the NICs to change a channel immediately beyond a current work may cause such problems as a packet loss or a deadlock problem In order to prevent the conceivable problems, the MIMC-SIM framework waits until a NIC completes its current work and assigns a channel to the NIC The actual work is handled in the *subControl* module. The *mainControl* module only commands a *subControl* module to set up its NIC and waits for the notification indicating that the *subControl* module completes setting the NIC.

When a node sets up its NICs appropriately, the NORM state is finally triggered Entering the NORM state means that a node joins a network and becomes ready to communicate via a network. In this state, nodes not only transmit packets, but also do some CA operations to maintain a CA protocol Usually, nodes keep their neighbor nodes and collect new channel information from neighbor nodes. As shown in Figure 5 2, the NORM state goes back to the ASSIGN state periodically, or especially when a node has a broken link with its parent node Since nodes update information of their neighbor nodes in the NORM state, nodes need to re-estimate their channels and routing based on the new updated information periodically In addition, the broken link to a parent node makes a node unable to connect to a gateway, and it decreases the throughput of a network. Thus, it is necessary to have the routine to go back to the ASSIGN state so that

nodes are able to re-estimate their channels and routing for applying new channel information and recovering their default routes

5.2 Commanding CA Operations

In MIMC networks, in order to manage CA protocols, nodes perform such CA operations as assigning a channel to a NIC, scanning and probing a channel. Both scanning and probing are to listen to a particular channel, but the difference between them is that probing broadcasts request packets during listening a channel while scanning does not. In the MIMC-SIM framework, instead of the *mainControl* module executes the CA operations, the *mainControl* module sends CA commands to the *subControl* module Then, the *subControl* module actually executes the CA operations according to CA commands. The *mainControl* module builds CA commands upon the *MICommand* class which the *subControl* module perceives and executes them accordingly. The *MICommand* class is defined as shown in Figure 5.3. In the *MICommand* class, when the *priority* variable is false, the CA

```
class MICommand    public cMessage {
public:
        enum MICommandKind{
                C_SETNIC,
                C_SCAN,
                C_PROBE,
        },

protected
        bool priority;
        .
        :
};
```

Figure 5.3: The *MICommand* Class

```
MICommand* MIMainControlBase::
    buildSetNicCommand(int channel, double bitRate, bool priority)
{
        MICommand *cmd =
        new MICommand("C_SETNIC", MICommand::C_SETNIC, priority),

        ChannelInfo *chInfo = new ChannelInfo();
        chInfo->setChannel(channel);
        chInfo->setBitrate(bitRate),

        cmd->setControlInfo(chInfo),
        return cmd;
}
```

Figure 5 4: The *buildSetNicCommand* Function

command has higher priority to be executed The enumeration declaration, *MICommandKind*, is used to identify kinds of CA commands. The CA command does not contain specific information to control CA operations Instead, it only carries control information which the *subControl* module performs CA operations according to. To build CA commands, the *MIMainControlBase* class provides a set of functions: *buildSetNicCommand*, *buildScanCommand*, and *buildProbeCommand.*

The *buildSetNicCommand* function is to build a CA Command to assign a channel to a NIC The CA command is identified as C_SETNIC. The C_SETNIC

```
class ChannelInfo{
public:
        short channel;
        double bitrate,
        .
        .
}
```

Figure 5.5: The *ChannelInfo* Class

```
MICommand* MIMainControlBase
    buildScanCommand(int beginChannel, int endChannel,
    double bitRate, double duration, bool priority, bool isPmode)
{
        if(beginChannel > endChannel)
                opp_error("illogical channel range"),

        MICommand *cmd =
        new MICommand("C_SCAN", MICommand..C_SCAN, priority),

        ScanControlInfo *ctrl = new ScanControlInfo();
        for(int i=beginChannel, i<=endChannel, i++){
                ChannelInfo *chInfo = new ChannelInfo(),
                chInfo->setChannel(i);
                chInfo->setBitrate(bitRate);
                ctrl->addChannelInfo(chInfo),
        }
        ctrl->setDuration(duration);
        ctrl->setPromiscuousMode(isPmode);

        cmd->setControlInfo(ctrl),
        return cmd;
}
```

Figure 5 6 The *buildScanCommand* Function

command carries a channel information which is defined in the *ChannelInfo* class The *ChannelInfo* class is declared as shown in Figure 5.5 and used to contain a channel number and its data rate When the *subControl* module receives the C_SETNIC command, it retrieves the channel information from the command, and assigns a channel to its NIC according to the channel information This function is usually called in the SETNIC state Figure 5 4 shows the implementation of the *buildSetNicCommand* function

The *buildScanCommand* function is to build a CA command to scan a set of channels The CA command is identified as C_SCAN. The C_SCAN command carries a control information defined in the *ScanControlInfo* class. The

ScanControlInfo class is defined as shown in Figure 5.7 and used to control not only scanning, but also probing. The control information contains a series of channels, bit rate of channels, and duration of scanning each channel In addition, it can enable scanning in promiscuous mode. The *subControl* module scans according to the control information that the C_SCAN command carries Figure 5 6 shows the implementation of the *buildScanCommand* function.

```
class ScanControlInfo{
public:
  std::vector<ChannelInfo *> ChannelInfoSeq;
  bool promiscuousMode;
  double duration;
  cMessage *probeMsg;
  int repeatMsgTime;

        .
}
```

Figure 5.7. The *ScanControlInfo* Class

```
MICommand* MIMainControlBase::
    buildProbeCommand(int beginChannel, int endChannel,
    double bitRate, double duration, cMessage* msg,
    int repeatMsg, bool priority, bool isPmode)
{
        MICommand *cmd = buildScanCommand(beginChannel, endChannel,
        bitRate, duration, priority, isPmode);
        cmd->setName("C_PROBE");
        cmd->setKind(MICommand::C_PROBE);

        ScanControlInfo *ctrl =
        check_and_cast<ScanControlInfo *>(cmd->getControlInfo());
        ctrl->setProbeMsg(msg);
        ctrl->setRepeatMsgTime(repeatMsg);

        return cmd;
}
```

Figure 5.8. The *buildProbeCommand* Function

The *buildProbeCommand* function is to build a CA command to probe a set of channels. The CA command is identified as C_PROBE. Since probing is also to scan channels, the *buildProbeCommand* function simply calls the *buildScanCommand* function to set configuration for scanning. Then, it attaches a packet and iteration number of sending the packet to the control information. The *subControl* module probes according to the control information. Figure 5.8 shows the code of the *buildProbeCommand* function. Both the *buildScanCommand* and *buildProbeCommand* functions are used in the SCAN state usually

5.3 Handling Channel Management Packets

In a MIMC network, the *mainControl* module builds channel management packets upon the *MIPacket* class and sends them out according to a CA protocol. Figure 5.9 shows the definition of the *MIPacket* class It simply extends the *cPacket* class and adds one more member variable, *type*, indicating types of channel management packets. When the *mainControl* module receives channel management packets, it handles the packets in the *handleMIPacket* function, which is defined in the *MIMainControlBase* class. Since CA protocols define and handle channel management packets differently, the *handleMIPacket* function is declared as a pure

```
class MIPacket   public cPacket {
protected
        int type;
        :
        :
}
```

Figure 5.9: The *MIPacket* Class

virtual function Then, a child class shall extend the base class and instantiate the *handleMIPacket* function to handle channel management packets according to its CA protocol.

When the *mainControl* module handles channel management packets, it usually updates the information into the *neighborTable* module and the *routingTable* module. The *neighborTable* module can be easily updated using functions provided by the *neighborTable* module. For updating the *routingTable* module, instead of using the functions that the *routingTable* module provides, the *mainControl* module uses its own functions. *updateRoute*, *removeRoute*, *updateDefaultRoute*, which are implemented in the *MIMainControlBase* class. Figure 5.10 shows the declaration of the functions. The *updateDefaultRoute* function updates default route information and marks a default host node in the *neighborTable* module as well. The *updateRoute* function adds or updates routing information into the *routingTable* module. The *removeRoute* function removes routing information by given host IP address. These functions allow developers to utilize the *routingTable* module conveniently.

```
class INET_API MIMainControlBase{
protected
virtual void updateDefaultRoute(const IPAddress& gatewayIP,
  InterfaceEntry *ie),
virtual void updateRoute(const IPAddress& hostIP,
  const IPAddress& gatewayIP, const IPAddress& netMaskIP,
  InterfaceEntry *ie);
virtual void removeRoute(const IPAddress& hostIP);
       :
       :
```

Figure 5.10: The Declaration Of The Functions Updating The *routingTable* Module

5.4 Computing Channel and Route

CA protocols have their own specific channel assignment algorithm which allows nodes to find the best channel and route so that CA protocols can accomplish their goal, maximizing overall throughput of a network. In a MIMC network, when a node collects sufficient information of traffic and assigned channels from its neighbors, the node computes better route and a proper channel according to its CA protocol's channel assignment algorithm In the MIMC-SIM framework, the channel assignment algorithm is performed in the *mainControl* module. More specifically, the *assignChannelAndRoute* function is defined to implement the algorithm in the *MIMainControlBase* class. However, since the channel assignment algorithm of each CA protocol is unique, the *assignChannelAndRoute* function is declared as a pure virtual function Then, a child class shall extend the base class and instantiate the *assignChannelAndRoute* function to implement the channel assignment algorithm according to its CA protocol. The *assignChannelAndRoute* function is usually executed in the ASSIGN state.

5.5 Transmitting Packets

In a computer network communication, packets are delivered with routing information which is selected in the network layer In a MIMC network, however, routing for each packet is usually chosen by a CA protocol. To achieve this, the MIMC-SIM framework allows the *mainControl* module to deal with packets for routing in the NORM state. This section presents how the *mainControl* module

deals with packets in order of process

When a packet is delivered from the Network module for transmission, the *mainControl* module first stores the packet in *dataQueue*, a queue defined in the *MIMainControlBase* class, without any routing information In a MIMC network, CA protocols allow nodes to update the routing table and change a default route frequently The frequent change of the routing table may cause that a packet transmission does not reflect the latest routing table For example, suppose that a route of a packet is already chosen, and the packet waits to be delivered in a node Then, suddenly, the node updates its routing table. Based on the new routing table, the packet should be delivered on a different route However, because the route of the packet has been made already, the packet will be still delivered in the original route, which might cause a packet loss eventually. For this reason, the route of outgoing packets should be chosen right before they are transmitted. Hence, the *mainControl* module keeps outgoing packets in the queue without routing information until the packets are actually transmitted.

In order to know when the outgoing packets can be actually transmitted, the *mainControl* module tracks the amount of packets that each *subControl* module can transmit at once. To do so, first, the *MIMainControlBase* class defines *requestedPacket* as a vector whose element index matches to the index of *subControl* modules, and sets every element of *requestedPacket* to a certain amount of packets that a *subControl* module can transmit at once. When the *mainControl* module sends a packet to a *subCotnrol* module, the corresponding element of *requestedPacket* is decremented. When a *subControl* module completes

a cycle of transmitting a packet (finishes transmission of a packet), the *subControl* module notifies the *mainControl* module of the completion, and the *mainControl* module increments the corresponding element of *requestedPacket*. Hence, when an element of *requestedPacket* is positive, the corresponding *subControl* module is available to transmit a packet. Conversely, when the element is zero, the corresponding *subControl* module is not capable to transmit a packet. So, the *mainControl* module does not use the *subControl* module for transmitting packets. Since each *subControl* module intends to handle one packet at a time, every element of requested packet is set to one and not incremented over one. Moreover, by receiving the notification, the *mainControl* module is able to know when a particular *subControl* module becomes ready to transmit another packet.

When the *mainControl* module is available to send a packet out, first, the *mainControl* module checks *requestedPacket* to figure out which *subControl* module is available. Then, the *mainControl* module retrieves a packet supposed to be transmitted via the NIC, which associates with the available *subControl* module, from *dataQueue* based on the routing table. When a packet is retrieved, the *mainControl* module attaches control information built upon the *MIPacketCtrl* class, defined as depicted in Figure 5.11. The control information contains routing information and channel information and indicates importance of a packet. The channel information is used when the *mainControl* module forces a packet to be transmitted on a particular channel. The importance indicates whether a packet is a non failure-free packet. When a non failure-free packet transmission fails at the MAC layer, the *subControl* module will notify the *neighborTable* module of the

failure. Finally, the *mainControl* module sends a packet to the *subControl* module

```
class MIPacketCtrl{
protected:
        cPolymorphic *controlInfo;
        ChannelInfo *channelInfo;
        bool reliance;

        .
}
```

Figure 5.11: The Declaration Of The *MIPacketCtrl* Class

```
void MIMainControlBase::sendDataPacket()
{
  if(isEmptyQueue()) return;

  for(int i=0; i<numNics; i++)
    if(requestedPacket[i] > 0){
      cMessage *msg = dequeue(i);
      if(msg)
        sendDown(msg, i);
    }
}
```

Figure 5 12. Implementation Of The *sendDataPacket* Function

In the *MIMainControlBase* class, the *sendDataPacket* function is defined and implemented to handle packet transmission as the aforementioned process. Figure 5.12 is the implementation of the *sendDataPacket* function. In Figure 5.12, the *dequeue* function is to retrieve a packet as explained above. In a child class of the *MIMainControlBase* class, the *sendDataPacket* function will be simply called for a packet transmission.

CHAPTER 6

SUBCONTROL

In this chapter, the design of the *subControl* module is presented in detail.

6.1 Work Flow

The *subControl* module is designed to perform common CA operations, such as assigning a channel, scanning and probing a set of channels, and transmitting packets on the correct channel. Assigning a channel means to tune a NIC into a certain channel Scanning means to listen to a channel for a certain time Probing is basically similar to scanning, but the *subControl* module broadcasts packets before listening to each channel. Transmitting packets on the correct channel means to ensure that packets are transmitted on the correct channel of the right next hop node. Since such CA operations are independent to CA protocols, the *subControl* module provides such CA operations as tool kits that various CA protocols can utilize. Developers only need to decide how to use these tool kits in CA protocols, instead of mixing these operations within CA protocols

In order to perform CA operations, the *subControl* module interacts with the *mainControl* module and its corresponding NIC Figure 6 1 shows how the *subControl* module interacts with other modules to perform CA operations

First, the *subControl* module receives CA commands and packets from the

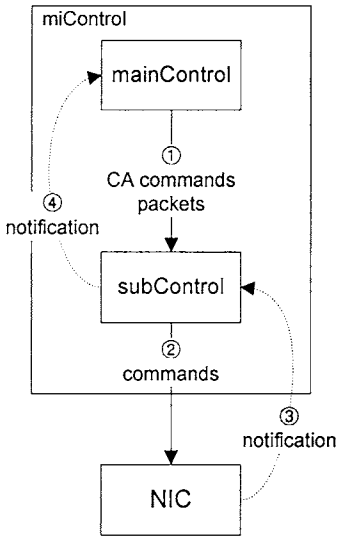


Figure 6.1: The Flow Of Commands Among Modules

mainControl module to activate CA operations. Although the *subControl* module performs the CA operations, it does not decide when to conduct CA operations. Instead, the *subControl* module only carries CA operations by receiving CA commands or packets from the *mainControl* module. In a MIMC network, CA operations are determined by a CA protocol. Since the *mainControl* module implements a CA protocol, the *mainControl* module takes charge of sending proper CA commands and packets in a proper manner according to a CA protocol. According to the CA commands and packets, the *subControl* module executes CA operations.

When the *subControl* module executes CA operations, it controls its corresponding NIC by sending a command. The command is to change such channel information as channel number and bit rate of a NIC. The *subControl* module implements the $sendRadioConfigMsg$ function which builds the command. In the

```
void sendRadioConfigMsg(ChannelInfo *channelInfo)
{
        if(channelInfo == NULL)
                return;

        PhyControlInfo *phyCtrl = new PhyControlInfo();
        phyCtrl->setChannelNumber(channelInfo->getChannel());
        phyCtrl->setBitrate(channelInfo->getBitrate());

        cMessage *msg =
         new cMessage("RadioConfigMsg", PHY_C_CONFIGURERADIO);
        msg->setControlInfo(phyCtrl);

        sendDown(msg);
}
```

Figure 6.2: The *sendRadioConfigMsg* Function

sendRadioConfigMsg function, the command is simply built as a message and identified as *PHY_C_CONFIGURERADIO*. Such command will be perceived as a command to change a channel and its bit rate in the *mac* module. However, the command only carries control information which contains specific channel information. The control information is built upon the *PhyControlInfo* class (provided in INET), which can be correctly executed in the *AbstractRadio* class, implementing the radio module in the *wlan* module. Figure 6.2 shows the implementation of the *sendRadioConfigMsg* function. When a NIC receives the command, it changes its channel information accordingly and notifies a *subControl* module. When the *subControl* module finishes executing a CA operation, it notifies the *mainControl* module about the completion of a CA operation.

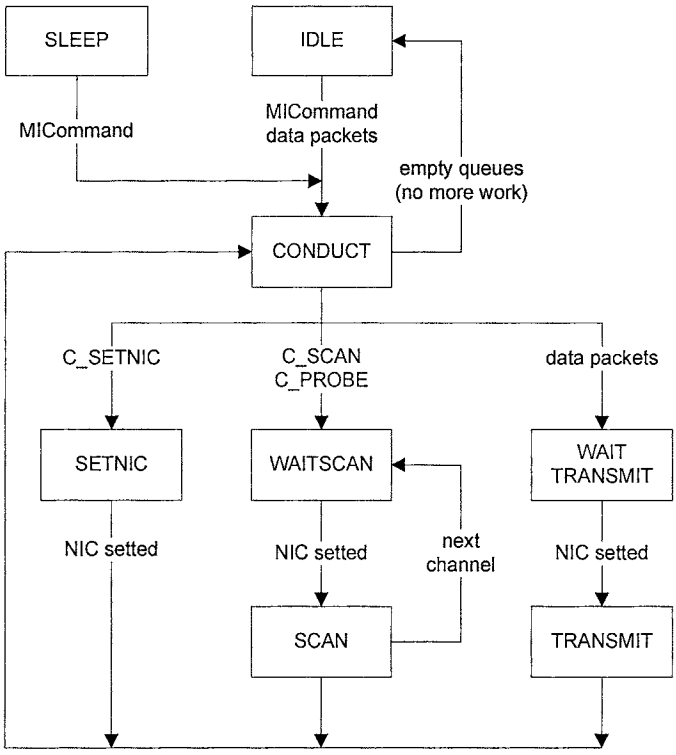


Figure 6 3 The State Diagram Of The *subControl* Module

6.2 State Machine

The *subControl* module performs CA operations according to a state machine. Figure 6 3 shows the state diagram of the *subControl* module In the SLEEP state, the *subControl* module is initialized In the IDLE state, the *subControl* module is idle In the CONDUCT state, other states are triggered to perform CA operations accordingly. The SETNIC state is to assign a channel. The WAITSCAN and SCAN states are to perform scanning and probing The WAITTRANSMIT and TRANSMIT states are to transmit packets on the correct channel

6.2.1 SLEEP and IDLE

The SLEEP state is the beginning state of the state diagram in Figure 6.3. In the SLEEP state, the *subControl* module is initialized and discards packets from other modules In simulation, even though the *mainControl* module is in its INIT state, NICs (the *wlan* modules) still receive packets from other nodes and forward packets to the *MIControl* module. If the *mainControl* module starts up while a NIC is receiving a packet, the *mainControl* module should discard the packet because the packet was received before the actual start time of a node in simulation However, the *mainControl* module still accepts the packet and deals with it Such acceptance may bring inaccurate results for the simulation of CA protocols In order to prevent the problem, the *subControl* module discards packets ahead of the *mainControl* module However, in other than the SLEEP state, the *subControl* module forwards packets to the *mainControl* module immediately when packets are received from its corresponding NIC In the SLEEP state, the *subControl* module is triggered to the CONDUCT state by receiving a CA command from the *mainControl* module. Once the *mainControl* module starts, it commands the *subControl* module to perform a CA operation So, when the *subControl* module receives a CA command in the SLEEP state, it assumes that the *mainControl* module has been started, then starts its process as well

In the IDLE state, the *subControl* module is simply waiting for receiving CA commands and packets When the *subControl* control receives them from the *mainControl* module, the *subControl* module stores them in a queue properly and

```
void MISubControl· enqueueOperation(MICommand *cmd)
{
        if(cmd->getPriority())
                $highCommandQueue$.insert(cmd),
        else
                $lowCommandQueue$ insert(cmd);
}
```

Figure 6 4· The Set Of Queues The *subControl* Maintains

enters the CONDUCT state When the *subControl* module enters the IDLE state, it notifies the *mainControl* module of that, the *subControl* module is in the IDLE state. This is because the *mainControl* module will want to know when the *subControl* module becomes idle.

When the *subControl* module receives CA commands, regardless of a state, it stores CA commands into two different queues, *highCommandQueue* and *lowCommandQueue*, according to their priorities The *subControl* maintains *highCommandQueue* and *lowCommandQueue* to buffer high priority CA commands and low priority CA commands respectively Since high priority CA commands are supposed to be handled prior to low priority CA commands, the *subControl* module executes CA commands buffered in *highCommandQueue* first If there is no CA commands in *highCommandQueue*, then the *subControl* module proceeds CA commands from *lowCommandQueue*. The priority of a CA command is determined in the *mainControl* module according to a CA protocol. In the *subControl* module, the *enqueueOperation* function stores CA commands into a proper queue as shown in Figure 6.4

In addition, The *subControl* module maintains two more queues,

highDataQueue and *lowDataQueue*, to maintain packets. The *subControl* module receives two kinds of packets from the *mainControl* module channel management packets and data packets In a MIMC network, for nodes to maintain a CA protocol, channel management packets should take precedence over data packets In all states, except the SLEEP state, when the *subControl* module receives packets, it buffers channel management packets into *highDataQueue* and data packets into *lowDataQueue*. The *subControl* module transmits the packets in *highDataQueue* before the packets in *lowDataQueue* Hence, whenever a channel management packet arrives, the *subControl* module sends it out first, even if the packet arrives later than data packets. In the *subControl* module, the *enqueueData* function stores packets into a proper queue as shown in Figure 6 5.

```
void MISubControl. enqueueData(cMessage *msg)
{
        if(dynamic_cast<MIPacket *>(msg))
                $highDataQueue$ insert(msg);
        else
                $lowDataQueue$ insert(msg),
}
```

Figure 6 5 The Set Of Queues The *subControl* Maintains

6.2.2 CONDUCT

In the CONDUCT state, the *subControl* module checks the aforementioned queues and leads to a proper state to perform CA operations. The *subControl* module handles CA commands prior to packets, because such CA operations as assigning, scanning, and probing a channel have higher priority than packet transmission to maintain CA protocols. So, in the CONDUCT state, the *subControl* module checks

```
class MISubControl{
protected
    ChannelInfo *newChannelInfo,
    ScanControlInfo *scanCtrlInfo,
        .
        .
}
```

Figure 6 6. Member Variables To Maintain Control Information In The *subControl* Module

the command queues (*highCommandQueue* and *lowCommandQueue*) first and the data queues (*highDataQueue* and *lowDataQueue*) later The *subControl* module retrieves a CA command from the command queues, if any If the command is the C_SETNIC command, assigning a channel to a NIC, the *subControl* module enters the SETNIC state If the command is either a C_SCAN command or a C_PROBE command (Section 5.2) to scan or probe channels, the WAITSCAN state is triggered. If no commands are in the command queues, the *subControl* module retrieves a packet from the data queues, if any. Then, the *subControl* module triggers the WAITTRANSMIT state to transmit the packet

When the *subControl* module leads CA commands to a proper state in the CONDUCT state, it retrieves control information from CA commands to perform CA operations properly CA commands carry control information to control CA operations (Section 5 2) The *subControl* module maintains two variables, $newChannelInfo$ and $scanCtrlInfo$, to keep the control information They are defined as shown in Figure 6 6 The $newChannelInfo$ variable keeps the control information of the C_SETNIC command. The $scanControlInfo$ variable keeps the control information of both the C_SCAN and C_PROBE commands Such variables

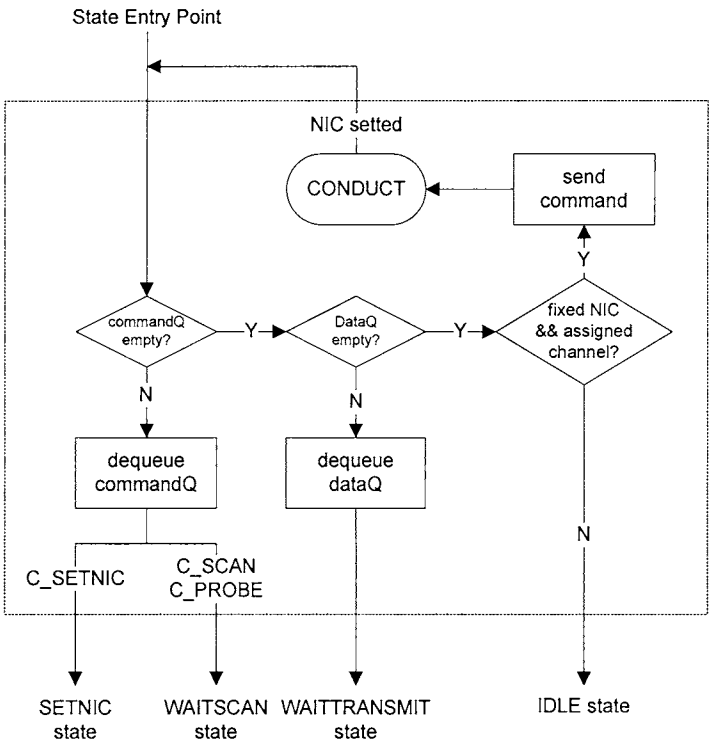


Figure 6.7: The Inside Structure Of The CONDUCT State In The *subControl* Module

are used in the SETNIC, WAITSCAN, and SCAN states to perform CA operations properly.

In addition, the *subControl* module maintains the channel assigned to its corresponding NIC in the CONDUCT state. If all the queues are empty, which means the *subControl* module does not have any work to do, then the *subControl* module tries to preserve an assigned channel of its corresponding NIC before entering into the IDLE state. After CA operations are performed, a channel of the NIC might have been changed to a different channel from its assigned channel. The *subControl* module retrieves a channel assigned to its corresponding NIC from the nicTable module and checks whether the NIC's current channel is equal to the assigned channel. If they are same, the *subControl* module enters into the IDLE

state. Otherwise, the *subControl* module commands the NIC to change to the assigned channel and waits until the channel is actually assigned to the NIC When the *subControl* module is notified of that, the channel is correctly assigned to the NIC, it enters the CONDUCT state once again The re-entrance is because the *subControl* module could receive CA commands and packets from the *mainControl* module while waiting for the assigning a channel operation. However, since a channel is assigned only to a fixed NIC, the *subControl* module preserves an assigned channel only if its corresponding NIC is a fixed NIC Figure 6.7 shows the inside structure of the CONDUCT state, which is depicted according to Section 8 3

6.3 SETNIC

In the SETNIC state, the CA operation to assign a channel to a NIC is performed This state is triggered by the C_SETNIC command. Once the *subControl* module enters the SETNIC state, it checks whether the channel, kept in $newChannelInfo$, is equivalent to the current channel of the NIC. If so, the *subControl* module notifies the *mainControl* module of that, a channel is assigned to a NIC because the same channel has been already used in a NIC. Then, it goes back to the CONDUCT state Otherwise, the *subControl* module sends a command to change a channel to a NIC and waits until a NIC actually assigns a channel When the *subControl* module is notified of that a NIC correctly assigns a channel, it updates the new channel information into the $nicTable$ module and notifies the *mainControl* module of that, it has assigned a channel. Then, the *subControl* module goes back to the CONDUCT state

6.4 WAITSCAN and SCAN

In the WAITSCAN and SCAN states, the two CA operations, scanning and probing, are performed. Scanning a channel is to tune a NIC to a particular channel and listen to the channel. Probing a channel is also to tune a NIC to a particular channel and listen to the channel. During probing, packets are sent to probe a channel before listening. However, since the two CA operations have the same procedure, tuning and listening, both CA operations are performed in the same states, WAITSCAN and SCAN.

In the WAITSCAN and SCAN states, the *subControl* module controls the CA operations by $scanControlInfo$ which contains a series of channels, data rate of channels, duration for listening to each channel, and indication to activate promiscuous mode. In addition, $scanControlInfo$ can contain additional information, such as a packet and iteration number of sending the packet for probing. With $scanControlInfo$, the *subControl* module allows for scanning or probing a series of channels at once.

In the WAITSCAN state, the *subControl* module tunes its corresponding NIC to a particular channel. When the *subControl* module enters in this state, it compares the current channel of the NIC with the first channel out of channels that $scanControlInfo$ contains. If they are same, the *subControl* module directly enters the SCAN state. Otherwise, the *subControl* module sends a command to tune a NIC to the channel. Then, it enters the SCAN state when the *subControl* module is notified that a NIC correctly tunes the channel.

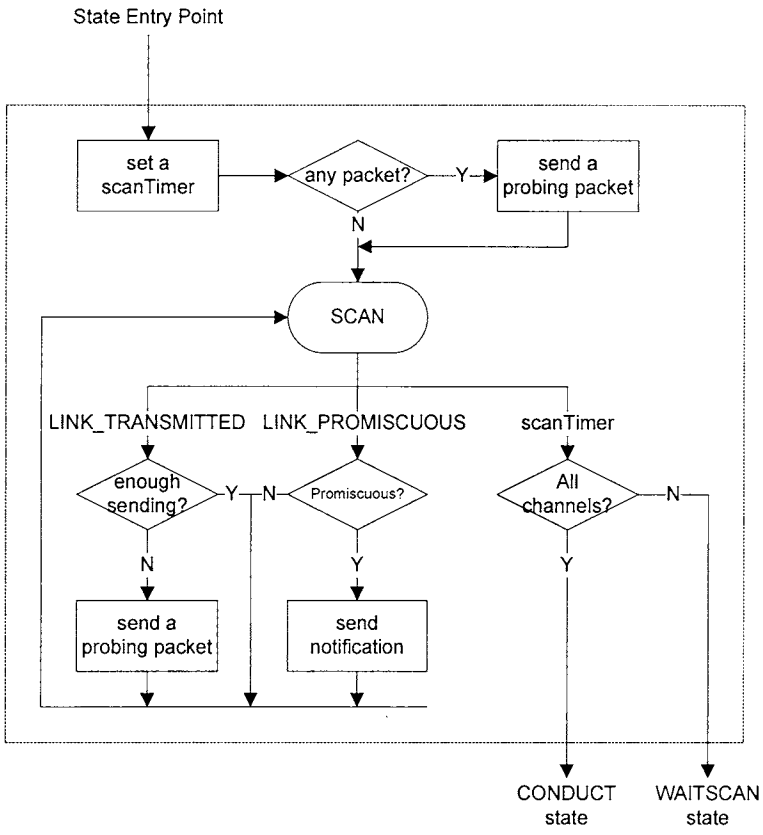


Figure 6.8: The Inside Structure Of The SCAN State In The *subControl* Module

In the SCAN state, the *subControl* module actually performs the CA operations, scanning and probing. Figure 6.8 shows the inside structure of the SCAN state, which is depicted according to Section 8.3. When the *subControl* module enters in the SCAN state, first, it sets a timer to be expired in duration of listening. For the duration of listening, scanning and probing are performed. For scanning, the *subControl* module simply waits to receive packets. For probing, the *subControl* module sends the packet, which $scanControlInfo$ contains as many times as the number that $scanControlInfo$ indicates. After sending enough packets, the *subControl* module waits to receive packets. If the *subControl* module receives packets, it forwards the packets to the *mainControl* module. When the

timer is expired, the *subControl* module checks whether the CA operation is performed on all the series of channels that $scanControlInfo$ indicates. If not, the *subControl* module goes back to the WAITSCAN state to perform the same CA operation for the next channel. Otherwise, it terminates a CA operation and enters into the CONDUCT state.

6.5 WAITTRANSMIT, TRANSMIT

In the WAITTRANSMIT and TRANSMIT states, the *subControl* module transmits packets on the correct channel In WAITTRANSMIT state, the *subControl* module tunes its corresponding NIC to a particular channel for a packet transmission To do so, first, the *subControl* module retrieves a proper channel information on which the packet needs to be transmitted To retrieve a proper channel information, the *subControl* module implements the $getChannelInfo$ function as shown in Figure 6 9 In the $getChannelInfo$ function, the *subControl* module gets a channel information for a packet transmission in the following order First, the *subControl* module gets the channel information from the control information of a packet, because the *mainControl* module can designate a specific channel for a packet Second, the *subControl* module gets the channel information from the *nicTable* module for broadcast packets when its NIC is a fixed NIC. When a channel is not specified for a broadcast packet, the *subControl* module assumes that a packet is broadcasted on its NIC's assigned channel. Finally, the *subControl* module gets the channel information from the *neighborTable* module for unicast packets. The *neighborTable* module provides mapping between destination MAC

```
ChannelInfo* MISubControl::getChannelInfo(cMessage *msg)
{
  MIPacketCtrl *miCtrl =
    check_and_cast<MIPacketCtrl *>(msg->getControlInfo());
  MACAddress macAddr =
    getDestMAC(miCtrl->getControlInfo());
  ChannelInfo *chInfo = new ChannelInfo();

  if(miCtrl->getChannelInfo())
    *chInfo = *miCtrl->getChannelInfo();
  else if(macAddr.isBroadcast() && nicEntry->isFixed())
    *chInfo = nicEntry->getChannelInfo();
  else if(!macAddr.isBroadcast())
    *chInfo = nbT->getChannelInfoByMAC(macAddr);

  return chInfo;
}
```

Figure 6.9: Implementation Of The *getChannel* Function

address and proper channel information. (In most of cases, a proper channel can be found in the *getChannelInfo* function. However, if any proper channel could not be found, a packet will be transmitted in a NIC's current channel.)

Once the *subControl* module gets the channel information for a packet, it compares the current channel of the NIC with the channel. If they are same, the *subControl* module directly enters the TRANSMIT state. Otherwise, the *subControl* module sends a command to tune a NIC to the channel. Then, it enters into the TRANSMIT state when the *subControl* module is notified of that, a NIC correctly tunes the channel.

In the TRANSMIT state, the *subControl* module sends a packet to its NIC for transmission. If the NIC successfully transmits a packet, it notifies the *subControl* module of the success. Then, the *subControl* module enters into the CONDUCT state. However, if the NIC notifies the *subControl* module of failure of a packet

transmission, the *subControl* module notifies the *neighborTable* module of the failure depending on the importance of the packet. If the packet is non failure-free, the *subControl* module notifies the *neighborTable* module about the failure. Otherwise, the *subControl* module does not. Whether a packet transmission succeeds or not, the *subControl* module notifies the *mainControl* module about the completion of a packet transmission before entering the CONDUCT state.

When the CA operation, scanning or probing, is commanded in the promiscuous mode, the *subControl* module notifies the *mainControl* module with received packets. In INET, when the *wlan* module receives packets which are not destined to the *wlan* module, it always sends a notification with the received packets regardless of the promiscuous mode. When the *subControl* module receives the notification in the promiscuous mode, the *subControl* module notifies a notification to the *mainControl* module with the packets passed with the notification so that the *mainControl* module can handle the packets.

CHAPTER 7

SUPPORTIVE MODULES

This chapter presents the design of the *neighborTable* module and the *nicTable* module in detail. Also, this chapter discusses the necessary modification in some modules in INET to add MIMC support.

7.1 neighborTable

In MIMC networks, CA protocols ask nodes to collect information of neighbor nodes. The *neighborTable* module is designed to maintain information of neighbor

```
class INET_API MINbEntryBase
{
protected:
  IPAddress ipAddr;
  std::vector<NicInfo *> nicCache;
  int hopDistance;
  int mainNicIndex;
        :
        :
};

class INET_API MINbTableBase
{
protected:
  typedef std::vector<MINbEntryBase *> NbVector;
  NbVector nbCache;
        :
        :
}
```

Figure 7.1: Declaration Of The *MINbEntryBase* And *MINbTableBase* Classes

```
class NicInfo{
public:
  MACAddress mac;
  ChannelInfo channelInfo;
      :
      :
}
```

Figure 7.2: Declaration Of The $NicInfo$ Class

nodes. The *neighborTable* module is built upon the $MINbTableBase$ class with the $MINbEntryBase$ class. Figure 7.1 shows the declaration of the $MINbTableBase$ class and the $MINbEntryBase$ class. The $MINbTableBase$ class maintains information of neighbor nodes in a vector which is composed of instances of the $MINbEntryBase$ class. The $MINbEntryBase$ class is used to store actual information of a neighbor node. Basically, the $MINbEntryBase$ class is implemented to store a neighbor node's IP address, MAC addresses, assigned channels, and hop distance. A MAC address and an assigned channel are maintained together in the $NicInfo$ class defined as shown in Figure 7.2. Since the MIMC-SIM framework assumes that each node is identified by the unique IP address, the unique IP address is considered as a primary key to distinguish among neighbor nodes.

In addition, the *neighborTable* module is designed to include the following major features. First, the *neighborTable* module provides a set of functions which retrieve neighbor information accordingly. Second, the *neighborTable* module maintains neighbor nodes properly by collaborating with other modules. Finally, the *neighborTable* module maps among IP addresses, MAC addresses, and channel information.

7.1.1 Access Retrieval Functions

The *neighborTable* module provides a set of functions to update and retrieve information of neighbor nodes. Figure 7.3 shows all the functions that the

```
class INET_API MINbTableBase
{
public:
  virtual int getNumNeighbors(int hop=0);

  virtual MINbEntryBase* operator[];
  virtual MINbEntryBase* getEntryByIndex(int index);
  virtual MINbEntryBase* getEntryByIP(const IPAddress& ip);
  virtual MINbEntryBase* getEntryByMAC(const MACAddress& mac);

  virtual const IPAddress& getIPByIndex(int index);
  virtual const MACAddress& getMACByIndex(int index);
  virtual const ChannelInfo& getChannelInfoByIndex(int index);
  virtual int getHopDistanceByIndex(int index);

  virtual const IPAddress& getIPByMAC(const MACAddress& mac);
  virtual const ChannelInfo& getChannelInfoByMAC
                                    (const MACAddress& mac);
  virtual int getHopDistanceByMAC(const MACAddress& mac);

  virtual const MACAddress& getMACByIP(const IPAddress& ip);
  virtual const ChannelInfo& getChannelInfoByIP
                                      (const IPAddress& ip);
  virtual int getHopDistanceByIP(const IPAddress& ip);

  virtual MINbEntryBase* getGatewayEntry();
  virtual const IPAddress& getIPOfGatewayEntry();
  virtual const MACAddress& getMACOfGatewayEntry();
  virtual const ChannelInfo& getChannelInfoOfGatewayEntry();

  virtual void setGatewayEntry(const IPAddress& ip);
  virtual MINbEntryBase* updateEntry(MINbEntryBase *entry);
  virtual void addEntry(MINbEntryBase *entry);
  virtual void removeEntry(MINbEntryBase *entry);
                :
                :
};
```

Figure 7.3: The Set Of Functions The *neighborTable* Module Provides To Update And Retrieve Information Of Neighbor Nodes

neighborTable module provides. For other modules to use the set of functions, the *neighborTable* module is directly accessed by calling an access function. The *neighborTable* module does not exchange messages to interact with other modules. Instead, other modules call an access function to access the *neighborTable* module. The MIMC-SIM framework defines the *MINbTableAccess* class, which is built upon the *ModuleAccess* class. The *MINbTableAccess* class provides the access function which returns a pointer of the *neighborTable* module. Once other modules get the pointer of the *neighborTable* module, they can use the set of functions that the *neighborTable* module provides. Figure 7.4 shows the code of how the *neighborTable* module is accessed by calling the access function.

```
MINbTableBase *nbT;
nbT = MINbTableAccess().get();
```

Figure 7.4: Access Function Of The *neighborTable* Module

7.1.2 Maintenance of neighbor nodes

The *neighborTable* module collaborates with the *subControl* and *mainControl* modules to maintain neighbor nodes accordingly and affects the *routingTable* module and the *mainControl* module when a neighbor node is removed. Basically, the *neighborTable* updates information of neighbor nodes by the *mainControl* module. When the *mainControl* module receives channel management packets, the *mainControl* module updates the information that the packets contain into the *neighborTable* module. The *neighborTable* module removes information of neighbor nodes in two different cases. One case is when a *subControl* module fails a packet

transmission to a specific neighbor node The other case is that information of a neighbor node has not been updated for long period.

The *neighborTable* module gets rid of a neighbor node to which the *subControl* module fails to transmit a packet. When the *subControl* module fails transmission for a non failure-free packets, it notifies the *neighborTable* module about the failure When the *neighborTable* module is notified about the failure, it removes the neighbor node which was the next hop node of the failure transmission. In the MIMC-SIM framework, the failure of non free-failure packet transmission means that the link with the next hop node is broken, and the next hop node is not a neighbor node any longer. Hence, when the *neighborTable* module is notified about the failure, it evaluates a MAC address of a next hop node and removes a neighbor node which associates with the MAC address

In addition, the *neighborTable* module does not keep information of a neighbor node permanently. Instead, the *neighborTable* module removes a neighbor node if it has not been updated for a certain time In a MIMC network, nodes exchange channel management packets with each other frequently Based on the channel management packets, nodes find and update their neighbor nodes However, if a node has not received channel management packets from a neighbor node for a certain time–the default value is 180 seconds in the MIMC-SIM framework–since the last update of the neighbor node, the node regards that the neighbor node disappeared, and information of the neighbor node is invalid. The *neighborTable* module sets a timer for each neighbor node when information is entered or updated When a timer is expired, the associating neighbor node is removed from the

neighborTable module.

When the *neighborTable* module removes a neighbor node, it updates the *routingTable* module accordingly. The *routingTable* module maintains a routing table in which destination IP addresses are corresponding to next hop IP addresses. Since a next hop node is considered as one of the neighbor nodes in a MIMC network, next hop IP addresses are also maintained in the *neighborTable* module, and a node which is not maintained in the *neighborTable* module cannot be a next hop node in a routing table. In other words, next hop node IP addresses in a routing table must be maintained in the *neighborTable* module as neighbor nodes' IP addresses. Where X is a set of IP addresses of neighbor nodes in the *neighborTable* node, and Y is a set of next hop IP addresses in the *routingTable* module, the relation between X and Y is formalized as $X \supseteq Y$. Hence, when the *neighborTable* node removes a neighbor node, it also removes the route whose next hop IP address is the same as the neighbor node's IP address from the *routingTable* module.

In addition, when the *neighborTable* module removes a neighbor node, it notifies the *mainControl* module about the removal, so that the *mainControl* module can cope with removals of neighbor nodes, especially a default route node. When the *mainControl* module decides new routing, it also indicates which neighbor node is used for a default route in the *neighborTable* module. When the *neighborTable* module removes a neighbor node, it notifies the removal differently depending on the neighbor node. If the default route node is removed, the *neighborTable* module notifies the *mainControl* module of the removal in a specific category, NF_NBTABLE_GWENTRY_DISMISSED. Otherwise, when other neighbor nodes

are removed, the *neighborTable* module notifies the *mainControl* module in the category, NF_NBTABLE_ENTRY_DISMISSED

7.1.3 Mapping information of neighbor nodes

The *neighborTable* module maintains information of neighbor nodes by mapping between IP addresses and MAC addresses and between MAC addresses and assigned channels. So, such information as a MAC address and an assigned channel can be retrieved based on an IP address and a MAC address respectively. Since the *neighborTable* module associates an IP address with multiple MAC addresses, it maintains a main MAC address which is primarily used to communicate with a specific neighbor node. The *MINbEntryBase* class defines a member variable, *mainNicIndex*, which indicates the index of a main NIC. The main MAC address of each neighbor node is determined according to a CA protocol.

7.2 nicTable

The *nicTable* module is designed to maintain such information of NICs as roles, MAC address, and assigned channel. The *nicTable* module is built upon the *MINicTableBase* class with the *MINicEntryBase* class. Figure 7.5 shows the implementation of the *MINicTableBase* and *MINicEntryBase* classes. The *MINicTableBase* class maintains information of NICs in a vector, *NicTableCache*, which is composed of instances of the *MINicEntryBase* class. The *MINicEntryBase* class is used to store actual information of a NIC. When the *nicTable* module is initialized, the *MINicTableBase* class retrieves a MAC address

```
class INET_API MINicEntryBase{
protected:
  int category;
  int type;
  NicInfo nicInfo;
        :

};

class INET_API MINicTableBase{
public:
  typedef std::vector<MINicEntryBase *> MINicVector;

protected:
  MINicVector NicTableCache;
          .
          .
};
```

Figure 7 5: Implementation Of The *MINicEntryBase* And *MINicTableBase* Classes

of each NIC from the *interfaceTable* module and generates an instance of the *MINicEntryBase* class based on the MAC addresses

In addition, when the *nicTable* module is initialized, the *classifyNics* function is executed to allot roles of NICs. The *classifyNics* function is defined in the *MINicEntryBase* class The *classifyNics* function allots roles of NICs in two different member variables of the *MINicEntryBase* class, *category* and *type* The *category* variable is to categorize NICs into a fixed NIC or a switchable NIC The *type* variable is to give a specific role that a CA protocol defines to a NIC. When the *nicTable* module is extended for a specific CA protocol, the *classifyNics* function shall be re-implemented to allot the *category* and the *type* of NICs according to a CA protocol. Figure 7 6 shows an example of implementation of the *classifyNics* function

```
void classifyNics()
{
  NicTableCache[0]->setCategory(FIXED);
  NicTableCache[0]->setType(RECV);

  NicTableCache[1]->setCategory(SHIFT);
  NicTableCache[1]->setType(SEND);
}
```

Figure 7.6: An Example Of The *classifyNics* Function

The *nicTable* module dose not exchange messages with other modules. Instead, the *nicTable* module is accessed by calling an access function. The MIMC-SIM framework defines the *MINicTableAccess* class, which is built upon the *ModuleAccess* class. The *MINicTableAccess* class provides the access function which returns a pointer of the *nicTable* module. Figure 7.7 shows the code of how the *nicTable* module is accessed by calling the access function. Through the access, the set of functions that the *nicTable* module provides to retrieve information of NICs (Figure 7.8) can be used.

```
MINicTableBase *nicT;
nicT = MINicTableAccess().get();
```

Figure 7.7: Access Function Of The *nicTable* Module

```
class INET_API MINicTableBase{
public:
  virtual int getNumOfNics();

  virtual MINicEntryBase* getEntryByIndex(int index);
  virtual MINicEntryBase* operator[] (int index);
  virtual MINicEntryBase* getEntryByMAC(MACAddress& mac);
  virtual MINicEntryBase* getEntryByCategory(int category);
  virtual MINicEntryBase* getEntryByType(int type);

  virtual InterfaceEntry* getInterfaceEntryByIndex(int index);
  virtual InterfaceEntry* getInterfaceEntryByMAC
                                              (MACAddress& mac);
  virtual InterfaceEntry* getInterfaceEntryByCategory
                                                (int category);
  virtual InterfaceEntry* getInterfaceEntryByType(int type);

  virtual const NicInfo& getNicInfoByIndex(int index);
  virtual const NicInfo& getNicInfoByMAC(MACAddress& mac);
  virtual const NicInfo& getNicInfoByCategory(int category);
  virtual const NicInfo& getNicInfoByType(int type);

  virtual const MACAddress& getMACByIndex(int index);
  virtual const MACAddress& getMACByCategory(int category);
  virtual const MACAddress& getMACByType(int type);

  virtual const ChannelInfo& getChannelInfoByIndex(int index);
  virtual const ChannelInfo& getChannelInfoByMAC
                                              (MACAddress& mac);
  virtual const ChannelInfo& getChannelInfoByCategory
                                                (int category);
  virtual const ChannelInfo& getChannelInfoByType(int type);
            :
            :
};
```

Figure 7.8: The Set Of Functions The *nicTable* Module Provides To Update And Retrieve Information Of NICs

7.3 Modification in INET

The MIMC-SIM framework modifies some modules in INET to support the MIMC network simulation.

7.3.1 Network

Although the *network* module represents the network layer and is not part of the *MIControl* module, in order to let the *MIControl* module handle the ARP protocol mechanism for CA protocols, the *network* module needs to be modified slightly. Since CA protocols replace the ARP protocol with mapping between IP addresses and MAC addresses, the MIMC-SIM framework does not need to have the ARP process in the network layer. Thus, in the MIMC-SIM framework, the *arp* module is removed from the *network* module as shown in Figure 7.9. Compared to the original network module in Figure 3.3(a) where the *ip* module sends outgoing packets to the *arp* module, the *ip* module in the new *network* module communicates with the *MIControl* module directly.

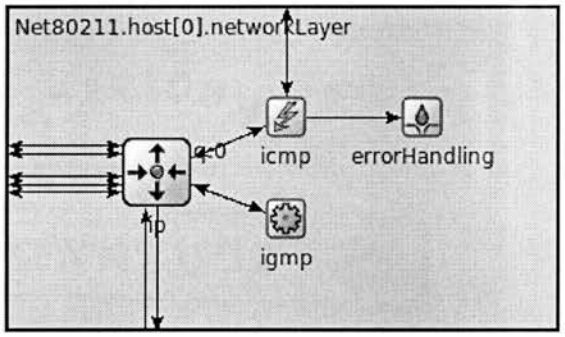


Figure 7.9: The Structure Of The *network* Module In The MIMC-SIM Framework

7.3.2 ChannelControl

The *channelControl* module maintains neighbor nodes in a network. It gets informed about the location of nodes and determines which nodes are within communication range This information is used by the *radio* modules at transmissions. The *channelControl* module is built upon the *ChannelControl* class The original class assigns and maintains one channel per node. However, in MIMC networks, multiple channels are assigned to multiple radios (included in NICs) in a single node. The class is modified to give each node a list of radios, so that multiple channels and radios can be considered for each node.

7.3.3 Radio

The *radio* module is built upon the *AbstractRadio* class which implements wireless communication at the physical layer When the *radio* module changes its channel to a new one while it is receiving, it should clear its past states so that new receiving and transmitting procedures can be started However, the clearing past states was not correctly implemented in the *AbstractRadio* class. In addition, the *radio* module shall change the data rate of a channel as well. Even though the *AbstractRadio* class implements a function to change the data rate of a channel, it did not execute the function when the *radio* module is asked to change the data rate of a channel The *AbstractRadio* class is modified to change a channel and its data rate accordingly.

The *ChannelAccess* class is a parent class of the *AbstractRadio* class It is

designed to support packet transmission among neighbor nodes The class retrieves neighbor nodes from the *channelControl* module and delivers a packet to the nodes Since the *ChannelControl* class is modified to allow nodes to assign multiple channels, the *ChannelAccess* class is also modified to coordinate with the *channelControl* module The original *ChannelAccess* class delivers a packet based on a node. The modified *ChannelAccess* class delivers a packet based on a NIC, so that a packet can be delivered to NICs on the same channel of a sending NIC.

7.3.4 Mgmt

The *mgmt* module is built upon the $Ieee80211Mgmt$ class The original class does not allow for receiving a command However, in MIMC networks, the *subControl* module sends a command to the *mgmt* module to change a channel of a NIC. The class is modified to receive a command and forward it to a lower module, the *mac* module

7.3.5 Mac

The *mac* module is built upon the $Ieee80211Mac$ class which implements the IEEE802.11 MAC protocol. In the original class, it is possible that the *radio* module is asked to change its channel while the *mac* module is still in waiting states, such as the DEFER and BACKOFF states of the IEEE802 11 MAC protocol. Apparently, the radio shall only change its channel when no packet is waiting for transmission in MAC, so that a waiting packet can be transmitted in its expected channel The class is modified accordingly

CHAPTER 8

STATE MACHINE

In this chapter, a set of predefined macros, FSME (Enhanced Finite State Machine), which is used to implement a state machine in the MIMC-SIM framework is introduced Also, the actual implementation of a state machine using FSME is shown in this chapter.

8.1 Enhanced Finite State Machine

In the MIMC-SIM framework, FSME provides a generic and flexible code structure to implement a state machine of a CA protocol A process of a protocol can be represented in a state machine logically A state machine is composed of a number of states, transitions among states, and actions in states Such components of a state machine can be described in a state diagram To implement a state machine in a constant code structure, the MIMC-SIM framework provides an implemental framework with a set of predefined macros, FSME. FSME is inspired by FSMA (Advanced Finite State Machine), provided in the INET distributed package [3] Both FSMA and FSME provide a set of macros that manipulate the variable built upon the *cFSM* class, which maintains a state of a state machine. Using the macros, a state machine can be implemented in a constant code structure. However, FSMA does not provide a complete set of macros to express a complete set of logic

required by CA protocols For example, when a state is just entered, FSMA cannot implement the actions which is executed after a state transition fails. Also, FSMA does not allow each state of a state machine to be implemented separately. In other words, implementation of a whole state machine has to be in one function. This causes inefficient works for extension For example, if a child class wants to extend only a specific state of a state machine from its parent class, FSMA cannot support that but forces a child class to extend a whole state machine at once. In contrast to FSMA, FSME provides a complete set of macros that can express a complete set of logic for CA protocols. In addition, FSME allows each state of a state machine to be implemented in a separate function for flexible extension. In the MIMC-SIM framework, the state machines of the *mainControl* module and the *subControl* module are implemented using FSME

Implementation of a state machine using FSME is separated into two parts: state definition and state embodiment State definition and state embodiment each can be represented in a function. A state definition function handles a state transition A state embodiment function implements actual actions of a specific state. For implementation of a state machine, only one state definition function exists with the

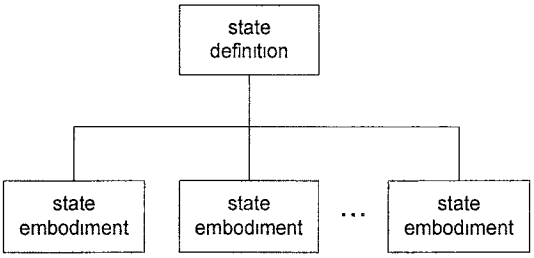


Figure 8.1· Relation Of Definition Function And State Embodiment

same number of state embodiment functions as the number of states in a state machine. A state definition function calls a proper state embodiment function according to a current state. Figure 8.1 shows the relation of a state definition function and a state embodiment function. In this chapter, to simplify, a state definition function and a state embodiment function are referred to a state definition and a state embodiment respectively. In rest of this chapter, implementation of a state definition and a state embodiment using FSME is presented in detail.

8.2 State Definition

In a state definition, each state of a state machine is associated with a specific state embodiment, and the state embodiments are called according to a current state. To implement a state definition, two FSME macros are used: FSME_Switch and FSME_State. A FSME_Switch manages a current state and iterates execution of FSME_States in the FSME_Switch for a state transition. A FSME_State is embedded in a FSME_Switch and executes a state embodiment according to a

```
void MIMainControlBase::handleWithFSM(cMessage *msg)
{
  FSME_Switch(fsm)
  {
    FSME_State(INIT, handleStateInit, msg);
    FSME_State(SCAN, handleStateScan, msg);
    FSME_State(ASSIGN, handleStateAssign, msg);
    FSME_State(SETNIC, handleStateSetnic, msg);
    FSME_State(NORM, handleStateNorm, msg);
  }
}
```

Figure 8.2: Implementation Of The *handleWithFSM* Function, A State Definition In The *mainControl* Module

current state with an event. Figure 8.2 shows the implementation of a state definition in the *mainControl* module using FSME_Switch and FSME_State.

```
#define FSME_Switch(fsm)                                          \
    bool ___event = true;                                         \
    bool ___transition = false;                                   \
    bool ___counter = 0;                                          \
    cFSM *___fsm = &fsm;                                          \
    EV << "processing event in state machine"                     \
       << ___fsm->getName() << endl;                              \
    while(___counter++ < FSM_MAXT ||                              \
    (opp_error(eINFLOOP, ___fsm->getStateName()), 0))
```

Figure 8.3: Definition Of FSME_Switch

Figure 8.3 shows definition of a FSME_Switch. A FSME_Switch takes the variable built upon the *cFSM* class as a current state and defines several variables, such as ___event and ___transition, to manage a state transition. The ___event variable indicates whether an event has triggered a state transition. The ___transition variable indicates whether a state transition happens in a state. Such variables are used for a FSME_State. Then, a FSME_Switch iterates its inside where FSME_States are embedded.

Figure 8.4 shows definition of a FSME_State. A FSME_State assigns a state, a

```
#define FSME_State(s, s_func, s_msg)                              \
    if(___fsm->getState() == s){                                  \
        s_func(___fsm, ___event, ___transition, s_msg);           \
        if(___transition){                                        \
            ___event = false;                                     \
            ___transition = false;                                \
            continue;                                             \
        }else break;                                              \
    }
```

Figure 8.4: Definition Of FSME_State

state embodiment, and an event and associates them together. Events are such messages as commands, channel management packets, timers, and notifications, which potentially trigger a state transition. At each execution of a FSME_Switch, if the state assigned in a FSME_State is same as a current state, the FSME_State calls its assigned state embodiment with an event and the variables managed in a FSME_Switch. When a state transition happens after execution of a state embodiment, FSME_Switch iterates its inside to execute the next FSME_State in which the assigned state is the same as a new current state.

8.3 State Embodiment

8.3.1 State Embodiment

A state embodiment implements actual actions of a state and is called from a state definition. To interact with a state definition, a state embodiment must be declared in a certain form as shown in Figure 8.5, which is the declaration of the state embodiments in Figure 8.2. A state definition is declared with four parameters: ___fsm, ___event, ___transition, and msg. The ___fsm parameter maintains a current

```
virtual void handleStateInit(cFSM *___fsm, bool ___event,
                             bool &___transition, cMessage *msg);
virtual void handleStateScan(cFSM *___fsm, bool ___event,
                             bool &___transition, cMessage *msg);
virtual void handleStateAssign(cFSM *___fsm, bool ___event,
                             bool &___transition, cMessage *msg);
virtual void handleStateSetnic(cFSM *___fsm, bool ___event,
                             bool &___transition, cMessage *msg);
virtual void handleStateNorm(cFSM *___fsm, bool ___event,
                             bool &___transition, cMessage *msg);
```

Figure 8.5: Declaration Of State Embodiments In The *mainControl* Module

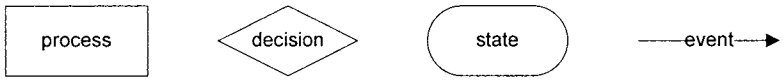


Figure 8.6: Components Used In A Flow Chart

state. The ___event parameter indicates whether an event has triggered a state transition. The ___transition parameter is used to indicate a state transition happens in a state embodiment. The msg parameter represents an event. A function declared with such four parameters can be utilized as a state embodiment.

To implement actual actions in a state embodiment, a process of a state should be represented in a flow chart. To draw a flow chart, four components are defined as

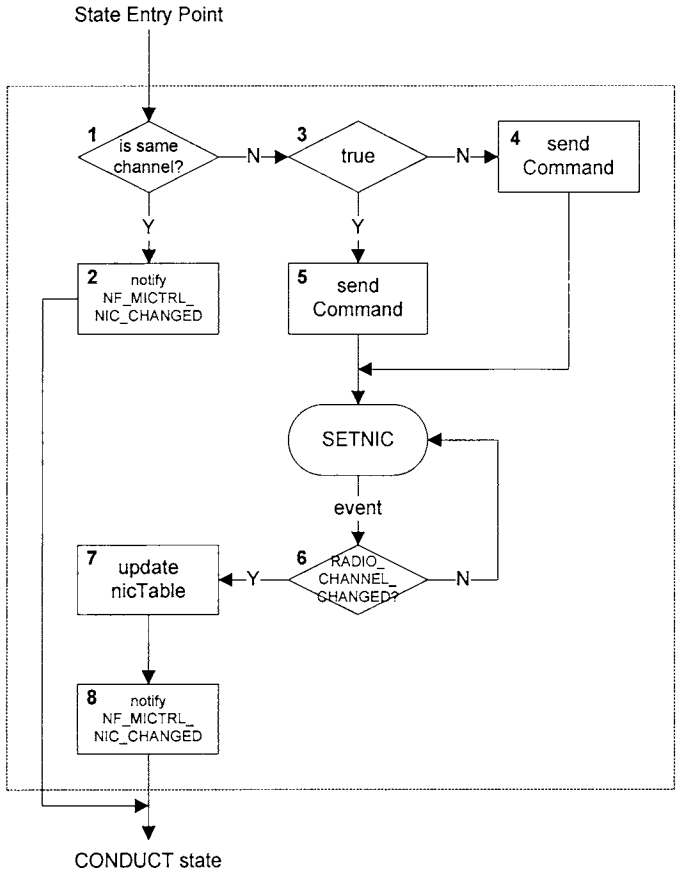
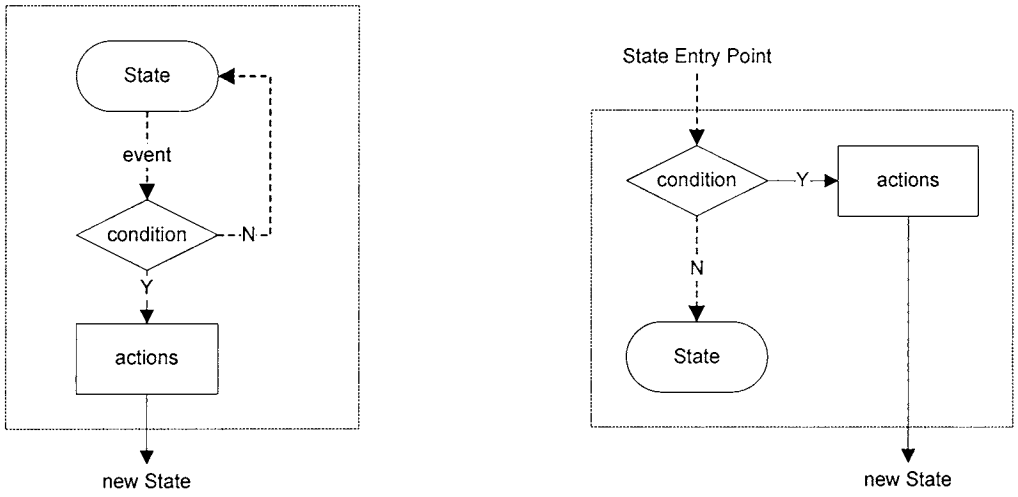


Figure 8.7: The Flow Chart Of The SETNIC State In The *subControl* Module

presented in Figure 8 6. The rectangle component represents a process used to implement actions. The diamond component represents a decision used to implement conditions The rounded rectangle component represents a state in which the flow of control stops and waits until the next event occurs The arrow component shows the flow of control in a flow chart When the arrow component is out from the state component, a specific event can be indicated on the arrow Since the state component represents a state, only one state component is allowed in a flow chart of a state With such components, for example, a flow chart of the SETNIC state in the *subControl* module can be depicted as Figure 8 7.

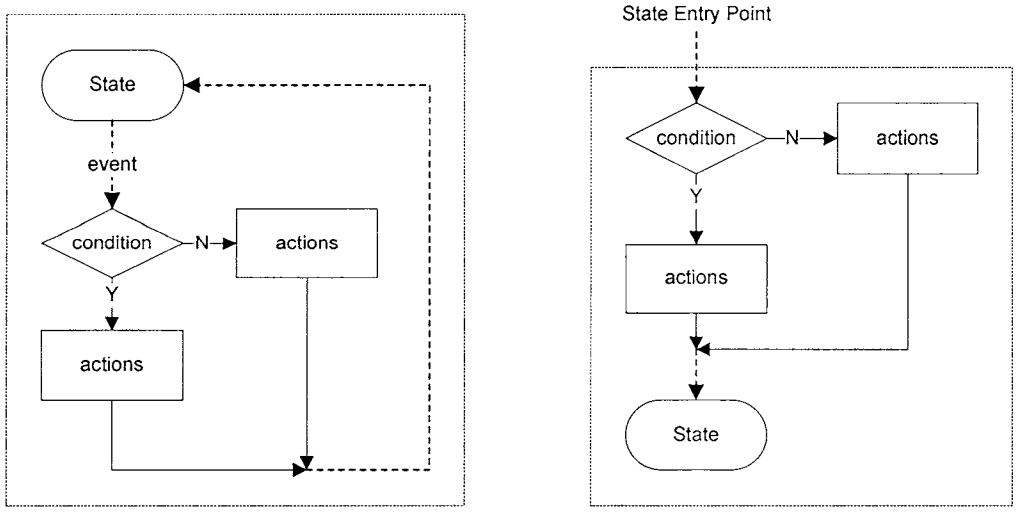
In order to implement a state embodiment as described in a flow chart, four FSME macros are used FSME_Event_Transition, FSME_No_Event_Transition, FSME_Event_Execute and FSME_No_Event_Execute. Figure 8 8 shows flow charts that each FSME macro can implement conceptually. Basically, all of the FSME macros assign a condition and actions and execute their assigned actions depending on their conditions. Among all, the Transition macros (FSME_Event_Transition and FSME_No_Event_Transition) participate in a state transition, while the Execute macros (FSME_Event_Execute and FSME_No_Event_Execute) do not In addition, the Event macros (FSME_Event_Transition and FSME_Event_Execute) are applied when a state machine stays in a state, and the No_Event macros (FSME_No_Event_Execute and FSME_No_Event_Execute) is applied when a state machine just enters in a state In other words, the Event macros are used to implement the components that occur after a state component, and the No_Event macros are used to implement the components that occur before a state component

For the FSME macros to be executed accordingly, they are defined to interact with the parameters of a state embodiment. Figure 8.9 shows the actual definition of the FSME macros. The Transition macros assign a condition, an action, and a state. They evaluate their conditions, and execute their assigned actions if the conditions are true. Then, they set the ___fsm parameter to their assigned state and



(a) FSME_Event_Transition

(b) FSME_No_Event_Transition



(c) FSME_Event_Execute

(d) FSME_No_Event_Execute

Figure 8.8: A Flow Chart Of The FSME Macros.
(The Dash Arrow Implies That There Might Be More FSME Macros.)

the ___transition parameter to true in order to indicate that a state transition is triggered. Afterward, the Transition macros terminate their state embodiment for a

```
#define FSME_Event_Transition                                          \
        (transition, condition, target, action)                        \
    if(___event && condition){                                         \
        FSME_Transition(transition, target, action);                   \
    }

#define FSME_No_Event_Transition                                       \
        (transition, condition, target, action)                        \
    if(!___event && condition){                                        \
        FSME_Transition(transition, target, action);                   \
    }

#define FSME_Print(exiting)                                            \
    (ev << "FSM " << ___fsm->getName()                                 \
        << ((exiting)?":leaving state " : ":entering state ")          \
        << ___fsm->getStateName() << endl)

#define FSME_Transition(transition, target, action)                    \
        FSME_Print(true);                                              \
        EV << "firing " << #transition << " transition for "           \
           << ___fsm->getName() << endl;                               \
        action;                                                        \
        ___fsm->setState(target, #target);                             \
        ___transition = true;                                          \
        FSME_Print(false);                                             \
        return;

#define FSME_Event_Execute(condition, actionT, actionF)                \
    if(___event && condition){                                         \
        actionT;                                                       \
    }else{                                                             \
        actionF;                                                       \
    }

#define FSME_No_Event_Execute(condition, actionT, actionF)             \
    if(!___event && condition){                                        \
        actionT;                                                       \
    }else{                                                             \
        actionF;                                                       \
    }
```

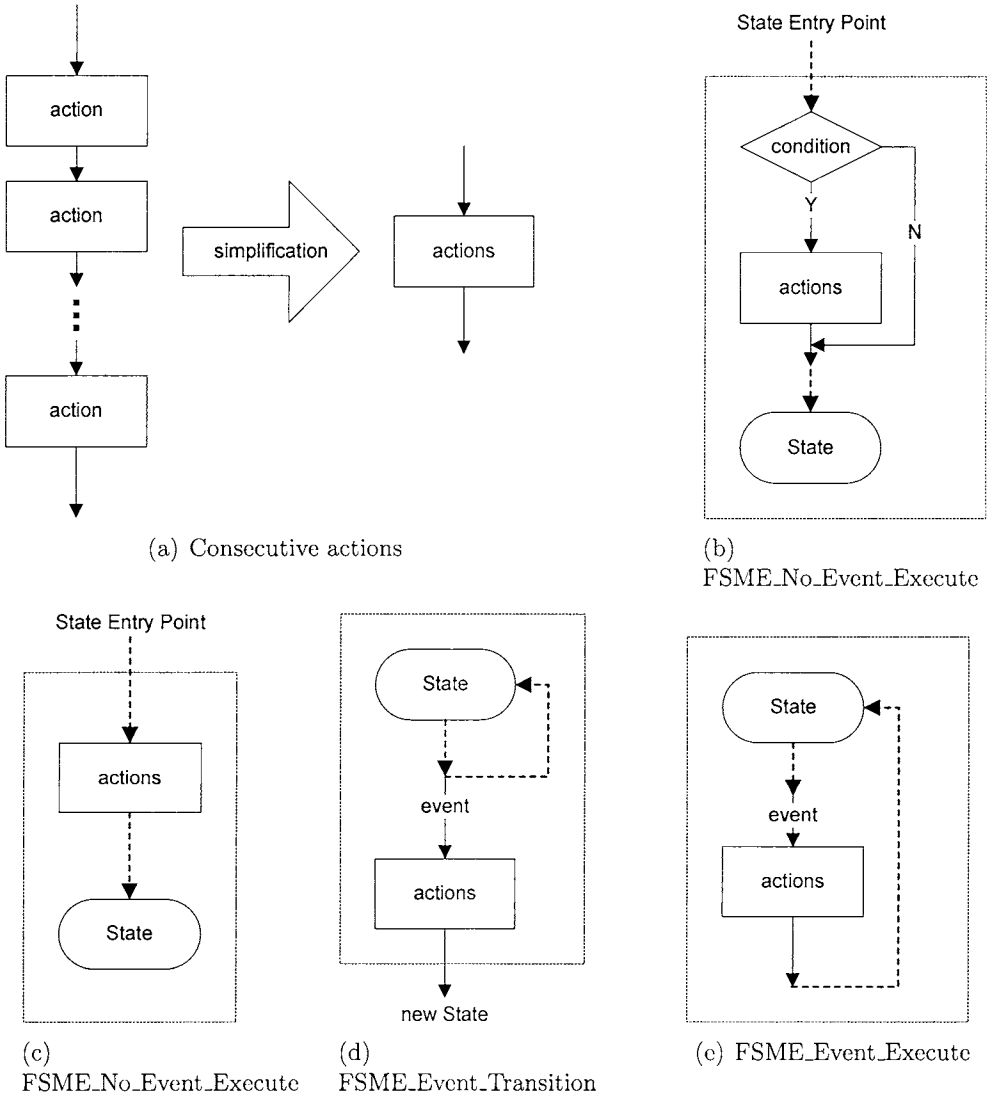
Figure 8.9: Definition Of The FSME Macros

state transition to be handled in a state definition. The Execute macros assign a condition and two different actions. one for true condition and the other for false condition. The Execute macros simply execute their assigned actions depending on their conditions. Furthermore, the Event macros are applied to be conducted, if __event is true. Otherwise, the No_Event macros are applied to be conducted. In addition, all the FSME macros can accept no actions So, for example, if a Transition macro assigns nothing for an action, it simply triggers a state transition without execution of any actions.

8.3.2 Example of State Embodiment

According to the definition of the FSME macros, the flow charts of each FSME macros can be simplified Figure 8.10 shows considerable simplifications of the flow chats depicted in Figure 8.8 (A) First of all, since consecutive actions in a flow chart can be assigned all together in a FSME macro, they can be represented in a process component as shown in Figure 8 10(a). (B) Second, since FSME macros can assign no action, a flow chart of a FSME macro can simply omit a process component for no action. For example, the flow chart of FSME_No_Event_Execute can be simplified as shown in Figure 8 10(b) when no action exists for the false condition. (C) Third, conditions of the Execute macros can be assigned to simply true or false (Since a state transition occurs in a certain condition in a state machine, FSME assumes that the Transition macros do not assign their condition to simply true or false. So, only the Execute macros are considered in this case.) If that is the case, the decision component and one of the process components of the

Execute macros can be omitted. For example, Figure 8.10(b) can be even more simplified as shown in Figure 8.10(c) when the condition is always true. (D) Finally, since the Event macros usually evaluate events for their conditions, a decision component can be replaced with an arrow indicating an specific event that an Event macro accepts for its condition. For example, FSME_Event_Transition and



(a) Consecutive actions

(b) FSME_No_Event_Execute

(c) FSME_No_Event_Execute

(d) FSME_Event_Transition

(e) FSME_Event_Execute

Figure 8.10: Simplification Of A Flow Chart
(The Dash Arrow Implies That There Might Be More FSME Functions.)

FSME_No_Event_Transition can be simplified as shown in Figure 8.10(d) and Figure 8.10(e) respectively.

According to such simplifications, Figure 8.7 can be simplified to Figure 8.11. Since the component 3 is always true and the component 4 has no actions, they can be simply omitted according to the simplification (B) and (C). The component 6 can be simplified to the arrow indicating the specific event according to the simplification (A). The components 7 and 8 can be simplified to one process component according to the simplification (D).

The simplified flow chart in Figure 8.11 can be applied to FSME macros. For example, the components A and B can be implemented by FSME_No_Event_Transition according to Figure 8.3.1. The component C can be implemented by FSME_No_Event_Execute according to Figure 8.10(c). The component D can be implemented by FSME_No_Event_Transition according to

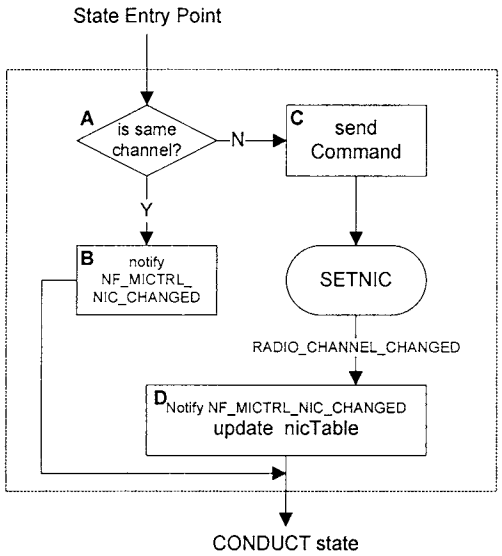


Figure 8.11: The Simplified Flow Chart Of The SETNIC State In The *subControl* Module

Figure 8.10(a) and Figure 8.10(d). Figure 8.12 shows the actual code structure implementing Figure 8.11 using the FSME macros. Since the FSME macros allow for using another FSME macro in their actions, FSME can implement a complete set of logic in a state embodiment.

```
void MISubControl::handleStateSetnic(cFSM *___fsm,
        bool ___event, bool &___transition, cMessage *msg)
{
  FSME_No_Event_Transition(,
    currentChannelInfo == *newChannelInfo,
    CONDUCT,
    delete newChannelInfo;
    nb->fireChangeNotification(NF_MICTRL_NIC_CHANGED, this);
  );

  FSME_No_Event_Execute(true,
    sendRadioConfigMsg(newChannelInfo);,
  );

  FSME_Event_Transition(,
    msg == fsmMsg && msg->getKind() == NIC_CHANGED,
    CONDUCT,
    updateNicInfo(*newChannelInfo);
    delete newChannelInfo;
    nb->fireChangeNotification(NF_MICTRL_NIC_CHANGED, this);
  );
}
```

Figure 8.12: Implementation Of Figure 8.11 Using FSME

CHAPTER 9

IMPLEMENTATION

In this chapter, the current implementation of two CA protocols in the MIMC-SIM framework is presented in defining channel management packets and implementing a state machine.

9.1 Current Implementation

The MIMC-SIM framework is implemented in INET snapshot 20100323 with OMNET++ 4.0. In the MIMC-SIM framework, two CA protocols are implemented. One is node-based channel assignment [18] which computes channels based on superimposed code according to the CA algorithm proposed in [26] The other CA protocol is link-based channel assignment and computes channels according to the CA algorithm proposed in [19] Each CA protocol is implemented by extending the *mainControl* module to adopt its own CA algorithm. Mainly, the *mainControl* module is extended to handle channel management packets and implement a state machine. Also, the *neighborTable* module and the *nicTable* module are extended accordingly. In addition, for simulating CA protocols in a mesh network, a gateway node is implemented for each CA protocol as well. In a mesh network, a gateway node connects to another network, such as the Internet. So, in simulation, a gateway node is considered as the node that first provides a network service

according to a CA protocol.

9.2 Superimposed Code Based CA Protocol

The superimposed code (SCODE) based CA algorithm is proposed in [26] However, the implementation of the CA protocol is not fully described in the paper. In order to implement the SCODE protocol in the MIMC-SIM framework, the idea of the node-based channel assignment as [18] is adopted In a network, nodes equip two NICs and distinguish them into a receiving NIC and a sending NIC. The computed channel according to the SCODE CA algorithm is assigned to the receiving NIC.

9.2.1 Channel Management Packets

To implement the SCODE protocol, three channel management packets are defined HELLO, BEACON, and NOTICE The HELLO packet is used to probe channels When the *mainControl* module receives the HELLO packet from other nodes, the *mainControl* module responds it by broadcasting the BEACON packet The HELLO packet contains a node's IP address, MAC address of a receiving NIC, channel information, and codeword, which is used to compute a channel The BEACON packet is used to advertise a node's information It contains not only the same information of the HELLO packet, but also one hop neighbor nodes' IP addresses, MAC addresses of their receiving NIC, channel information, and codewords as well. The NOTICE packet is used to notice of new channel information of a node to neighbor nodes. The NOTICE packet contains the IP address, the MAC address of a receiving NIC, the new channel information, and

```
packet ScodePacket extends MIPacket{
    IPAddress ipAddr,
    int codeword;
    int hopPath;
    NicInfo nicInfo,
    IPAddress nIpAddr[],
    int nCodeword[];
    NicInfo nNicInfo[],
}
```

Figure 9.1: Packet Declaration For Channel Management Packets Of The Scode Protocol In A .msg File

codeword of a node. The NOTICE packet is unicasted to be transmitted to each neighbor node reliably at MAC layer. When the *mainControl* module receives all the three channel management packets, the *mainControl* module updates the *neighborTable* module accordingly Figure 9.1 shows packet declaration in a .msg file, on which the channel management packets are built Figure 9 2 shows how the *mainControl* module handles the channel management packets in a state machine

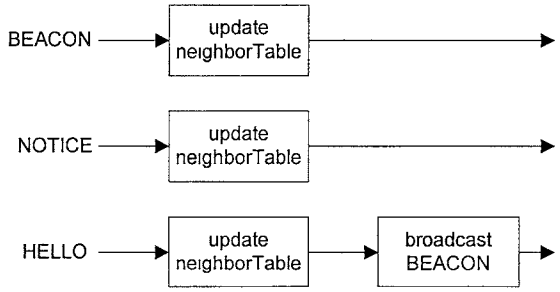


Figure 9 2 Handling Channel Management Packets In The SCODE Protocol

9.2.2 State Transition Diagram

The state machine of the SCODE CA protocol can be described and implemented in the *mainControl* module within the five states proposed in Section 5 1. In the INIT state, as mentioned in Section 5 1, the *mainControl* module is initialized and

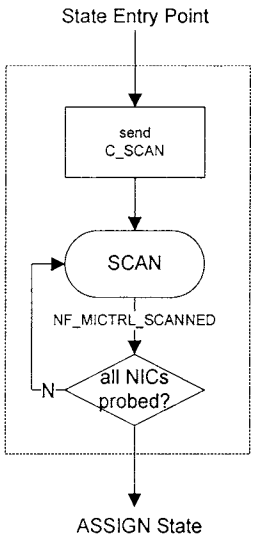


Figure 9.3: SCAN State Of The SCODE Protocol

substantiated and waits until the wakeUpTimer is expired.

In the SCAN state, the *mainControl* module sends CA commands with the HELLO packet to *subControl* modules to probe all the channels. While the *subControl* modules are probing channels, the *mainControl* module will receive the channel management packets and update the *neighborTable* module accordingly. When all the *subControl* modules notify the *mainControl* module of the completion of probing channels, the *mainControl* module enters the ASSIGN state. The inside structure of the SCAN state is depicted in Figure 9.3.

In the ASSIGN state, the *mainControl* module analyzes the information gathered in the SCAN state to compute the best channel according to [26] and decide routing based on the shortest path algorithm. If the node cannot find any valid route, then the *mainControl* module goes back to the SCAN state to probe again until a valid path is detected. The ASSIGN state is depicted in Figure 9.4.

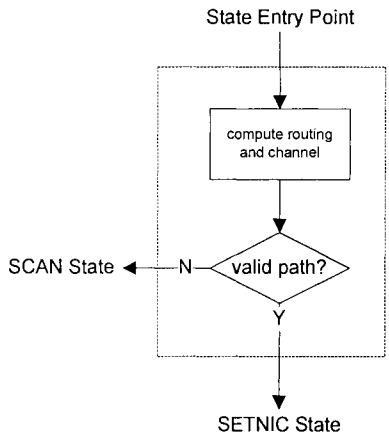


Figure 9.4: ASSIGN State Of The SCODE Protocol

In the SETNIC state, the *mainControl* module sends the CA command to assign the channel computed in the ASSIGN state to the *subControl* module associating with the receiving NIC. Once the *mainControl* module is notified by the *subControl* module about the completion of setting a NIC, it sends the NOTICE

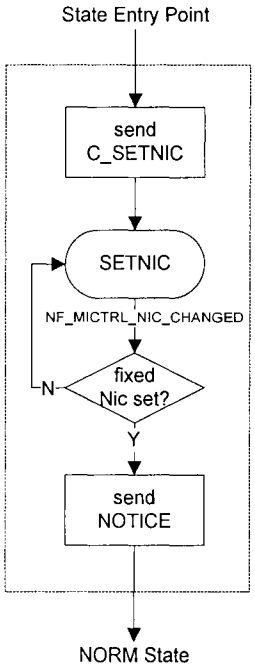


Figure 9.5: SETNIC State Of The SCODE Protocol

packets to one hop neighbor nodes about its channel information change. The SETNIC state is depicted in Figure 9.5

In the NORM state, the *mainControl* module first schedules two different timers: *beaconTimer* and *estTimer*. The *beaconTimer* indicates the next time point to broadcast the BEACON packets on all the channels. The *mainControl* module reschedules the *beaconTimer* periodically in the NORM state. The *estTimer* indicates the time point to go back to the ASSIGN state. Since nodes would learn new channel information of neighbor nodes in the NORM state, the *mainControl* module needs to go back to the ASSIGN state periodically to

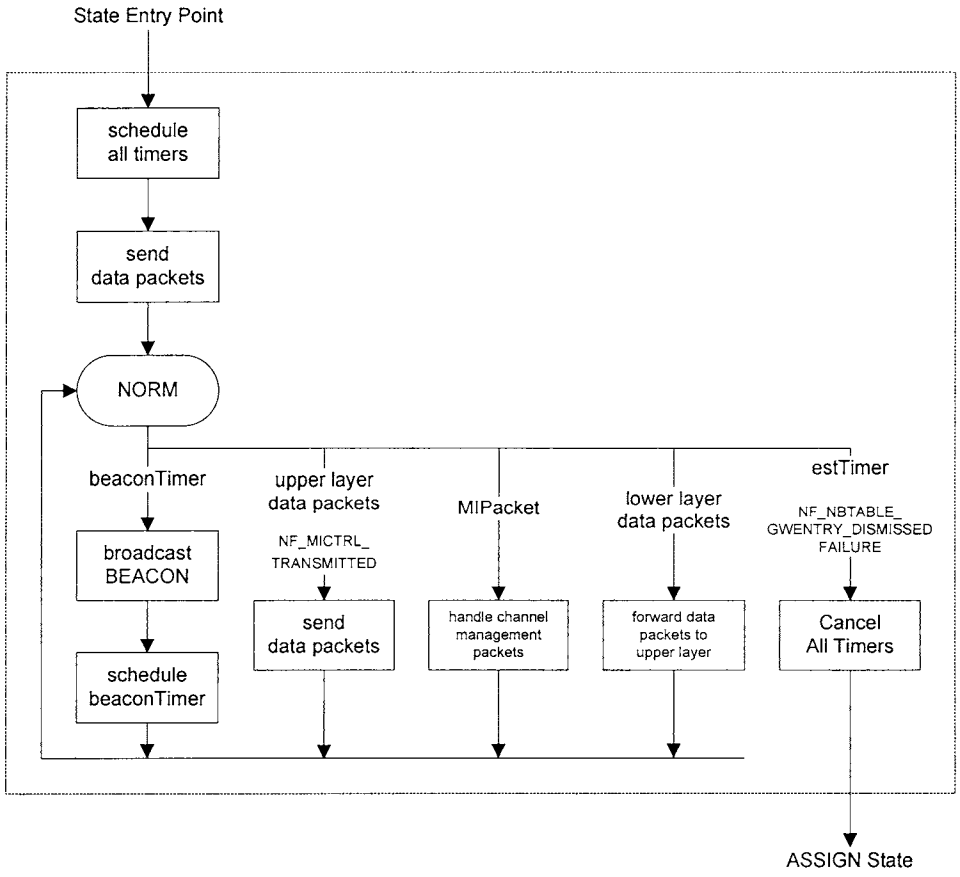


Figure 9.6: NORM State Of SCODE Protocol

re-estimate the channel information and routing For the same reason, the *mainControl* module also goes back to the ASSIGN state when the *mainControl* module is notified by the *neighborTable* module about the link break on a default route In addition, when the *mainControl* module receives channel management packets in the NORM state, it handles with them accordingly. Once the *mainControl* module enters into the NORM state, it executes the *sendDataPacket* function to send data packets in *dataQueue*. The *mainControl* module also executes the *sendDataPacket* function when it is notified by a *subControl* module of the completion of a packet transmission or receives a data packet from the upper layer. The Figure 9.6 shows the inside structure of the NORM state

In most of states, except the INIT state, the *mainControl* module receives channel management packets and data packets from lower modules. When channel management packets are received, the *mainControl* module handles them according to Section 9.2 1. When data packets are received, the *mainControl* module immediately forwards them to the upper module.

9.2.3 neighborTable

The *neighborTable* module for the SCODE protocol extends the base *neighborTable* module and is implemented as shown in Figure 9 7 The *neighborTable* module stores two more information for each neighbor node codeword and hopPath Codeword is the 13-bit code used to compute a channel according to the SCODE CA algorithm [26] HopPath is the hop distance from a gateway node It is used to decide routing according to the shortest path. The update function in the Figure

```
class INET_API ScodeNbEntry : public MINbEntryBase{
protected:
    int codeword;
    int hopPath;
        :
        :
};

class INET_API ScodeNbTable : public MINbTableBase
{
public:
    void update(const IPAddress& ip, const NicInfo& nicInfo,
        int codeword, int hopPath, int hopDistance = 1,
        bool isMainNic = false),
};
```

Figure 9.7. The Implementation Of The *neighborTable* Module In The SCODE Protocol

9.7 is implemented to update neighbor node's information conveniently

9.2.4 nicTable

The *nicTable* module for the SCODE protocol extends the base *nicTable* module and is implemented as shown in Figure 9 8. It categorizes two NICs of a node into a fixed NIC and a switchable NIC Then, the *nicTable* module classifies the fixed NIC and the switchable NIC into the receiving NIC and the sending NIC respectively

```
class INET_API ScodeNicTable    public MINicTableBase{
public:
        enum nicType{
                SEND,
                RECV,
        };

protected
        virtual void classifyNics(),
};
```

Figure 9.8: The Implementation Of The *nicTable* Module In The SCODE Protocol

Table 9 1 shows that in each node, the first NIC is categorized into a fixed NIC and set its type to a receiving NIC, and the second NIC is categorized into a switchable NIC and set its type to a sending NIC

	Category		Type	
	fixed	switchable	receiving	sending
NIC[0]	✓		✓	
NIC[1]		✓		✓

Table 9.1· Category Of NICs In The *nicTable* Module Of The SCODE Protocol

9.2.5 Gateway Node

The gateway node is assumed that it is connected to a wired network and provides a network service according to the SCODE protocol. The *mainControl* module of the gateway node does not follow the state machine explained in the previous section Instead, it always stays in the NORM state and does not try to find either better channel or route. So, the *mainControl* module does not schedule the *estTimer*, and there is no link break on the default route. The *mainControl* module broadcasts the BEACON packet periodically It also broadcasts the BEACON packet to respond the HELLO packet When the *mainControl* module receives channel management packets from other nodes, it updates them into the *neighborTable* module Figure 9.9 shows the inside structure of the NORM state of the gateway node

9.3 Multi-channel Wireless Mesh Network CA protocol

The Hyacinth CA protocol is proposed in [19] To implement the Hyacinth protocol, each node equips two NICs and categorizes them into a up NIC and a down NIC

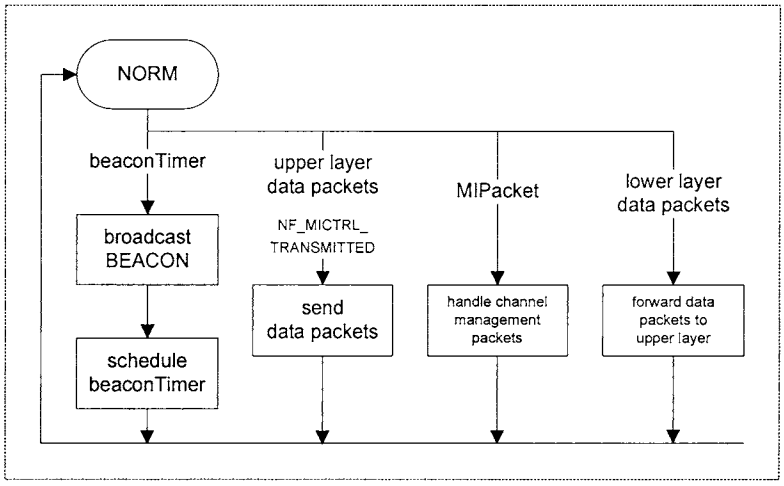


Figure 9.9: The NORM State Of A Gateway Node In The SCODE Protocol

9.3.1 Channel Management Packets

The Hyacinth protocol defines ten channel management packets: HELLO, ADVERTISE, JOIN, ACCPET, REJECT, LEAVE, RT_ADD, RT_DEL, CHNL_CHANGE, and FAILURE. To implement the channel management packets accordingly, channel management packets are defined to contain such information as described in Table 9.2.

In Table 9.2, host IP is the IP address of a node which sends or forwards a

	host IP	target IP	up NIC	down NIC	priority	neighbor info
HELLO	✓		✓		✓	
ADVERTISE	✓		✓	✓	✓	✓
JOIN	✓	✓	✓	✓		
ACCEPT	✓	✓	✓	✓	✓	
REJECT	✓	✓				
RT_ADD	✓	✓				
RT_DEL	✓	✓				
LEAVE	✓					
FAILURE	✓					
CHNL_CHANGE	✓		✓	✓		

Table 9.2: Channel Management Packets In The Hyacinth Protocol

```
class NeighborInfo{
    IPAddress ip;
    int priority,
    NicInfo nicInfo[],
}

packet HyacinthPacket extends MIPacket{
    IPAddress hostIPAddr,
    IPAddress targetIPAddr,
    int priority = 10000,
    NicInfo upNicInfo,
    NicInfo downNicInfo,
    NeighborInfo nbInfo[],
}
```

Figure 9.10 Packet Declaration For Channel Management Packets Of The Hyacinth Protocol In A .msg File

channel management packet at the last Target IP is the IP address of a node which originally generates a channel management packet or a channel management packet targets to. For example, the target IP information of the JOIN packet is the IP address of the node which generates the JOIN packet In the ACCEPT packet, the target IP information is the IP address of the node to which the ACCEPT packet is sent. Each of up NIC and down NIC information contains a MAC address and channel information. Priority is a node's hop distance from a gateway node The neighbor information includes one hop neighbor nodes' IP address, information of both up NIC and down NIC, and priority. The channel management packets are built upon HyacinthPacket declared in a msg file as shown in Figure 9 10.

Basically, all channel management packets are used as described in [19] The HELLO packet is used to probe channels When the *mainControl* module receives the HELLO packet, it broadcasts the ADVERTISE packet as response. Also, the *mainControl* module broadcasts the ADVERTISE packet periodically. The JOIN

packet is used to ask a neighbor node to join the neighbor node's routing path. When the *mainControl* module receives the JOIN packet, it sends the ACCEPT packet or the REJECT packet back to the node which originally sent the JOIN packet. The ACCEPT packet is used to accept a node to join, and the REJECT packet is used to reject a node from joining When the *mainControl* module sends the ACCEPT packet, it sends the RT_ADD packet to a parent node as well. The RT_ADD packet is used to announce that a new node joins a routing path The LEAVE packet is used to notice a parent node that a node leaves from the parent node's routing path When the *mainControl* module receives the LEAVE packet, it sends the RT_DEL packet to its parent node The RT_DEL packet is used to

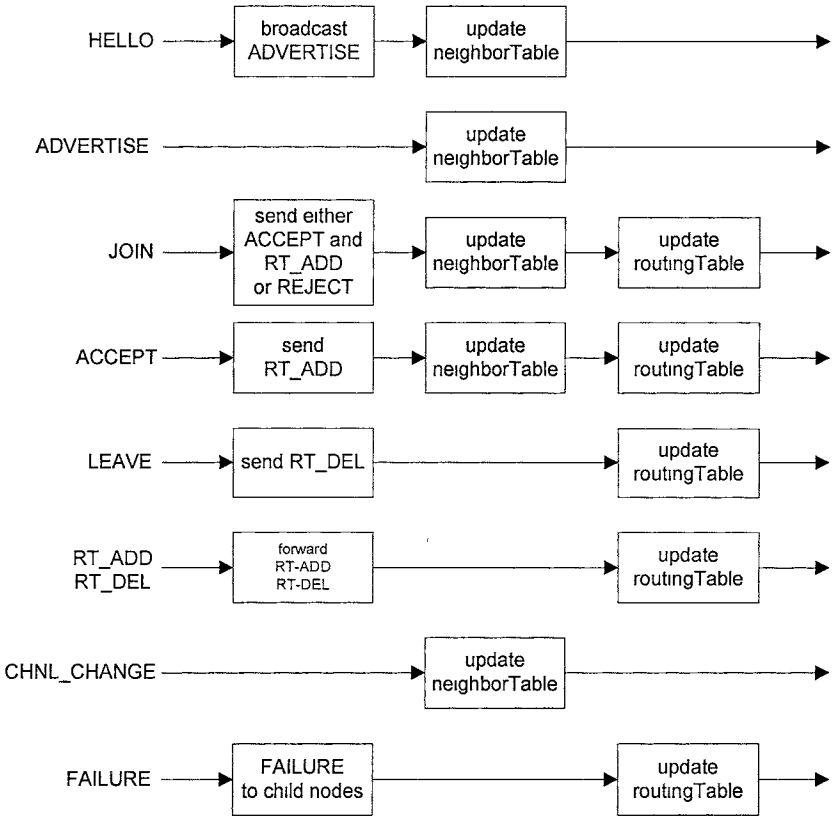


Figure 9.11: Handling Channel Management Packets In The Hyacinth Protocol

announce that a node left from the routing path. When the *mainControl* module receives the RT_ADD packet or the RT_DEL packet, it forwards such packets to its parent node until a gateway node receives the packets. The CHNL_CHANGE packet is used to announce the new channel information. The FAILURE packet is used to announce that a routing path is broken. The channel management packets are handled in a state machine as described in Figure 9.11.

When the *mainControl* module receives the HELLO, ADVERTISE, JOIN, ACCEPT, and CHNL_CHANGE packets, the *mainControl* module updates the information that they contain into the *neighborTable* module accordingly. When the *mainControl* module receives the JOIN and RT_ADD packets, it adds a route in which the host IP address is the target IP of the packet, the next hop IP address is the host IP of the packet into the *routingTable* module. When it receives the LEAVE and RT_DEL packets, it removes a route whose host IP address matches to the target IP information of the packets from the *routingTable* module. When it receives the ACCEPT and FAILURE packets, it updates the default route of the *routingTable* module. The ADVERTISE and FAILURE packets are broadcasted, while other channel management packets are unicasted to a neighbor node for reliable transmission at MAC layer.

9.3.2 State Transition Diagram

The state machine of the Hyacinth protocol can be described within the five states proposed in Section 5.1. In the INIT state, as mentioned in Section 5.1, the *mainControl* module is initialized and substantiated and waits until the

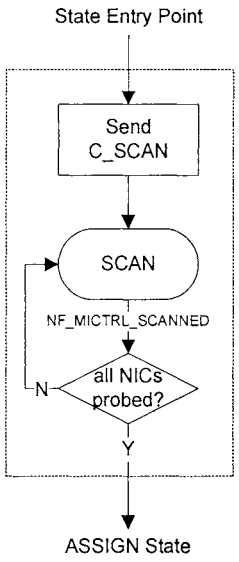


Figure 9.12: The SCAN State Of The Hyacinth Protocol

wakeUpTimer is expired.

In the SCAN state, the *mainControl* module sends CA commands with the HELLO packet to *subControl* modules to probe all channels. While *subControl* modules are probing channels, the *mainControl* module will receive the channel management packets and update the *neighborTable* module accordingly. When all the *subControl* modules notify the *mainControl* module of the completion of probing channels, the *mainControl* module enters the ASSIGN state. The inside structure of the SCAN state is depicted in Figure 9.12.

In the ASSIGN state, the *mainControl* module analyzes the information gathered in the SCAN state and selected the shortest route. If the *mainControl* module cannot find any valid route, then the *mainControl* module goes back to the SCAN state to gather more neighbor nodes' information until a valid route is detected in the ASSIGN state. After selecting a valid route, the *mainControl*

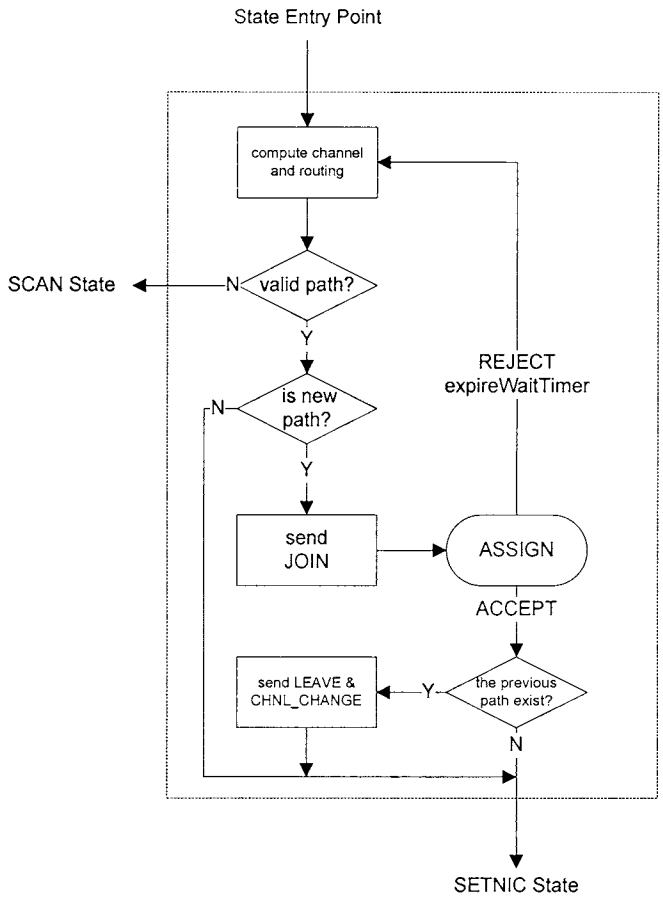


Figure 9.13: The ASSIGN State Of The Hyacinth Protocol

module sends the JOIN packet to the node providing the route. If the *mainControl* module receives the REJECT packet or does not receive either the ACCEPT or REJECT packet in a certain time the default value is 5 seconds , the node fails to join the route. In such case, the *mainControl* module re-selects another valid path. Otherwise, the *mainControl* module receives the ACCEPT packet and succeeds to join the route. If a node has a previous route, then the *mainControl* module sends the LEAVE packet to its old parent node and the CHNL_CHANGE packets to its child nodes. Then, the *mainControl* module enters the SETNIC state. The inside structure of the ASSIGN state is depicted in Figure 9.13.

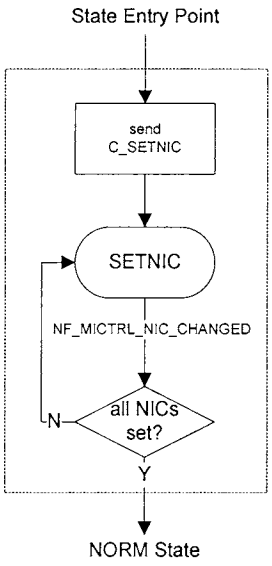


Figure 9.14: The SETNIC State Of The Hyacinth Protocol

In the SETNIC state, the *mainControl* module sends CA commands to assign channels to the up NIC and the down NIC. The up NIC is assigned to the channel which is used in the down NIC of a parent node. The down NIC is assigned to the least used channel. The *mainControl* module enters the NORM state when it is notified by all the *subControl* module about the completion of setting their NICs. The inside structure of the SETNIC state is depicted in Figure 9.14

In the NORM state, the *mainControl* module first schedules two different timers: *advertiseTimer* and *estTimer*. The *advertiseTimer* indicates the next time point to broadcast the ADVERTISE packet on the channel of the down NIC. The *mainControl* module reschedules the *advertiseTimer* periodically in the NORM state. The *estTimer* indicates the time point to go back to the ASSIGN state. Since nodes would learn new channel information of neighbor nodes, the *mainControl* module needs to go back to the ASSIGN state periodically to re-estimate the

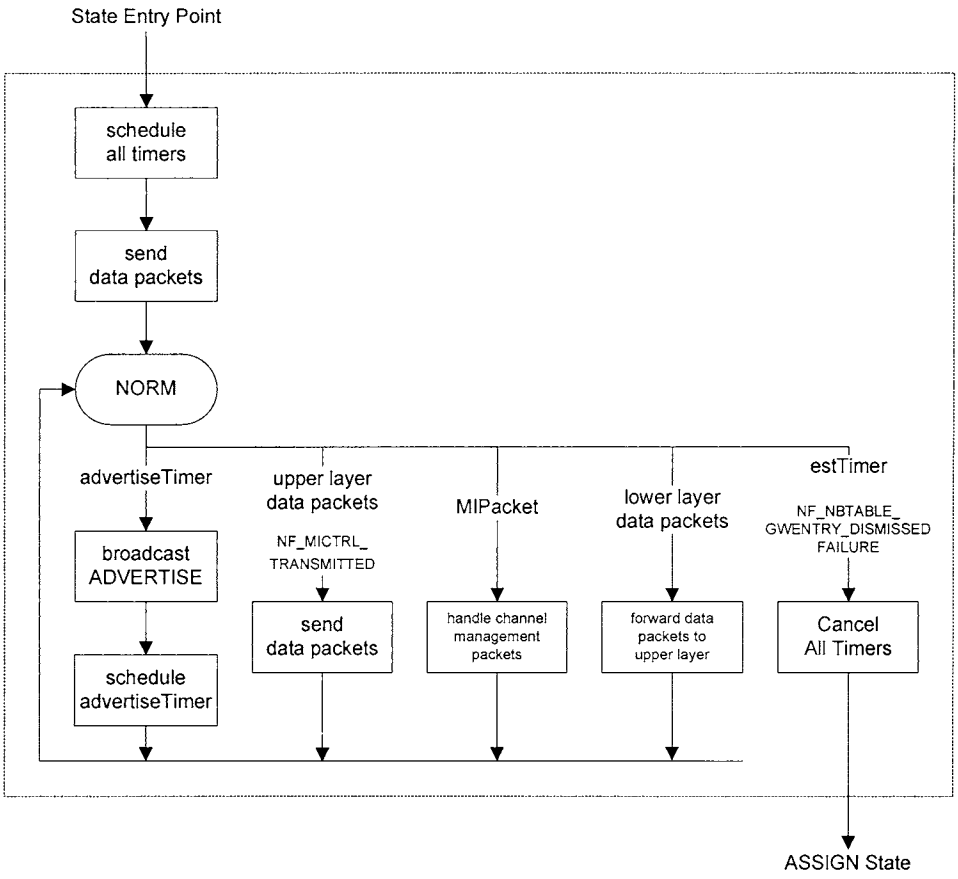


Figure 9.15: The NORM State Of The Hyacinth Protocol

channel information and routing. For the same reason, the *mainControl* module also goes back to the ASSIGN state when the *mainControl* module is notified by the *neighborTable* module about the link break on a default route or receives the FAILURE packet. In addition, when the *mainControl* module receives channel management packets, it handles with them accordingly. Once the *mainControl* module entered into the NORM state, it executes the *sendDataPacket* function to send data packets in *dataQueue*. Then, the *mainControl* module also executes the *sendDataPacket* function when it is notified by a *subControl* module of the completion of a data transmission or receives a data packet from the upper layer.

The Figure 9.15 shows the inside structure of the NORM state.

In most of states, except the INIT state, the *mainControl* module receives the channel management packets and data packets from lower modules. When channel management packets are received, the *mainControl* module handles them according to Section 9.3.1. When data packets are received, the *mainControl* module immediately forwards them to the upper module.

9.3.3 neighborTable

The *neighborTable* module for the Hyacinth protocol extends the base *neighborTable* module and is implemented as shown in Figure 9.16. The *neighborTable* module stores one more information, priority, for each neighbor node. Priority represents the hop distance from the a gateway node. The update function in the Figure 9.16 is implemented to update neighbor node's information conveniently.

```
class INET_API DrcaNbEntry : public MINbEntryBase{
protected:
    int priority;
        :
        :
};

class INET_API DrcaNbTable : public MINbTableBase
{
public:
    void update(const IPAddress& ip, const NicInfo& nicInfo,
    int priority = MAX_PRIORITY, int hopDistance = 1,
    bool isMainNic = false);
};
```

Figure 9.16: The Implementation Of The *neighborTable* Module In The Hyacinth Protocol

```
class INET_API DrcaNicTable : public MINicTableBase{
public:
    enum nicType{
        UP,
        DOWN,
    };

protected:
    virtual void classifyNics();
};
```

Figure 9.17: The Implementation Of The *nicTable* Module In The Hyacinth Protocol

9.3.4 nicTable

The *nicTable* module for the Hyacinth protocol extends the base *nicTable* module and is implemented as shown in Figure 9.17. It categorizes both two NICs into fixed NICs. Then, the *nicTable* module classifies one for a up NIC and the other one for a down NIC. Table 9.3 shows that in each node, both NICs are categorized in fixed NICs. Then, the types of the first NIC and the second NIC are set to a up NIC and a down NIC respectively.

	Category		Type	
	fixed	switchable	up	down
NIC[0]	✓		✓	
NIC[1]	✓			✓

Table 9.3: Category Of NICs In The *nicTable* Module Of The Hyacinth Protocol

9.3.5 Gateway node

The gateway node is assumed that it is connected to a wired network and provides a network service according to the Hyacinth protocol. The *mainControl* module of the gateway node does not follow the state machine explained in the previous

section. Instead, it always stays in the NORM state and does not try to find either better channel or route. So, the *mainControl* module does not schedule the *estTimer*, and there is no link break on the default route. The *mainControl* module broadcasts the BEACON packet periodically. It also broadcasts the BEACON packet to respond the HELLO packet. When the *mainControl* module receives channel management packets from other nodes, it updates them into the *neighborTable* module. Figure 9.9 shows the inside structure of the NORM state of the gateway node.

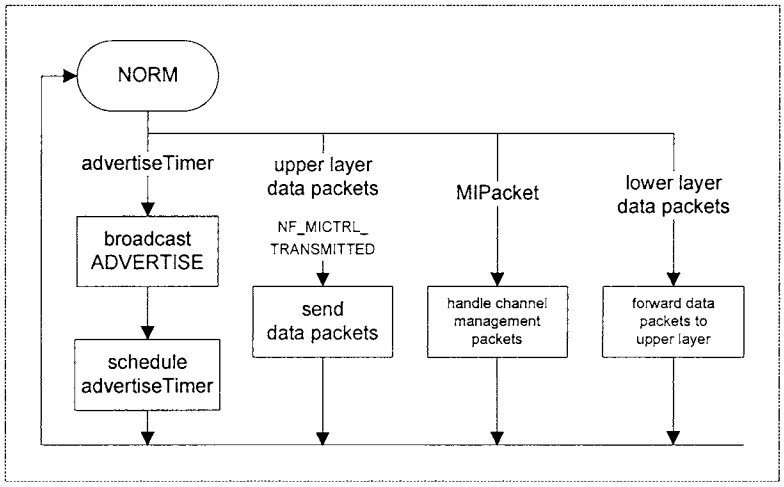


Figure 9.18: The NORM State Of A Gateway Node In The Hyacinth Protocol

CHAPTER 10

EXPERIMENTS

In this chapter, the performance of SCODE and Hyacinth protocols [19, 26] are evaluated in the MIMC-SIM framework The evaluation uses the same experimental settings as their original papers The experimental results are compared with the results reported in the original paper to verify the fidelity of MIMC-SIM. The comparison shows that MIMC-SIM can be used to study CA protocols. Furthermore, the MIMC-SIM framework is tested to evaluate various performance metrics of CA protocols, including throughput, time to obtain channels, channel management overhead, and the number of conflict channels in two hops

10.1 SCODE Protocol

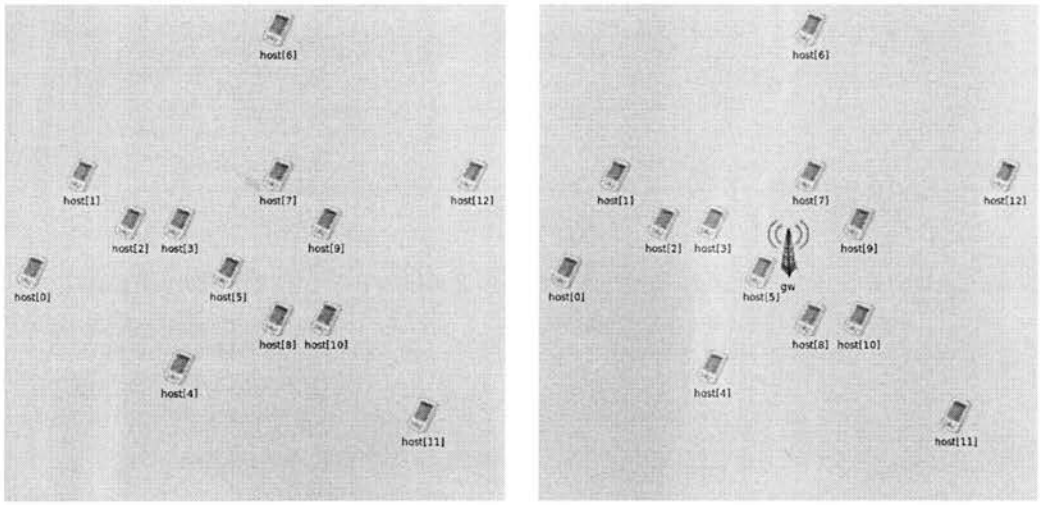
10.1.1 Setting

The SCODE protocol implemented in the previous chapter is experimented according to the original testbed described in [26]. In a 100x100 square units network, 13 nodes are deployed randomly over 100 different network topologies where average node degree is 3 Every node equips two NICs The number of available channels in a network is set to 13 The superimpose code as shown in Figure 10 1 is applied to simulate the SCODE protocol Each node randomly picks a unique codeword from the superimpose code set Since the SCODE protocol in

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

Figure 10.1: Superimposed Code

[26] is experimented in an ad hoc network, a gateway node is not deployed in a network used to compare with [26]. An example of such network topology is depicted in Figure 10.2(a). In addition, data of other metrics, such as throughput, time to get channels, and overhead traffic, are collected in a mesh network where a gateway node is deployed at center. An example of such network topology is depicted in Figure 10.2(b). The bandwidth of every link is set to 2 Mbps. Each



(a) An Example Of An Ad Hoc Network

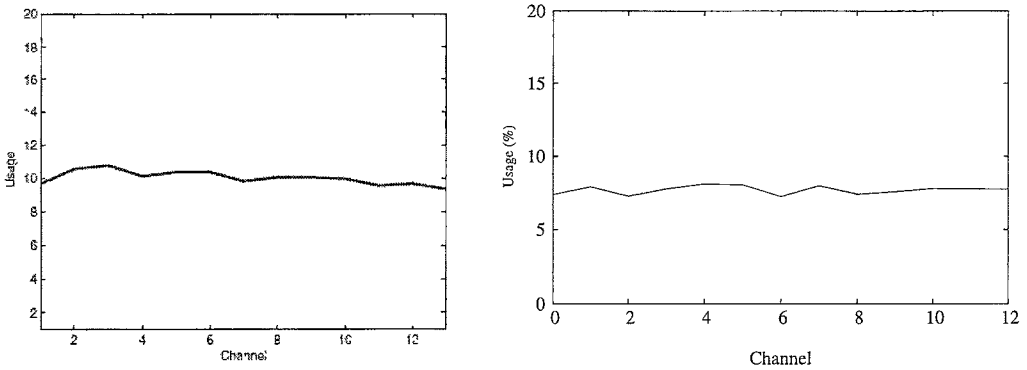
(b) An Example Of An Mesh Network

Figure 10.2: Examples Of A Network Topology

node turns on at random between 0 and 5 seconds Every node broadcasts the BEACON packet every 30 seconds and re-estimates channel information and routing every 60 seconds. Every node, except for a gateway node, starts generating UDP flows to a gateway node after 30s. The average bandwidth of each UDP flow in a network varies in 32, 64, 96, and 128 Kbps Each network topology is simulated for 600 seconds in simulation time.

10.1.2 Comparison in the original testbed

The MIMC-SIM framework produces a compatible result with [26]. [26] shows that the SCODE protocol produces fairly usage of each channel in a network as depicted in Figure 10.3(a). The similar result is also validated in the MIMC-SIM framework Figure 10 3(b) depicts the experiment result of the SCODE protocol in the MIMC-SIM framework. Figure 10 3(a) shows average number of channel usage of each channel, and Figure 10.3(b) shows average percentage of channel usage of each channel The comparison between Figure 10.3(a) and 10 3(b) can verify that the



(a) The Channel Usage Of Each Channel Tested In [26]

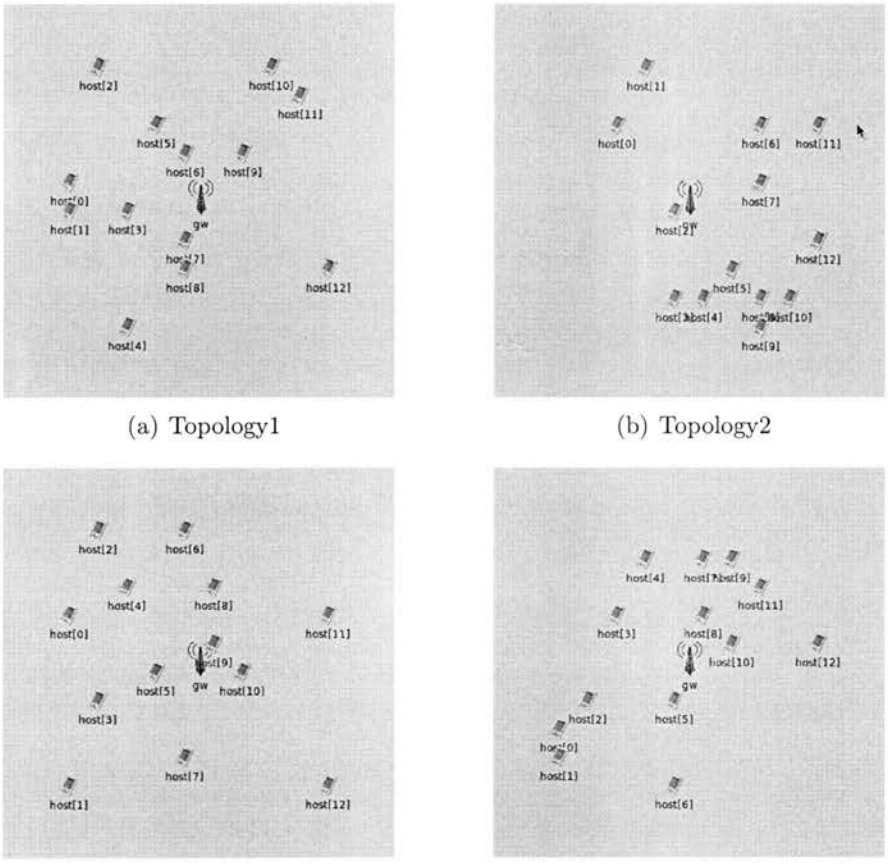
(b) The Channel Usage Of Each Channel Tested In The MIMC-SIM Framework

Figure 10.3: The Channel Usage Of Each Channel

MIMC-SIM framework is compatible to study the SCODE protocol as described in is compatible with [26].

10.1.3 Performance Study

The MIMC-SIM framework studies performance of the SCODE protocol in a mesh network with such metrics as throughput, time to get channels, and overhead traffic. (Since there is no conflict channel during simulation of the SCODE protocol, the number of conflict channels is not studied.) To study performance of the SCODE



(a) Topology1

(b) Topology2

(c) Topology3

(d) Topology4

Figure 10.4: Four Different Network Topologies To Study Performance Of The SCODE Protocol

protocol in such metrics, four specific network topologies are picked as depicted in Figure 10.4.

10.1.3.1 Throughput

Figure 10.5 shows throughput in the topology3 network when the average bandwidth of each UDP flow varies in 32, 64, 96, and 128 Kbps. (The network topology3 is picked because nodes are most evenly distributed in the network out of the four topologies.) The throughput of the network is measured by the sum of all useful bandwidth between traffic generating nodes and the gateway node in the network. In Figure 10.5, when the traffic load is bigger, average deviation of throughput is increased. This is because only one channel is used to receive packets at each node, and it causes the hidden terminal problem more often. The hidden

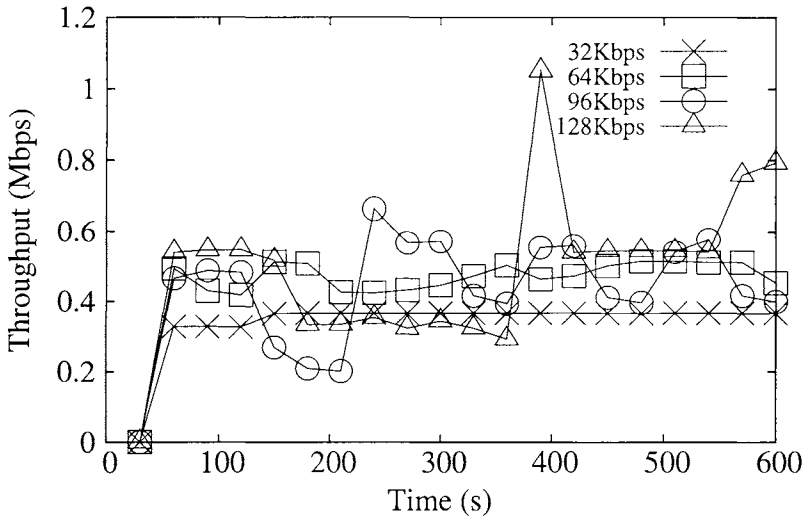


Figure 10.5: Throughput In The Topology3 Network

terminal problem is that when two nodes not in the same communication range try to send packets to the same node on the same channel at the same time, the two transmissions are interfered with each other. This suddenly aggravates throughput of a network. Since the throughput of the network is stable with better performance when the average bandwidth of each UDP flow is 64 Kbps, other metrics are studied in the situation

10.1.3.2 Channel to get channels

Figure 10 6 shows the cumulative distribution function (CDF) of the time to get the channels in the four network topologies In the SCODE protocol, the time to get channels means the time that nodes spend to obtain steady channels for their receiving NICs and will not change the channels no longer In Figure 10.6, after 300

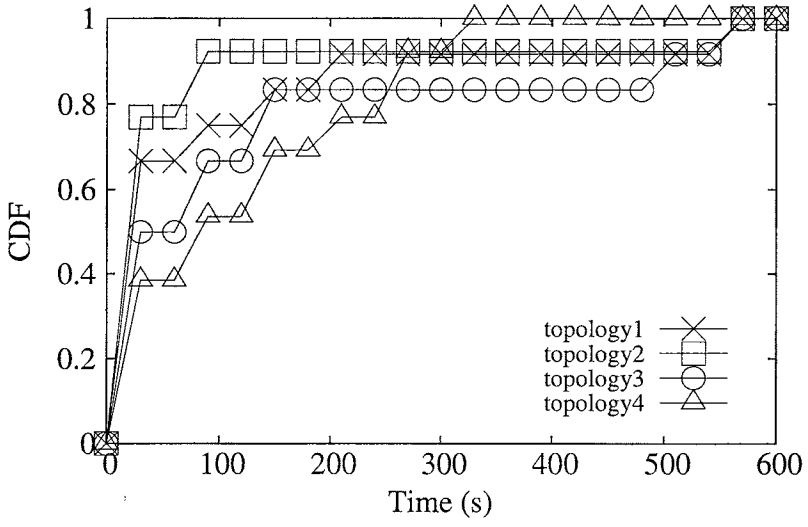


Figure 10 6· Time To Get Channels

seconds, approximately 90 percent of nodes get their final channels and stay on their channels

10.1.3.3 Overhead

Figure 10.7 shows the traffic volume of channel management packets that each node generates in a network. The traffic volume of channel management packets can be considered as overhead traffic in a MIMC networksFigure 10 7 represents that the average traffic of channel management packets becomes stable after 250 seconds when most of nodes found their final channels as depicted in Figure 10 6.

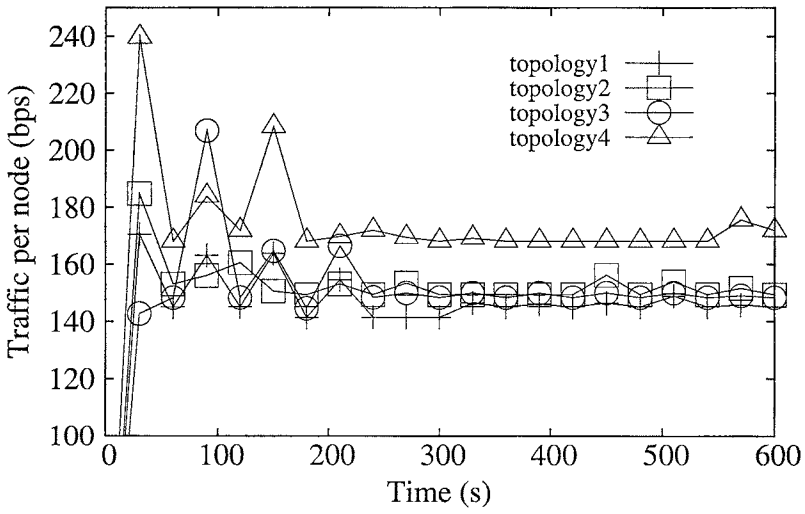


Figure 10.7: Traffic Of Channel Management Packets Per Node

10.2 Hyacinth Protocol

10.2.1 Comparison in original testbeds

10.2.1.1 Setting

The Hyacinth protocol implemented in the previous chapter is experimented according to the original testbed described in [19]. 64 nodes are evenly distributed in the 8x8 square grid network where each node could communicate with up to 4 neighbor nodes. In such network, 4 gateway nodes are uniformly deployed. An example of such network topology is depicted in Figure 10.8. Every node equips two NICs. The number of available channels in the network is set to 13. The bandwidth of every link is set to 54 Mbps. Each node turns on at random between 0 and 5 seconds. Every node broadcasts the ADVERTISE packet every 30 seconds and re-estimates the channel information and routing every 60 seconds. For 10 different traffic profiles, 20 different nodes are randomly chosen to generate UDP flows to

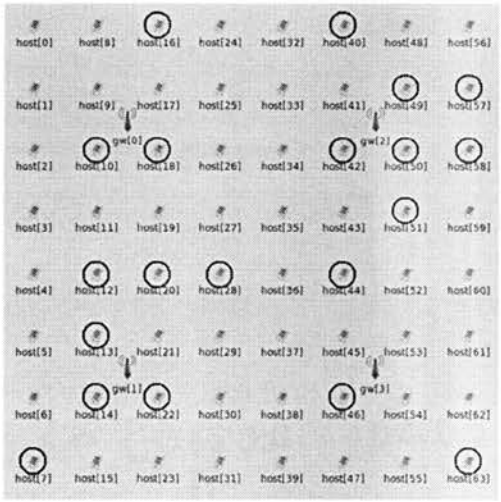
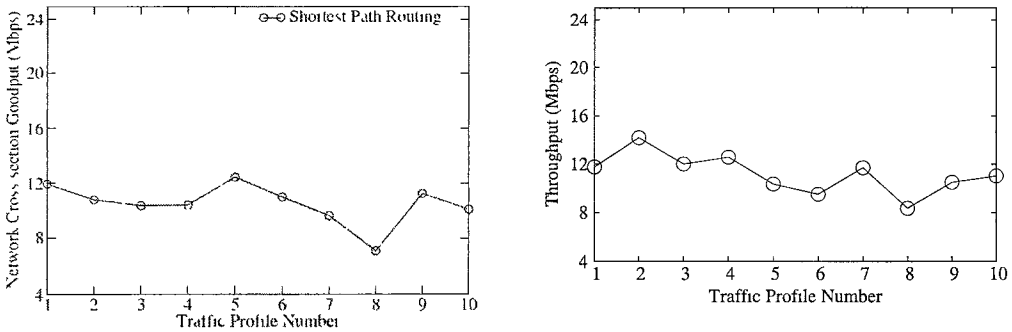


Figure 10.8: An Example Of A Network Topology

their corresponding gateway nodes in a skewed manner, specially closer to two of the gateway nodes In Figure 10 8, the circled nodes represents the nodes chosen to generate UDP flows. The average bandwidth for each flow is set to 1 5 Mbps Each traffic profile network is simulated for 600 seconds in simulation time

10.2.1.2 Analysis

The MIMC-SIM framework produces a compatible result with [19] In Figure 10 9(a), [19] shows throughput of a network that the Hyacinth protocol produces with the shortest path routing The similar result is also validated in the MIMC-SIM framework Figure 10 9(b) depicts the experiment result of the Hyacinth protocol in the MIMC-SIM framework. The throughput in the two graphs are presented within between about 7 to 14 Mbps. Both graphs retrieve equivalent average throughput. The comparison between Figure 10.9(a) and 10.9(b) can verify that the MIMC-SIM framework is compatible to study the Hyacinth protocol as described in [19]



(a) Throughput Tested In [19]

(b) Throughput Tested In The MIMC-SIM Framework

Figure 10 9 Throughput

10.2.2 Performance Study

The MIMC-SIM framework studies performance of the Hyacinth protocol in such metrics as throughput, time to get channels, overhead traffic, and the number of conflict channels. To study performance of the Hyacinth protocol in different network topologies, the testbed described in Section 10 1 1 is used Moreover, the four network topologies depicted in Figure 10.4 are used for simulation of the Hyacinth protocol.

10.2.2.1 Throughput

Figure 10.10 shows throughput of a network when the average bandwidth of each UDP flow varies in 4, 8, 12, 16 Kbps. The throughput of a network is measured by the sum of all useful bandwidth between traffic generating nodes and the gateway

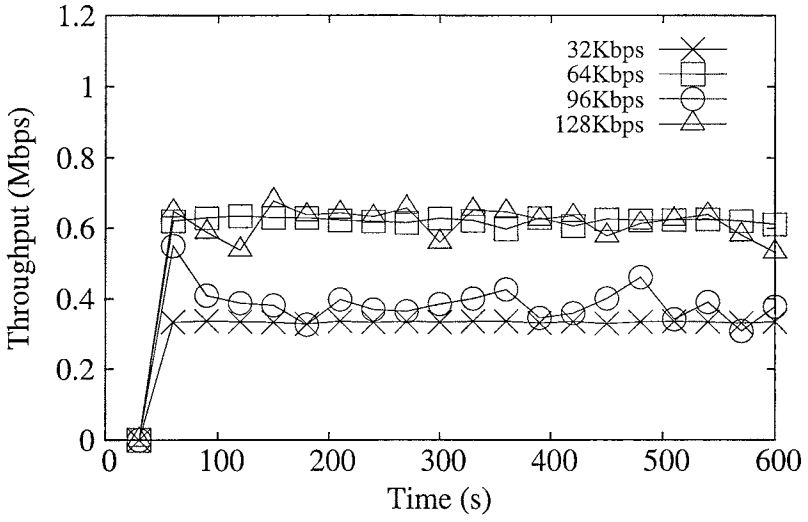


Figure 10.10· Throughput In The Topology3 Network

node in the topology3 network as depicted in Figure 10.4(c) (The network topology3 is picked by the same reason described in Section 10 1 3 1.) Figure 10 10 shows that the throughput of the network is stable This is because two different channels are used to receive packets at each node, and the hidden terminal problem less occurs in the network In addition, even though the throughput of the network is stable, the throughput can not exceed over about 0 7 Mbps Since the throughput of the network is stable with better performance when the average bandwidth of each UDP flow is 64 Kbps, other metrics are studied in the situation

10.2.2.2 Channel to get channels

Figure 10.11 shows the cumulative distribution function (CDF) of the time to get the channels in the four network topologies In the Hyacinth protocol, the time to

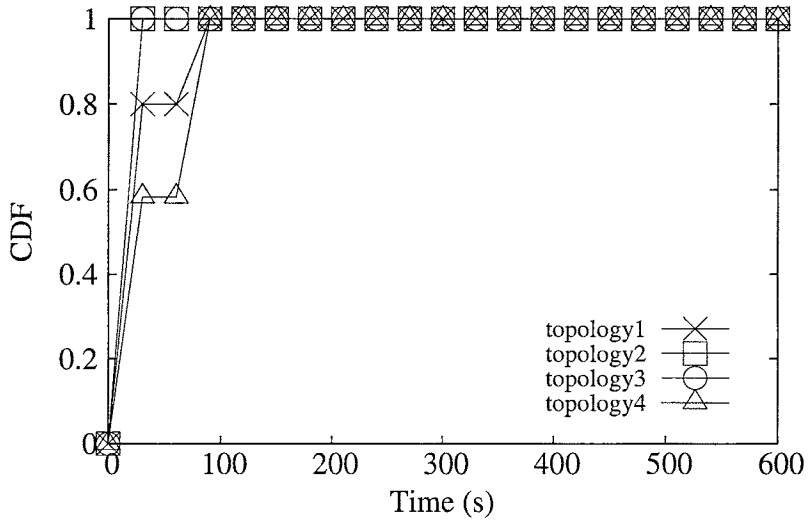


Figure 10.11: Time To Get Channels

get channels means the time that nodes spend to obtain steady channels for their DOWN NICs and will not change the channels no longer In Figure 10 11, after 90 seconds, all the nodes get their final channels and stay on their channels.

10.2.2.3 Overhead

Figure 10.12 shows the traffic volume of channel management packets that each node generates in a network. Figure 10 12 represents that the traffic volume of channel management packets becomes stable after 90 seconds when all the nodes found their final channels depicted in Figure 10.11.

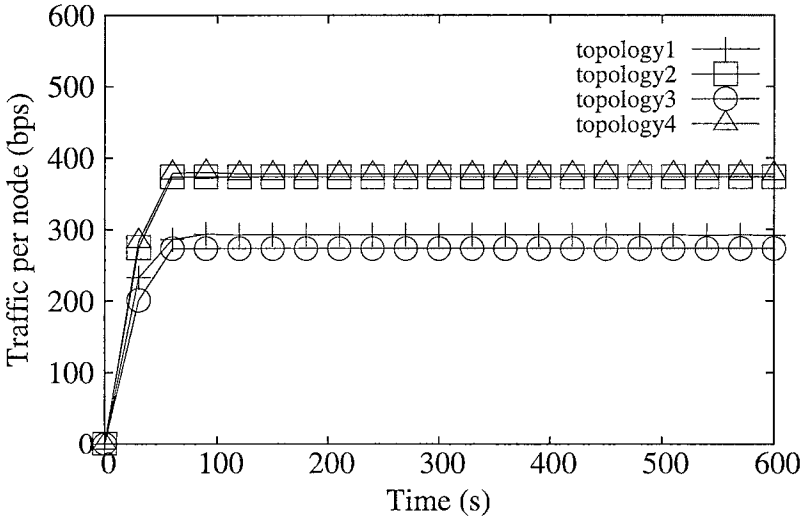


Figure 10.12. Traffic Of Channel Management Packets

10.2.2.4 Conflict

Figure 10.13 shows that the number of conflict channels per node In the Hyacinth protocol, when two nodes within two hops use the same channel for their DOWN NICs, the channel is considered as a conflict channel Figure 10.13 represents no conflict channels occur after 90 seconds

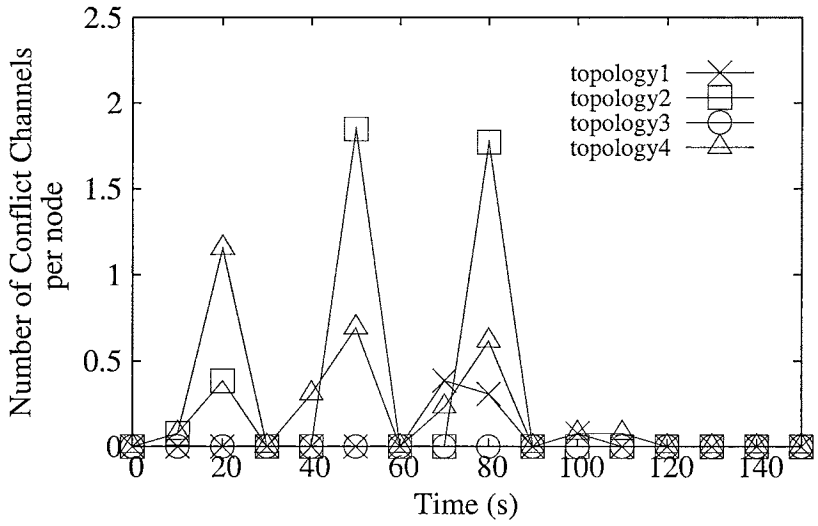


Figure 10.13. The Number Of Conflict Channels

CHAPTER 11

SECURITY

This chapter shows that the MIMC-SIM framework can be used to study vulnerability of CA protocols To study vulnerability, an attacking node is implemented to break a link between nodes Then, this chapter shows that the attacking node can aggravate throughput of a network

11.1 Attack in a MIMC Network

One of the possible attacks in a MIMC network is a link break attack between nodes. In a MIMC network, for nodes to communicate with each other, they must tune their NICs to the same channel In order to tune the same channel among nodes, nodes maintain the channel information of their neighbor nodes by exchanging channel management packets However, if nodes maintain incorrect channel information about their neighbor nodes, the neighboring nodes lose connection among themselves. An attacker can exploit this discrepancy by sending manipulated channel management packets which contain incorrect channel information of neighbor nodes

Figure 11 1 shows the steps that an attacking node M breaks a link between the nodes A and B. Nodes A and B are neighbor nodes and already established a link between them according to the same CA protocol. Also, node M knows the CA

protocol. In Figure 11.1(a), node A broadcasts channel management packets containing channel information of itself and its one-hop neighbor node, B. Also, node B broadcasts the same information in Figure 11.1(b). When node M receives the channel management packets from both nodes, it can get the channel information being used on the link between the two nodes. Node M manipulates the channel management packet to pretend node A and contain an incorrect channel information. In Figure 11.1(c), node M sends the manipulated channel management packet to node B. Node B is deceived to change its channel to the incorrect channel information for node A. After all, node B is not able to communicate with node A as shown in Figure 11.1(d). In simulation, node A is referred to a gateway node, and node B is referred to an one-hop neighbor node of the gateway node. Node M is referred to an attacking node.

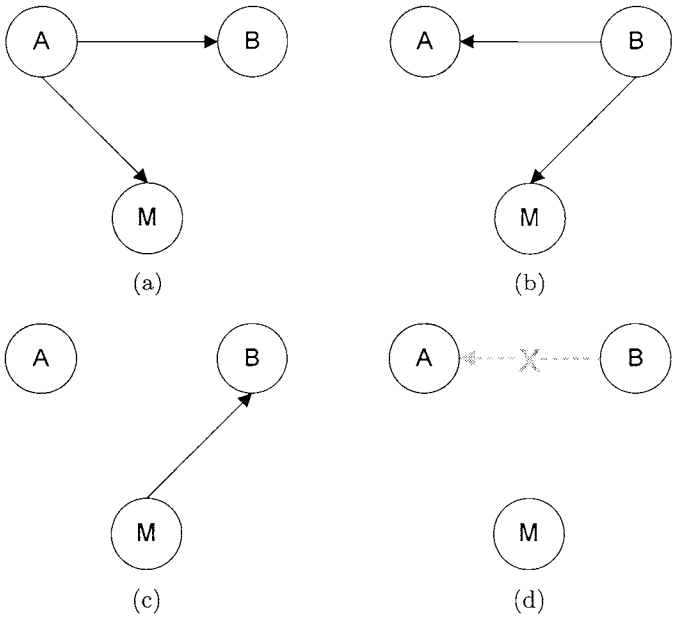


Figure 11.1: Steps Of A Link Break Attack

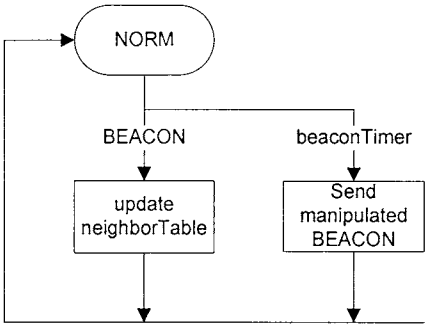


Figure 11.2: State Machine Of An Attacking Node

11.2 Implementation of Attacking Node

Two attacking nodes are implemented to study vulnerability of the two CA protocols, SCODE and Hyacinth, respectively. The attacking nodes intend to break a link between a gateway node and its one-hop neighbor nodes. Each attacking node implements its attacking mechanism in the *mainControl* module. Although the two attacking nodes manipulate different channel management packets, the structure of them can be simply generalized. Figure 11.2 shows the state machine of the attacking nodes. The state machine does not follow the state machine proposed in Section 5.1. Instead, an attacking node simply stays in the NORM state. An

```
ScodePacket *pk = new ScodePacket();
pk->setIpAddr(nbT->getIPOfGatewayEntry());

NicInfo nicInfo = *(nbT->getMACOfGatewayEntry());
nicInfo.getChannelInfo().setChannel(
  (nicInfo.getChannelInfo().getChannel() + 1) % numChannels);

pk->setNicInfo(nicInfo);
pk->setHopPath(0);
```

Figure 11.3: Implementation Of Manipulating The BEACON Packet In An Attacking Node

attacking node simply waits to receive channel management packets, especially the BEACON packet for SCODE protocol and the ADVERTISEMENT packet for Hyacinth protocol respectively. After sufficient channel information is collected, an attacking node manipulates the channel management packet to pretend a gateway node and contain incorrect channel information. Figure 11.3 shows the implementation of manipulating the BEACON packet in an attacking node. Then, an attacking node sends the manipulated channel management packet to gateway's one-hop neighbor nodes periodically.

11.3 Experiment

A few experiments are conducted to test vulnerability of the two CA protocols. To test the link break attack in a stable network, a testbed network is picked from Figure 10.4(c) and is set as described in Section 10.1.1. The average of each UDP flow is set to 64 Kbps (when the network shows better and stable throughput). Then, an attacking node is deployed close to the gateway node as depicted in Figure

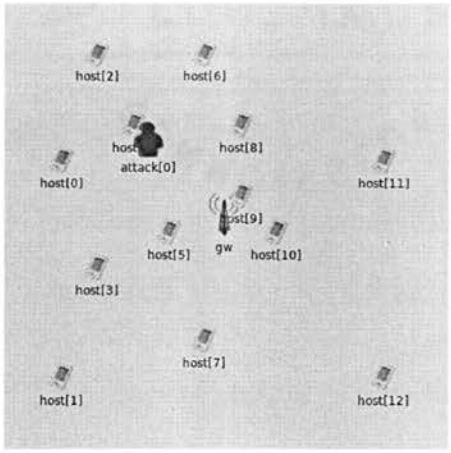
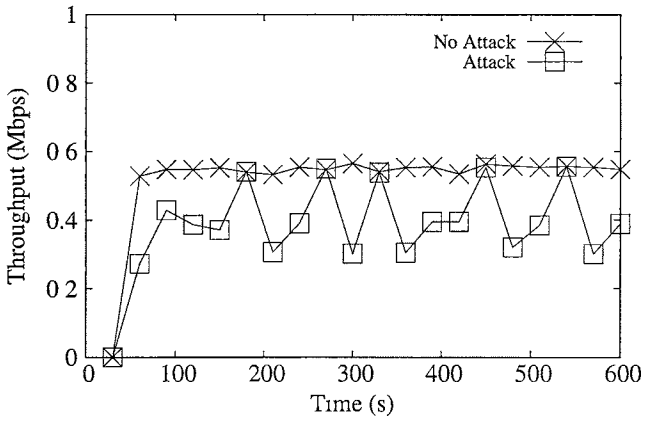


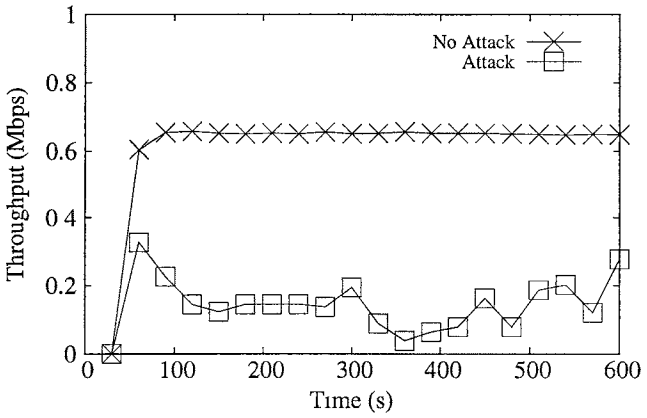
Figure 11.4: Network Topology In Which The Link Break Attack Is Tested

11 4 In the network, the attacking node can affect the hosts 4, 5, 8 and 9. In simulation, the attacking node sends the manipulated packet to its victim nodes every 15 seconds, while the gateway node sends the BEACON/ADVERTISE packets every 60 seconds

Figure 11.5(a) and 11.5(b) show the throughput of the SCODE and Hyacinth



(a) Throughput Of SCODE Protocol



(b) Throughput Of Hyacinth Protocol

Figure 11.5: Throughput Under The Link Break Attack

protocols under the link break attack respectively Both Figures show that the link break attack can reduce the throughput of a MIMC network Since the link break attack is caused by manipulating channel management packets, and the packets are maintained by a CA protocol, CA protocols should carefully design their mechanisms against such attack. Hence, such experimental result verify that the MIMC framework can be used to study vulnerability of CA protocols

CHAPTER 12

CONCLUSION AND FUTURE WORK

In this thesis, a generic simulation framework, MIMC-SIM, is designed and developed to study CA protocols in MIMC networks The MIMC-SIM framework is implemented in INET/OMNeT++ which provides great features for network simulations. In the MIMC-SIM framework, a new module is added as a new layer between the network layer and the MAC layer The new module is constructed· *mainControl*, *subControl*, *neighborTable*, and *nicTable* modules The *mainControl* module handles the operations according to the specification of CA protocols, such as handling channel management packets and computing channel and routing Also, the *mainControl* module handles packet transmission for CA protocols New CA protocols will be implemented in the *mainControl* module by extending its base class The *subControl* module performs command CA operations for all CA protocols, such as assigning channels to a NIC and scanning and probing channels. It also ensures that packets are transmitted on correct channels The *neighborTable* module maintains various information of neighbor nodes The *nicTable* module maintains information of NICs according to their roles Both *neighborTable* and *nicTable* modules are extended according to a new CA protocol In addition, the MIMC-SIM framework provides FSME to implement a state machine of CA protocols in generic and flexible code structure In the MIMC-SIM

framework, the SCODE and Hyacinth protocols are implemented and evaluated The experimental results show that the MIMC-SIM framework can be used for research and development of CA protocols Furthermore, the vulnerability of CA protocols can also be studied in the framework.

For the future work, the activities of the NORM state in the *mainControl* module can be generalized. According to the implementations of the SCODE protocol and the Hyacinth protocol, they have very similar internal structure in the NORM state. Basically, in the NORM state, both protocols allow nodes to broadcast their information, transmit packets, handle packets from lower layer and channel management packets, and go back to the ASSIGN state periodically Generalizing such operations in the NORM state will make the implementation of CA protocols more efficiently.

BIBLIOGRAPHY

[1] *EmuLab*, 2010. Avaılable at http //www.emulab net/.

[2] *GloMoSım*, 2010. Avaılable at http //pcl cs.ucla edu/projects/glomosım.

[3] *INET*, 2010 Available at http //ınet omnetpp org/

[4] *INETMANET*, 2010 Avaılable at http.//gıthub com/inetmanet/ınetmanet/wıkı.

[5] *MAP*, 2010. Avaılable at https·//engıneerıng purdue edu/MESH.

[6] *NS2*, 2010. Avaılable at http·//www ısı.edu/nsnam/ns/

[7] *NS3*, 2010. Avaılable at http·//www.nsnam org/.

[8] *OMNET*, 2010. Avaılable at http://www.omnet org/

[9] *OPNET*, 2010 Avaılable at http //www.opnet com/.

[10] *OverSım*, 2010. Avaılable at http.//www.oversım org/

[11] *UCR-Testbed*, 2010. Available at http·//networks cs ucr edu/testbed/

[12] *WINLAB*, 2010 Available at http.//www.winlab rutgers edu/

[13] A. Adya, P. Bahl, J Padhye, A Wolman, and L Zhou A multı-radıo unıfıcatıon protocol for ıeee 802 11 wıreless networks In *Proceedıngs of IEEE BROADNETS*, pages 344–354, 2004

[14] V. Bhandari and N. H Vaıdya. Capacıty of multı-channel wıreless networks wıth random (c, f) assignment. In *Proceedıngs of the 8th ACM ınternatıonal symposıum on Mobıle ad hoc networkıng and computıng*, pages 229–238 ACM, 2007.

[15] C. M. Cheng, P. H. Hsıao, H Kung, and D. Vlah. Adjacent channel ınterference ın dual-radıo 802 11 a nodes and ıts impact on multı-hop networkıng. In *Proc. of IEEE Globecom*, pages 1–6 Cıteseer, 2006.

[16] A Dhananjay, H. Zhang, J. Lı, and L Subramanıan Practıcal, dıstrıbuted channel assignment and routıng ın dual-radıo mesh networks *ACM SIGCOMM Computer Communıcatıon Revıew*, 39(4)·99–110, 2009.

[17] P. Kyasanur and N. H. Vaidya Capacity of multi-channel wireless networks· impact of number of channels and interfaces In *Proceedings of the 11th annual international conference on Mobile computing and networking*, page 57 ACM, 2005.

[18] P. Kyasanur and N. H Vaidya Routing and link-layer protocols for multi-channel multi-interface ad hoc wireless networks *ACM SIGMOBILE Mobile Computing and Communications Review*, 10(1) 43, 2006

[19] A Raniwala and T. Chiueh Architecture and algorithms for an ieee 802.11-based multi-channel wireless mesh network In *IEEE INFOCOM*, volume 3, page 2223. INSTITUTE OF ELECTRICAL ENGINEERS INC (IEEE), 2005.

[20] A Raniwala, K. Gopalan, and T Chiueh Centralized alogrithms for mutli-channel wireless mesh networks. In *Proceedings of ACM Mobile Computing and Commnunications Review*, April 2004

[21] T. Rasheed. Technical Report N 200700017, CREATE-NET Technical Report, 2007

[22] A Varga. Omnetppcomparison http //ctieware eng monash.edu au/twiki/bin/view/Simulation/ OMNeTppComparison?skin=print, 2006

[23] A Varga and R. Hornig An overview of the omnet simulation environment In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60 ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008

[24] H vom Lehn, E Weingartner, and K Wehrle. Comparing recent network simulators: A performance evaluation study Technical Report AIB 2008-16, RWTH Aachen University, 2008.

[25] E Weingartner, H. vom Lehn, and K Wehrle A performance comparison of recent network simulators In *IEEE International Conference on Communications, 2009 ICC'09*, pages 1–5, 2009

[26] K Xing, X. Cheng, L. Ma, and Q Liang Superimposed code based channel assignment in multi-radio multi-channel wireless mesh networks In *Proceedings of the 13th annual ACM international conference on Mobile computing and networking*, page 26 ACM, 2007

VITA

Heywoong Kım was born ın Jeonju, South Korea on January 14, 1982, the son of Gıljung Kim and Kanja Lee. In 2004, he receıved the degree of Bachelor of Scıence from Hanshin University in Korea. In 2008, he entered the Graduate College of Texas State Unıversıty-San Marcos Together wıth Dr Qıjun Gu, he publıshed "A Simulatıon Framework for Performance Analysıs of Multı-Interface and Multı-Channel Wıreless Networks ın INET/OMNeT++" ın 2010 He ıntends to graduate ın the fall of 2010, with the degree of Master of Scıence

Permanent Address: Kangnam Apt 101-1601, Sungbok

Suji, Yongın, Kyunggı, South Korea

Thıs thesıs was typed by Heywoong Kım.