INFORMATION TO USERS

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

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

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

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

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

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

Realtime Routing and Messaging in Wireless Networks

Chengyi Wu

A Thesis

in the Department of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montreal, Quebec, Canada

July 2001

ABSTRACT

Realtime Routing and Messaging in Wireless Networks

Chengyi Wu

Wireless technology has rapidly expanded in recent years; however, until now, it is only used to extend services in wired networks, applications that can take fully advantage of mobile communication are not appear yet. In this thesis, a pratical messaging application for mobile computing is presented and a realtime Zone Routing Protocol (ZRP) for underlying wireless networks is evaluated.

Mobile Yellow page Messaging and Retrieval (MYMAR) system is a new application for self-organized IEEE 802.11 compliant WLANs which consist of one Base Station (BS) and several Mobile Stations (MS) in each WLAN segment. In this distributed free public-access mobile network, BSs provide yellow page like local information, such as digital maps, business advertisements, and business locations, to enable mobile receivers to quickly determine driving direction and navigate facilitates of an intended service. Users retrieve and navigate such information via a user-friendly, voice recognition based hand-free command system. Other services available are voice messaging between users and voice-activated global web wireless access. The basic architecture and system operation of this wireless application system is shown.

A suitable routing strategy is also necessary to forward packets between WLAN segments. A realtime Zone Routing Protocol (ZRP), which combines proactive routing for in-zone traffic and on-demand routing for out-zone traffic, was implemented. Some optimal configurations, such as Early Detection/Termination and Backward Route Accumulation, were made to improve the ZRP performance in mobile wireless network. The processing time for each ZRP subtask was evaluated.

To my lovely daughter

ACKNOWLEDGEMENT

First of all, I would like to express my most sincere gratitude to my thesis supervisor, Dr. Ahmed K. Elhakeem, for his guidance, support, patience and encouragement during the entire course of this thesis. This work was born from his ideas and his intellectual properties; also he provided constructive suggestions from the system design to the thesis organization. I am deeply touched by his scientific attitude toward knowledge and rich experiences in the area of wireless communication. My opportunity to work in this new and exciting field of research was most appreciated. More importance to me, not only does he give me so much guidance in the research work, but also enlightens me on the love of knowledge.

Among others, I would like to thank Mr. Osman Hasan and Mr. Vanukuri D. Reddy. They were good teammates to work with in the development of the MYMAR system part of this thesis; they had provided a lot of proposal and participation.

The financial support from Communication Research Centre (Canada) is greatly acknowledged. The equipments CRC supplied made field tests of our MYMAR system possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter One Wireless Local Area Networks

Wireless communicate is a rapidly developed technology which provide the wireless network connectivity for portable, hand-held, or moving users. Like wired local area networks (LANs), wireless local area networks (WLANs) are being developed to provide universal connectivity and high connection bandwidth to mobile users in a limited geographical area [1]. WLANs are self-organizing, robust and free public-access wireless networks that use electromagnetic airwaves to communicate information from one point to another without relying on any physical connection. The data to be transmitted is superimposed on the radio carrier so that it can be accurately extracted at the receiving end.

Ideally, users of wireless networks will want similar services and capabilities that they have in common with wired networks. However, due to the characteristics of the radio channel, the wireless community faces certain challenges and limitations that may not be imposed on their traditional wired counterparts, such as a common frequency band, interference and reliability, range and bandwidth limitation, power consumption, RF safety, freedom of mobility, throughput capacity [2] [3]. It is very important that a suitable standard for WLANs is designed so that mobile users are able to communicate with other mobile and "wired" hosts in the transparent manner. The WLAN standard should appear to layers above Data Link Layer just like any other IEEE 802.x LAN (e.g. Ethernet or Token Ring). This means, in particular, that the mobility aspects are handled at the MAC level or below. The Institute of Electrical and Electronics

Engineers (IEEE) has drafted an international WLAN standard identified as IEEE 802.11 [4].

I. IEEE 802.11

The 802.11 working group took on the task of developing a global standard for radio equipment and networks operating in the 2.4GHz unlicensed frequency band for data rates of 1Mbps (mandatory) and 2Mbps (optional) for wireless connectivity for fixed, portable and moving stations within a local area. The standard describes functions and services required by an 802.11-compliant device to operate within a wireless network as well as aspects of station mobility weaving these networks.

The fundamental building block of the IEEE 802.11 architecture is the Basic Service Set (BSS), which is defined as a group of stations under the direct control of a single coordination function. Conceptually, all the stations in the BSS can communicate directly with all the other stations in a BSS. The geographical area covered by the BSS is known as the Basic Service Area (BSA), which is equivalent to a cell in a cellular communication network. Usually several BSSs can be connect to form an extended service set (ESS) to extend the wireless coverage area.

The IEEE 802.11 defines Physical Layer options and Media Access Control (MAC) layer protocol for wireless transmission. The standard does not specify technology or implementation but simply specifications for the Physical

Layer and MAC layer. Neither does the standard specify the handoff mechanism to allow mobile stations to roam between BSSs.

a. Physical Layer

Like any other network, the Physical Layer defines the modulation and signaling characteristics for the transmission of data. At the Physical Layer, 802.11 specification allows three transmission options: one infrared and two RF transmission methods, Frequency Hopping Spread Spectrum (FHSS) and Direct Sequence Spread Spectrum (DSSS), respectively. The function of the MAC protocol is common to all three Physical Layer options and is independent of the data rate.

The FHSS utilizes the 2.4 GHz Industrial, Scientific and Medical (ISM) band (i.e. 2.4000-2.4835 GHz), which is a global band primarily set aside for industrial, scientific and medial use without the need for end-user licenses. Operation of the WLAN in unlicensed RF bands requires the spread spectrum modulation [5] to meet the requirements for operation in most countries. In North America, a maximum of 79 channels is specified in the hopping set. The first channel has a center frequency of 2.402 GHz, and all subsequent channels are spaced 1 MHz apart. In order to enable multiple BSSs to coexist in the same geographical area, three different hopping sequence sets are established with 26 hopping sequences per set. The selection of a particular channel is achieved by using a pseudo random hopping pattern, thus maximizing the total throughput and reducing congestions in BSSs. The basic 1 Mbps access data rate uses two-level Gaussian Frequency Shift Key (GFSK), where a logical 0 or 1 are encoded

using frequency $Fc \pm f$ respectively. The enhanced access rate of 2 Mbps uses 4-level GFSK, where 2 bits are mapped to one frequency at a time and totally 4 frequencies are encoded.

Similar to FHSS, DSSS also uses the 2.4 GHz ISM frequency band, whereas the 1 Mbps basic rate is encoded using Differential Binary Phase Shift Key (DBPSK) and a 2 Mbps enhanced rate using Differential Quadrature Phase Shift Key (DQPSK). The spreading is done by dividing the available bandwidth into 11 subchannels, using an 11-chip Barkers sequence (10110111000) to spread each data symbol, result in 11 MHz wideband for 1Mbps data rate [6]. Overlapping and adjacent BSSs can be accommodated by ensuring that the center frequencies of each BSS are separated by at least 30 MHz. This rigid requirement will enable only two overlapping or adjacent BSSs to operate at the same time without too much interference.

The IR specification uses an infrared wavelength range from 850 to 950 nm. The maximum range is about 10m if no sunlight or heat source interfere. Typically, the IR band is designed for indoor use only and operated with non-directed transmissions to enable stations to receive line-of-site and reflected transmissions. Because of the limited distance an infrared wave can travel, IR is often used in short distance peer-to-peer connection.

The Physical Layer uses a clear channel assessment (CCA) algorithm to determine if the channel is clear. This is accomplished by measuring the RF energy at the antenna and determining the strength of the received signal, commonly known as RSSI measurement. If the received signal strength is below

a specified threshold, the channel is declared clear for data transmission. If the RF energy is above the threshold, data transmissions are deferred in accordance with the protocol rules. The standard provides another option for CCA that can be alone or with the RSSI measurement. This option uses carrier sense to determine if the channel is available. This technique is more selective sense since it verifies that the signal is the same carrier type as 802.11 transmitters. The best method to use depends upon the levels of interference in the operating environment.

The Physical Layer is divided into two sub-layers (see Fig. 1-1). The Physical Medium Dependent (PMD) sublayer deals with the characteristics of the wireless medium and defines a method of transmitting and receiving data through the medium. The Physical Layer Convergence Procedure (PLCP) sublayer specifies a method of mapping the MAC-layer Protocol Data Units (MPDUs) into a framing format suitable for the PMD sublayer. It is important to note that the PLCP preambles and headers (12 and 4 octets in DSSS) are always transmitted at the basic bit rate of 1 Mbps. This allows a lower-rate (and lower cost) IEEE 802.11 WLAN to interoperate with a higher-rate (and higher cost) counterpart. At the same time, the relatively low rate of 1 Mbps enables the PLCP preambles and PMD headers to be decoded

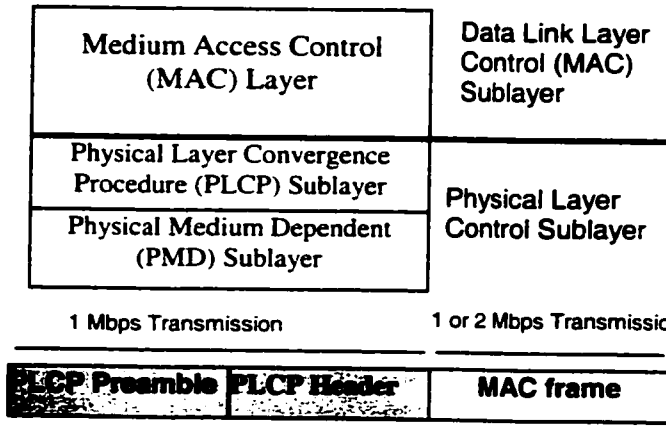


Figure 1-1 IEEE 802.11 Basic Reference Model

without using power hungry equalizers.

b. Medium Access Control Layer

The MAC layer is responsible for the channel allocation procedures, protocol data unit (MPDU) addressing, frame formatting, error checking, and fragmentation and reassembly. It also offers support for roaming, authentication, and power conservation for wireless nodes. The MAC layer specification for 802.11 has similarities to the 802.3 Ethernet wired LAN standard. While CSMA/CD algorithm of detecting a collision used in the 802.3 wired LAN, the IEEE 802.11 MAC protocol uses the carrier sense multiple access with collision avoidance (CSMA/CA) scheme to avoid collisions. Unlike wire transmission, in an RF transmission of wireless network, a station cannot transmit and sense the channel at the same time because of its receiver would be overwhelmed by the transmitting power of its own transmitter. Therefore, it is impossible for a station to detect collisions like in 802.3 LAN, which listens for collision at the same time of transmitting. It is for this reason that a new method of collision avoidance is introduced in 802.11.

The standard IEEE 802.11 MAC frame format is illustrated in Figure 1-2:

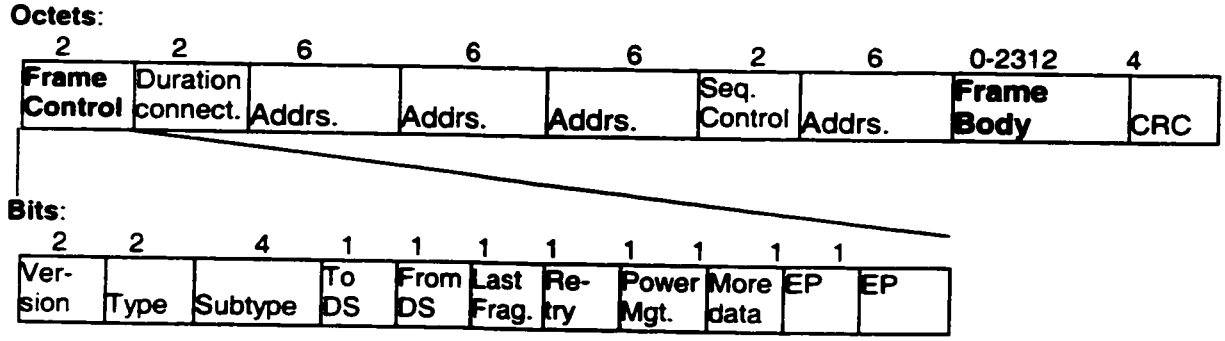


Figure 1-2 Standard IEEE 802.11 MAC frame format

Some significant features of IEEE 802.11 MAC frame are:

1. The frame body (MPDU) is a variable length field consisting of the data payload and 7 octets for encryption/decryption if the optional Wired Equivalent Privacy protocol is implemented.
2. The IEEE standard 48 bit MAC addressing is used to identify a station.
3. The meaning of Address 1 to 4 depends on the 2 DS-bits in the frame control field, which will identify the physical receiver(s), transmitter and BSS identification (see Table 1-1).

Table 1-1 Address structure in 802.11 MAC frame

	To DS	From DS	Addr. 1	Addr. 2	Addr. 3	Addr. 4
Ad hoc	0	0	Physical Receiver	Physical Sender	BSS ID	Unused
From AP	0	1	Physical Receiver	BSS ID	Logical Sender	Unused
To Ap	1	0	BSS ID	Physical Sender	Logical Receiver	Unused
Between AP	1	1	Receiving AP	Sending AP	Logical Receiver	Logical Sender

4. The 2-octet duration indicates the time (in microseconds) the channel will be allocated for successful transmission of a MAC protocol data unit (MPDU).
5. The type bits identify the frame as either one of control, management, or data.
6. The subtype bits further identify the type of frame (e.g. Clear to Send control Frame).
7. A 32-bit cyclic redundancy check (CRC) is used for error detection.

In MAC data or management frame, there is a field called duration information which specifies the amount of time (in microseconds) after the end of the present frame the channel wiil be utilized to complete the successful transmission. All stations in the BSS extract MAC header and use this duration information to adjust their network allocation vector (NAV), which indicates the amount of time that must elapse until the current transmission session is complete and the channel can be sampled again for idle status. This algorithm is referred to as virtual carrier sensing, which combines with the air interface physical carrier sensing in the Physical Layer to detect the channel idle status. While the physical carrier sensing detects the presence of other IEEE 802.11 WLAN users via related signal strength from other sources, virtual carrier sensing is used by the source station to inform other stations in the BSS of how long the channel will be utilized for the successful transmission of an MPDU. The channel is marked busy if either the physical or the virtual carrier sensing mechanism indicates channel is busy.

The basic access method in 802.11 is the Distributed Coordination Function (DCF) which implements the IEEE 802.11 MAC protocol, known as CSMA/CA protocol. IEEE 802.11 also provides another contention free access method, Point Coordination Function (PCF). The PCF is based on polling algorithm that is controlled by an access point (AP), which cyclically polls stations in its BSS giving them the opportunity to transmit in the shared medium. The PCF itself relies on the DCF method.

According to DCF, before initiating a transmission, a station must sense

the state of the channel to determine if another station is transmitting. The station proceeds with its transmission if the medium is determined to be idle for a time interval greater than a distributed interframe space (DIFS) (see Fig. 1-3). If the medium is busy, the station defers until after another DIFS is detected, and then generates a random back-off period (Contention Window) for an additional interval before transmitting. In particularly, Contention Window and idle DIFS are slotted, and stations are only allowed to transmit at the beginning of each slot time. This slotted random back-off scheme further minimizes collisions during contention between multiple stations.

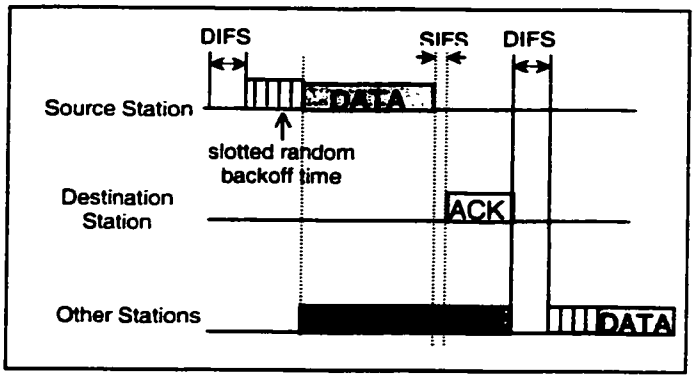


Figure 1-3 Basic MAC transmission sequences

Explicit acknowledgments are required since a transmitter cannot determine whether a data frame is successfully received by listening to its own transmission as in wired LANs. Immediate positive acknowledgment is employed to determine the successful reception of each data frame. This is accomplished by the receiver initiating a transmission of an acknowledgement frame (ACK) after a time interval, the short interframe space (SIFS), immediately following the reception of the data frame. Since the SIFS is, by definition, less than the DIFS (10 *us* comparing to 50 *us* in DSSS), the receiving station does not need to sense the medium before transmitting the acknowledgment. In case an acknowledgment is not received, the data frame is recognized as lost and a

retransmission of the frame is scheduled.

c. RTS/CTS Extension

However, due to multipath fading, signal blocked, or signal interference from nearby BSSs reusing the same physical-layer characteristics, transmission medium degradation can cause transmission corruptions to occur very often in wireless communication compared to wired LANs. Another very common limitation with wireless LAN systems is the "hidden node" problem. This can disrupt 40% or more of the communications in a highly loaded WLAN environment [7]. It happens when there is a station that cannot detect the media is busy while another station in the same service set is transmitting. For example, a station is able to successfully receive frames from two different transmitters, but an obstruction prevents the two transmitters to receive signals from each other. Thus, a transmitter may sense the medium as idle even if the other one is transmitting. This result in the both transmitters could try to transmit at the same time, and a collision occurs at the receiving station. Furthermore, since a source station in a BSS cannot hear simultaneously to detect collisions while transmitting, when a collision occurs, the sources continuous transmitting the complete MPDU and waiting for acknowledgment time-out. If the MPDU is large, a lot channel bandwidth is wasted due to a corrupted MPDU.

To deal with these problems, the IEEE 802.11 MAC protocol includes an option that can minimize collisions by using request to send (RTS), clear-to-send (CTS), data, and acknowledge (ACK) transmission frames, in a sequential fashion (see Figure 1-4). The short RTS and CTS frame includes the destination

and the length of message. Communications is established when one of the wireless nodes sends a short message RTS frame to the receiver announcing the upcoming transmission after sensing DIFS. Having received the RTS, the receiver sends back a corresponding CTS frame in response to the received RTS frame to indicate the readiness to receive the data. If the CTS frame is not received within a predefined time interval, the RTS is retransmitted by the transmitter after executing the back-off algorithm. After a successful exchange of the RTS and CTS control frames, the channel bandwidth can be reserved, and a data frame transmission is performed after waiting for a SIFS interval. After the data frame is received, an ACK frame is sent back verifying a successful data transmission.

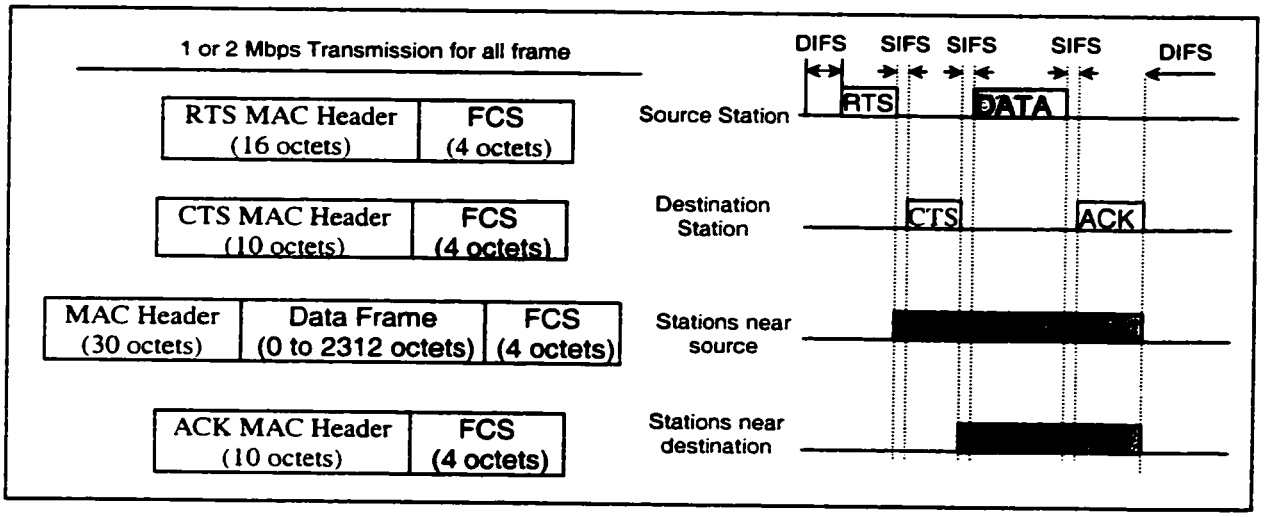


Figure 1-4 802.11 MAC frames and transmission sequences

The use of RTS, CTS, Data, and ACK sequences helps to prevent the disruptions caused by the hidden-terminal problem. Both RTS and CTS frames contain a duration field that reserves the time interval necessary to completely transmit the data frame and the related acknowledgement. This duration

information is used by all stations which can hear either the transmitter RTS frame or the receiver CTS frame to update their NAV timer. Stations need only receive either one of RTS/CTS frame to know how long the channel will be used. Thus, the probability of a collision caused by a hidden station is greatly reduced.

Furthermore, the RTS/CTS extension can be regarded as a way to improve the MAC protocol performance. RTS and CTS control frames are relatively small when compared to the maximum size date frame, thus they do not consume channel bandwidth much. However, the dynamic channel reservation abilities greatly reduce the probability of collisions. Moreover, with RTS/CTS enabled, collisions can obviously occur only during the transmission of the RTS frame instead of data frame. Since the RTS frame is shorter (20 octets) than the ordinary data frame (34 to 2346 octets), the wastage in bandwidth and time due to the collision are reduced, especially when MPDUs are large.

However, for a lightly loaded medium environment, the overheads of RTS/CTS frames introduce additional delay. In addition, the increased overhead may be significant for short data frames. Thus, the effectiveness of the RTS/CTS mechanism depends on the length of the data frame to be "protected". It is reasonable to think that the RTS/CTS mechanism improves the performances when data frame sizes are large compared to the size of the RTS frame. As a result, the RTS/CTS mechanism may rely on a threshold, a manageable parameter RTS_Threshold. The RTS/CTS mechanism is enabled for data frame sizes over the threshold and disabled for data frame sizes under the threshold. Stations can choose to never use RTS/CTS, use RTS/CTS whenever the MPDU

exceeds the value of RTS threshold, or always use RTS/CTS.

d. Broadcast or Multicast

IEEE 802.11 only support reliable unicast traffic, in some case such as management and multimedia applications, broadcast or multicast traffic is common. While broadcast is the nature of wireless communications, multicast often heavily relies on broadcasting packets to achieve multicast operations. Even though IEEE 802.11 do not specify broadcast or multicast protocols, broadcast or multicast is possible within the current 802.11 standard. In PCF access method, as an AP controls all traffic, source sends its packet to the AP, and then broadcasting or multicasting is done by AP itself, so there is no problem on broadcast or multicast. In DCF access method, however, problems arise while multiply acknowledgements or CTS frames from receivers collide with each other. Thus, broadcast or multicast cannot be treated the same as unicast in the DCF access method.

Broadcast or multicast frames use a special broadcast address or pre-registered multicast grouping address in the address field of MAC frames. All broadcast or multicast frames are required to be transmitted at the basic rate of 1Mbps to insure that all stations in the BSS can properly receive. Moreover, broadcast or multicast MPDUs are not acknowledged since this will lead to collision among multiple ACKs. Similarly, RTS/CTS cannot be applied to MPDUs with broadcast and multicast addressing due to a high probability of collision among a potentially large number of CTS frames. That is said, broadcast or multicast MPDUs are sent blindly without any control frames to assure the

availability of the destinations, source nodes simply execute collision avoidance (DIFS carrier sensing and random back-off scenarios) and then transmit the data frame. As the transmitter does not employ acknowledgement mechanism to check whether collision happenes or the data frame is correctly received by all receivers, the delivery of the broadcast or multicast traffic is not guaranteed due to hidden terminals or channel noise.

To insure reliable transmission broadcast or multicast packets in wireless communications, many proposals have been suggested to improve broadcast reliability by extending 802.11 protocols in ad hoc networks. The simplest way is to unicast the MPDU to each destination one by one. Although this method guarantees the reliability but it is costless. Another way to solve the above problem is simply to broadcast the multicast frame multiple times with a higher sequence number each time. A mobile node needs to receive correctly only once. If under the assumption that the neighbor nodes must be able to "hear" the source station and the packet error rate is usually small, another approach based on the use of negative acknowledgement (NAK) instead of normally positive acknowledgement (ACK) will work reasonably well. However, reliable packet delivery is still not guaranteed. Some other approaches suggested receivers send CTS/ACK only with a certain probability, or send a CTS/ACK feedback with a random delay to avoid collisions.

To prevent CTS/ACK frames collision problem, a leader-based error recovery multicast protocol (LBP) was proposed in [8]. LBP approach involves the election of one of the multicast receivers as a "leader" or representative of the

multicast group for the purpose of sending CTS/ACK feedback to the sender in response to RTS and Data frame, respectively. The leader node works same as the standard; on erroneous reception of a packet, the leader does not send an acknowledgement, prompting a retransmission. In the mean time, receivers other than the leader only send negative CTS (NCTS) or negative acknowledgement (NAK) to the sender. If one or more group members are not ready or receive erroneous transmission, one or more NCTSs/NAKs will collide with the positive feedback from the leader, thus destroying the feedback from the leader and prompting the sender to retransmit the packet. This approach is very simple to implement and can be integrate easily with the current IEEE 802.11 standard.

[9] proposed a straightforward broadcast support extension to IEEE 802.11 MAC protocol in ad hoc networks, Broadcast Support Multiple Access (BSMA). BSMA facilitates the transmission of broadcast packets and therefore supports multicast protocols too. BSMA incorporates the collision avoidance and RTS/CTS control frames of 802.11 and relies on negative acknowledgements to deliver broadcast packets. The protocol assumes the radio station has so called direct sequence capture ability, which is the capability to lock onto a sufficiently strong signal in the presence of other interfering, less powerful signals. That is, if more than one signal overlaps at a receiver, the receiver always pick up the most powerful signal, so a packet with the strongest signal can be successfully received while the other colliding packets are dropped. This principle is used by the transmitter to correctly receive CTS frames from its neighbors.

The following outlines the extension steps of BSMA to support broadcast packets:

1. Same 802.11 collision avoidance phase.
2. Source sends RTS to all neighbors and sets timer to WIAIT_FOR_CTS.
3. Upon receiving RTS, neighbors send CTS if not in YIELD state and set timer to WIAIT_FOR_DATA. (Receivers are busy or in YIELD state can abandon current activities, change to CTS state).
4. If source receives CTS (from the strongest neighbor), sends DATA (senses the channel state again before transmitting) and sets timer to WIAIT_FOR_NAK. Otherwise, if no CTS received and WIAIT_FOR_CTS timer expires, back off the transmission and go to step 1. Neighbors who not involved in broadcast/multicast, upon receiving RTS or CTS, set their state to YIELD and update their NAV timer according to the duration information.
5. Neighbors send NAK when WIAIT_FOR_DATA timer expires, if DATA has not been received, or errors occur in the DATA frame. Neighbors that correctly receive the data frame will remain silent.
6. If source receives NAK before WIAIT_FOR_NAK timer expires, back off and go to step 1. Otherwise, no NAK received and WIAIT_FOR_NAK expires, source node assumes all neighbors have successful received DATA frame and the broadcast is complete.

In BSMA, the sender sends the data packet as long as it receives any CTS. Some of receivers may in YIELD state and not send CTS to respond RTS, they may miss broadcast or cause collisions. Thus, BSMA scheme enhances delivery but does not guarantee reliable broadcast or multicast services. Nevertheless, BSMA does reserve channel bandwidth for correct delivery and does solve hidden terminal problem, greatly improves the reliability of packet broadcasting or multicasting.

II. AD HOC Networks

The IEEE 802.11 standard defines two types of WLAN network protocols: client/server mode and ad hoc mode. The standard specifies the manners that each station must observe so that they all have fair access to the wireless media. It provides methods for arbitrate requests to use the media to ensure that the throughput is maximized for all of the users in the BSS.

The client/server networks like the traditional client/server wired networks but with wireless connections. The client/server network uses an access point (AP) that controls the allocation of transmitting time for all stations. While an AP works as the centralized server to control network accesses and communications, all other stations can only establish network connectives with the AP as clients. The AP is analogous to the base station in a cellular communications network. In client/server networks, communication typically takes place only between the wireless nodes and the AP, but not directly between the wireless nodes (forwarding via the AP). The client/server network mode is based on Point Coordination Function (PCF) access method, a contention free polling algorithm controlled by the AP. Such centralized control over multiple accesses provides pretty reliable medium access control and robustness in the presence of fading and dynamic channel characteristics. Particularly, since typical wireless networks have an infrastructure that connects base stations to serve mobile nodes under their coverage, the client/server mode is usually established to provide wireless users with specific services and range

extension. The AP acts as an integrated bridge to neighboring WLAN segments or wired backbone LAN. Thus, allows mobile stations to roam from cell to cell.

On the other hand, an ad hoc network is a simple network where communications are established peer-to-peer among any wireless stations in a given coverage area without the help of an access point. Any station can establish a direct communication session with any other station in the same BSS. The client/server network relies on the existence of at least one AP or some infrastructure segments. However, in some areas or in certain circumstances, where no infrastructure is provided or no fixed AP is possible, ad hoc network is the only choice. In contrast to the client/server network, ad hoc network is the distributed like system where stations in a BSS are grouped together for the purpose of inter-networked communications and communications are established between multiple stations without the help of APs. An ad-hoc network exhibits the greatest flexibility possible without the requirement of channeling all traffics through an infrastructure network, so no access point controlling is necessary. Ad hoc networks are based on the Distributed Coordination Function (DCF) access method that implements the CSMA/CA scheme mentioned above. The DCF is designed for asynchronous data transport, where all the users with data to transmit have equally fair chance of accessing the network. All stations have equal priorities and hence an equal chance of accessing the shared channel. It is similar to traditional packet networks supporting best effort delivery of the data.

III. Future Developmen

While more and more products following the IEEE 802.11 standard are available, researchers have begun to discuss enhancements of the standard and development of new applications.

Most of past wireless communications are focus on extending wired network to wireless end-points via APs by 802.11 client/server access mode. However, Bluetooth [10], a de facto standard established and promoted by the wireless industry since 1998, supports ad hoc like WLAN with a very limited coverage and without the need for an infrastructure. All Bluetooth devices have the same networking capacities, either master or slave. In a BSS, called piconet in Bluetooth, connections can be initiated by any device, which then become the master whereas all other devices act as slaves. Within a piconet, only one master can exist at any time, which controls medium access using a polling and reservation scheme. A master can also template leave its piconet and act as a slave in another piconet, this jumping between piconets enable packet communication between these nets.

IEEE 802.11a [11] is a WLAN standard following the 802.11 standard but using the 5 GHz band. The higher carrier frequency allows for transmission rates up to 20Mbps, also called broadband communication. IEEE 802.11b deals with the provision of higher data rates at 2.4 GHz ISM band. It discusses fully compatible packets with headers at the standard 1 Mbps transfer rate, but payloads at higher data rates of about 3 Mbps for FHSS and 11 Mbps for DSSS

Physical Layer.

The European Telecommunication Standards Institute (ETSI) also standardized a WLAN, High Performance Local Area Network (HIPERLAN [12] for short). It allows for node mobility and support ad hoc and infrastructure-based topologies. HIPERLANs operate at 5.1-5.3 GHz with data transfer at 23.5 Mbps. The service offered by a HIPERLAN is compatible with the standard MAC services known from IEEE 802.x LANs.

To extend the need for the global Internet to support the mobility of hosts, mobile IP is developed recently to deliver IP packets to mobile users that may roam from their home networks to foreign networks. Also, to participate in the rapidly growing field of mobile communications, ATM network is extended to wireless ATM (WATM) to offer the mobility for wireless terminals. WATM supports the same integrate services and different types of traffic streams as ATM does in fixed networks.

Chapter Two MYMAR SYSTEM - -

Mobile Yellow page Messaging and Retrieval system

Since IEEE 802.11 standardized the wireless LAN communication, many mobile wireless devices have been developed. Providing wireless networking abilities for portable and mobile computing devices, such as Laptops, Palms, Personal Digital Assistants (PDAs), Cellular Phones, Pagers, and other consumer electronics; and allowing these devices to communicate and inter-operate with one another are rapidly evolving areas of research and development. However, until now, there are few practical application products appearing in the market to take full advantage of the mobility and self-organizing features of the wireless communication.

In this chapter, we try to integrate the current available technologies into an application product for the wireless devices. The overall objective of this application system is to provide yellow pages like information of a specific area to mobile users via RF links while users are roaming out of town. We call it Mobile Yellow page Messaging and Retrieval system, MYMAR for short. Basically, MYMAR is an Intelligent Transportation System (ITS) intended to be a commerce advertisement tool and to offer non-real-time two-way communications for tourists, travelers and dwellers who carry wireless mobile devices. It is a distributed wireless telecommunication system with no centralized base station, no cellular system, and no payment for service providers. The MYMAR

information may consist of business advertisement, driving maps and direction, interested destinations, and current local events. The roamer with a mobile computing device equipped with GPS and RF receiver can access to the MYMAR system and get the local information immediately through simple voice commands. Furthermore, short voice messages can be exchanged between MYMAR users, and wireless access to the Global Internet can be provided. The culmination of such services within the MYMAR system gives rise to a new industry approach, Wireless IP via WLANs, rather than the existing Cellular systems [13].

In this chapter, we present the basic architecture and system operation of the MYMAR system with emphasis on the system design and software development. The intent of the present work is to set up commercial IEEE 802.11 compliant WLANs and develop MYMAR application programs. Due to the mobility factor, the user may move out of the current WLAN, in this case the associated packets may be routed via a suitable wireless routing strategy [14][15][16] (details see next chapter). The present work is only concerned with the application aspects of MYMAR and thus no underlying packet routing strategy is used. But the applications are developed in a fashion that they can be compatible with any appropriate routing strategy that will be developed in the near future. By the use of an appropriate routing algorithm to route traffic to mobile users in different WLANs, our MYMAR system can be implemented globally. Our MYMAR system has been developed and tested in collaboration with Communication Research Center (CRC), Canada.

I. MYMAR System

MYMAR is an application of setting a network of IEEE 802.11 compliant interconnected WLANs, with each WLAN segment consisting of one MYMAR information provider, called Base Station (BS) and a number of mobile MYMAR users, called Mobile Stations (MS). MYMAR system resides over a number of interconnected wireless local area networks operating in RF bands. An example of the MYMAR network illustrated in Figure 2-1 consistings of three interconnected WLANs, which can scale to a larger number as well. Depend on the wireless transfer range and traffic congestion, it is recommended that each BS covers an area of 2-3 km radius. Thus, a big region is usually divided into several WLANs with each spanning 2-3 km radius region and acting like cells in cellular telephony.

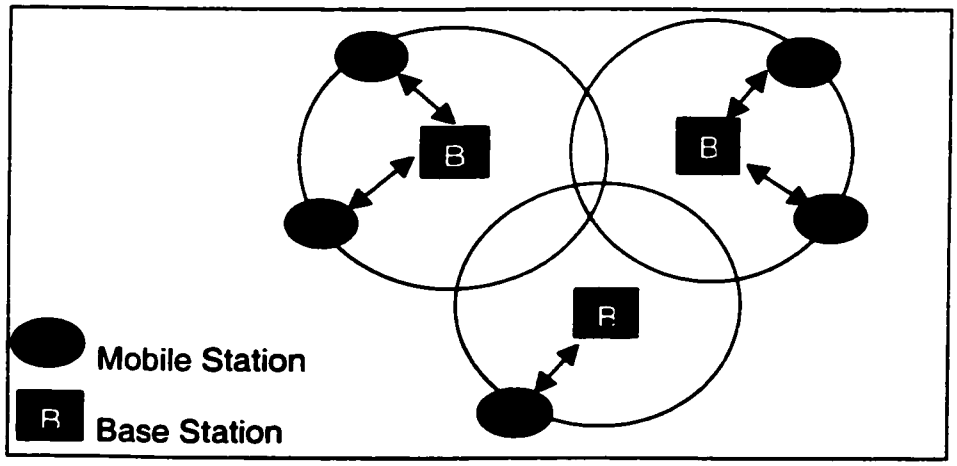


Figure 2-1 MYMAR Network

To fulfill wireless communication, all end users, both BSs and MSs, must be equipped with an IEEE 802.11 compliant wireless LAN adapter. The WLAN adapter is implemented as ISA card for desktop computer or PCMCIA card for

notebook; for some small mobile device, it is even integrated inside the device. The nature of the wireless connection is transparent to the client network operating system (NOS) via device driver. In the present prototype design, GRE GINA 2002 (PCMCIA Card) WLAN adapters [17] are used to develop wireless connectivity within the MYMAR network. Main specifications of this GINA adapter are listed in Fig. 2-2. The GINA unit utilizes Direct Sequence Spread Spectrum (DSSS) of the IEEE 802.11 standard to provide WLAN connectivity for portable computers.

- PCMCIA or ISA Bus Card
- 2.4 GHz Direct Sequence Spread Spectrum (DSSS) Modulation
- Support: 802.11, ad-hoc
- Data Rate: Up to 2 MHz.
- Range: (line of sight)
  - Unipole 12 db Antenna - 8 km
  - Yagi Antenna - 20 miles
- Transmitter Power: 800 mW
- Operation: Windows 95/98, NT4.0

Figure 2-2 Specifications of GRE GINA Wireless Adaptor

GINA's unique design provides a very high performance over other wireless adapters with a long-range distance communication, high output power, and high receiver sensitivity capabilities. At present, GINA only supports Window operating system; this limits our MYMAR system only exists in Window system for the time being.

Each WLAN in MYMAR consists of two types of stations, a Base station and several mobile stations. BSs are fixed and placed near advertisement providers (McDonalds, Sheraton, Malls etc.), so MYMAR information will direct and attract their customers, who are MSs moving across MYMAR WLAN segments. The BSs keep up-today MYMAR information about their own surrounding regions, such as region maps, desired services, interesting destinations, and advertisements. All these MYMAR Yellow page information will

deliver to MSs upon requests. BSs may also provide services like voice message and global Internet access.

A Mobile Station (MS) is a client station and a number of MSs may be located in the same WLAN communicating with the same BS. Whenever a MYMAR user (MS) enters a MYMAR network, it initiates a BS identification mode to locate the nearest BS; this is followed by the MS request of MYMAR Yellow Page data retrieval from the associated BS. Then, having local MYMAR data, roaming travelers with MYMAR application would be able to display the regional maps, get directions, and discover information about intended services and facilitate around them, such as Hotels, Restaurants, Shopping Centers, Road Services, and emergency services like hospitals, police stations etc. Other MYMAR services include short voice messaging to other users and global web access. To facilitate using MYMAR application while driving a vehicle, all MYMAR application commands are voice activated, so a traveling driver can access all MYMAR services without getting his hands off the wheel. A wireless network, digital maps, and user-friendly voice recognition system enables the driver to reach an intended destination in the shortest possible time.

To transfer MYMAR data between BS and MS, File Transfer Protocol (FTP) is used. In addition, we set the BS as a proxy server to allow wireless global Internet access from MSs. To improve the compatibility of our MYMAR system to all wireless devices and operating environment in the future development, most of our MYMAR programs are written in the well-suited JAVA language. Java Running Environment (JRE) 1.3 programs must be installed in all

MYMAR machines. In our developing progression, Java Developing Kit (JDK) version 1.3 from SUN Microsystems, Inc. [18] is used.

II. MYMAR Base Station

The Base Stations (BSs) of MYMAR system are usually provided by local advertisers and distributed among them. BS may not necessary be a portable computer like the laptop, for economic reason, it can be any desktop PC equipped with a ISA-bus wireless LAN adaptor, which have same specifications as above mentioned PCMCIA card for the portable computer. To cover much larger areas, a more powerful antenna for BS usually is used and is mounted in a higher altitude in an open-air environment. In addition, a high-speed Internet connection is preferred if this BS acts as a ground-based proxy server to provide global Internet wireless access to its MS users. The hardware components of the BS used in our test is shown in Figure 2-3, using a laptop equipped with a PCMCIA wireless adaptor and Internet access with 56K modem.

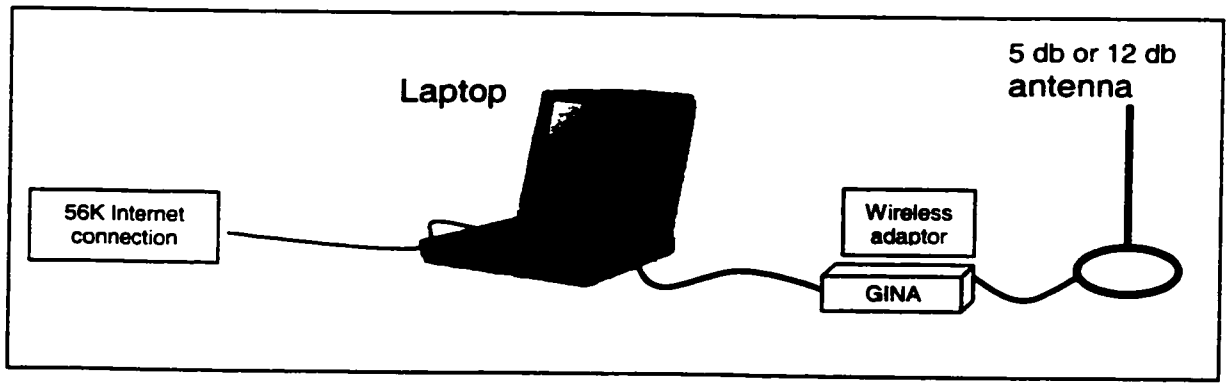


Figure 2-3 Basic Components of the MYMAR Base Station

The MYMAR BSs are the main source of yellow page information in the MYMAR system. All queries and requests from MSs are directed to the BSs. BSs keep the MYMAR Yellow Page data, which consists of digital maps for the local region along with longitude/latitude coordinates for each map, the business locations with GPS longitude/latitude value, as well as respective commercial messages or advertisements including special features, discount offers etc. File transfer protocol (FTP) is used for data transfer between a BS and a MS, thus every BS must have an FTP server program installed. Furthermore, to support global Internet wireless access for the mobile clients, a proxy server program must be installed in BSs. All these BS server side programs are running automatically in background once upon a BS is powered on.

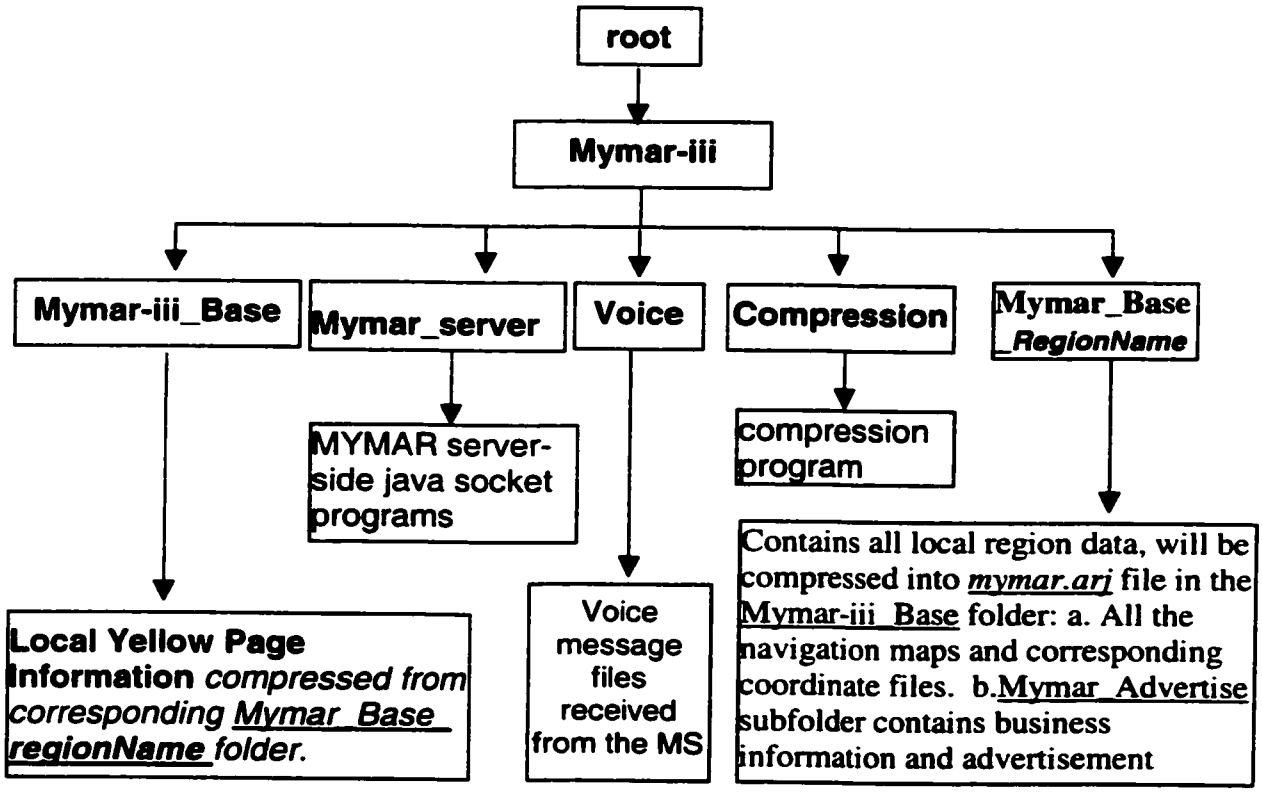


Figure 2-4 **Directory Structure in the Base Station**

Figure 2-4 is the directory structure of MYMAR system resided in the Base Station. The main folder is the Mymar-iii under the root directory, with 5 main sub-folders correspond to each implemented application:

* *Mymar_server folder*: This folder has a MymarMultiServer.java program which is MYMAR server-side Java-socket programs for BS identification. This server program waits for the requests from MSs and serves the request.

* *Voice folder*: This folder has the voice message file (Voice_Message.wav) which is sent from the mobile-station to the nearest BS. The file is supposed to redirect to the registered BS of the receiver (which will be implemented in future). However, at present, we can see it as the message has delivered to the destining registered BS and waiting for the receiver to retrieve. In future development, there should have separated subfolders for each registered MS to store voice messages.

* *Mymar-iii_Base_RegionName folder*: Here *RegionName* is the corresponded name for a region or municipal city, that is, there is a separate folder named "Mymar_Base_Brossard/Montreal... " depended on the location name. This folder contains local Yellow Page Information data. It includes the region maps and the GPS coordinates of those maps. The main city map (Big-map) file is named as "Map.gif" and the zoomed-in region maps are named as "Map*.gif", where * is the region number 1,2...n shown in big map. The corresponding GPS coordinates for each map are in a text format file named map*.txt. This folder also has a sub-folder *Mymar_Advertise,* which includes all the advertisement related files in its different subfolders, such as: BurgerKing, GasStation, Police... the list goes on

depending on each catalogue of business advertisements. Each of these subfolders has the information of the corresponding business beginning in a web page named *.htm, where * is the business name, and the GPS coordinates of its location (destination) in *.txt file. The web page may contain links to browse through its whole business web site, and the GPS coordinates will be used in the MSs to show the location on the map depending on the user request.

* *Compression folder:* There contains a small compression utility used to compress data sent to the client by the BS. The files *arj.com* and *compress.bat* are used to compress MYMAR data in the "Mymar_Base_*regionName*" folder depending on the location name of *regionName*.

* *Mymar-iii Base folder:* This is the folder holding all of the local MYMAR data in the compressed format, which will be retrieved by the mobile users. The compressed file is named "Mymar.arj", compressed from a folder named Mymar_Base_*RigionName*, where *RegionName* is the corresponded name for each region or municipal city.

III. MYMAR Mobile Station

A Mobile Station (MS) can be either a laptop or some similar portable computer with a small monitor screen and a sound card. Figure 2-5 illustrates the basic elements of our MYMAR MS. GINA wireless LAN adaptor provides wireless communication between nodes of the MYMAR system. A microphone is needed to take voice commands while speakers to playback voice messages, so

usually a high quality headset microphone will be used to increase the accuracy of voice recognition. Furthermore, every MS may equip with a GPS receiver to get its current position with longitude/latitude value, which in turn can be projected on the digital map to provide navigating direction.

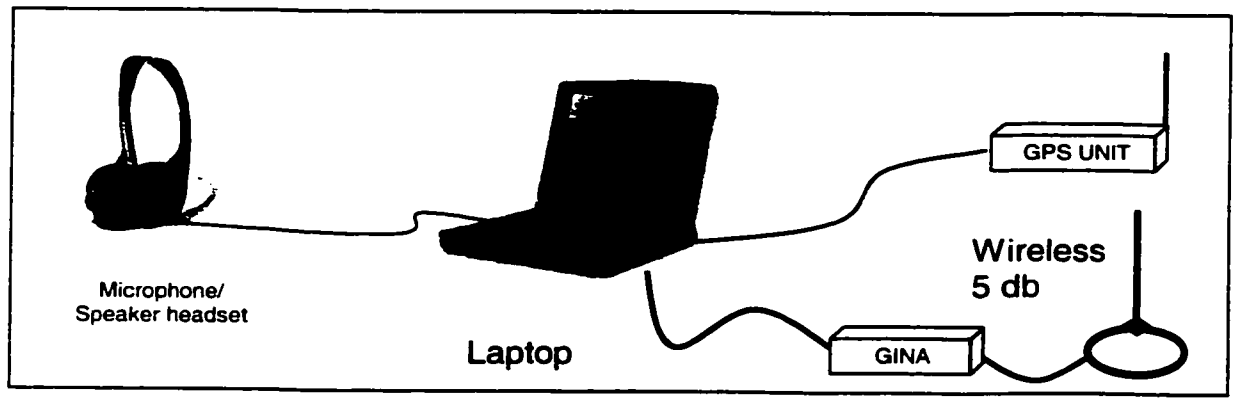


Figure 2-5 Basic elements of the MYMAR Mobile Station

Nevertheless, each component in our system is separately removable. For instance, at any time the mouse and keyboard can always be used even if MS does not use voice recognition software to take voice commands. Without the GPS receiver unit, only affect on showing the user's current position in the displayed map. A more important feature of our MYMAR system is that once local MYMAR data is successfully transferred to the user, MYMAR Yellow Pages still can be used to show the last updated information, even the MS roams out in a rural region with no BS nearby.

a. Voice Recognition

To provide hand-free access to the yellow page information for a mobile user while driving a vehicle, all system commands in the MYMAR application are

voice activated. In the present prototype model, REALIZE VOICE LITE [19] software from Realize Software Com. is used for the voice recognition. This package includes almost all the system commands for Windows environment such as 'Close Window', 'Switch Window', 'Press Enter', and ' Page Down' etc. It also supports user defined voice shortcuts (macros), allowing association of a user defined phrase with a command or batch file, thus a particular voice phrase command can result in the execution of the associated command or batch file. Thus, batch files play a key role in our voice activated MYMAR application. Every MYMAR application and command has its own BATCH file, activated by a short voice phrase (voice command).

However, using voice recognition technology also limits our selection of application programs in the MYMAR system. First, a voice command is associated to a DOS batch file, so it only supports programs that can be started from old DOS environment. In order to use fewer voice commands to perform many functions of the MYMAR system, we associate one voice command to a batch of programs in the batch file. Each of these programs may run several sessions automatically. However, it is hard nowadays to find a program that can perform automatically according to presetting controls (command line parameters) without the activity of the mouse and keyboard.

b. GPS Receiving Unit

As described above, one of the important features in MYMAR is that the MS is equipped with global positioning receiver (GPS). The GPS receiving unit

continuously receives its position from GPS system, and the continuously updating position values will then be projected on the local map to yield the driving trail. In this way, a user has instant and real time knowledge of its current position relative to the surroundings, which are also being displayed on the map. This GPS guiding system, along with yellow page messaging and retrieving mechanism through simple voice commands, makes it possible for a mobile user to navigate safely into a new geographical areas where he is unfamiliar with.

The Global Positioning System (GPS) is a worldwide satellite radio-navigation system, formed from a constellation of 24 satellites and their ground stations [20]. A GPS unit uses these satellites as reference points to calculate global positions in terms of longitude and latitude with an error of a few meters. In our test, we use a GPS receiver provided by Canadian Marconi Company [21].

GPS receiver produces a message about its current location twice every second. The output message is a long string of characters that can be read by the computer from the serial port to which it is connected. There are many GPS message formats, but we use a standard NMEA protocol, GLOBAL POSITIONING SYSTEM FIX DATA format (GPGGA). The GPGGA data string has fields and formats showed in Figure 2-6.

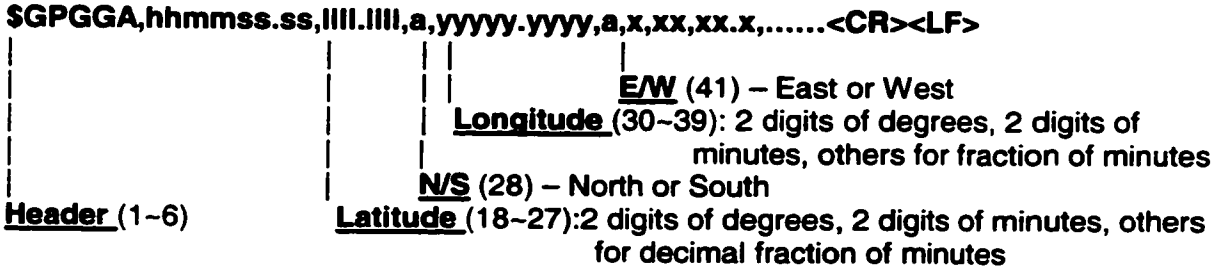


Figure 2-6 GPS Output Message Format

A Java program -- GPSRead.java (see Figure 2-7), using Java Communications API packet [22], acts as an interface between the GPS receiver and its connected serial port, usually at COM1. GPSread.java continuously searches GGA output strings from the GPS unit, strips the longitude/latitude value from the strings, and stores the GPS values in a text file called GPSReading.txt. In GPSReading.txt file, there are two lines, one for longitude and another for latitude. All GPS data is transformed into minutes as its unit, with positive value for East/North and negative value for West/South. The data in the GPSReading.txt file is used to project the current geographical position of the MS in the displayed map. As the GPSReading.txt file is updated synchronizing to the GPS output (twice a second), while the MS is roaming, the user will see himself as a small blue dot moving on the map.

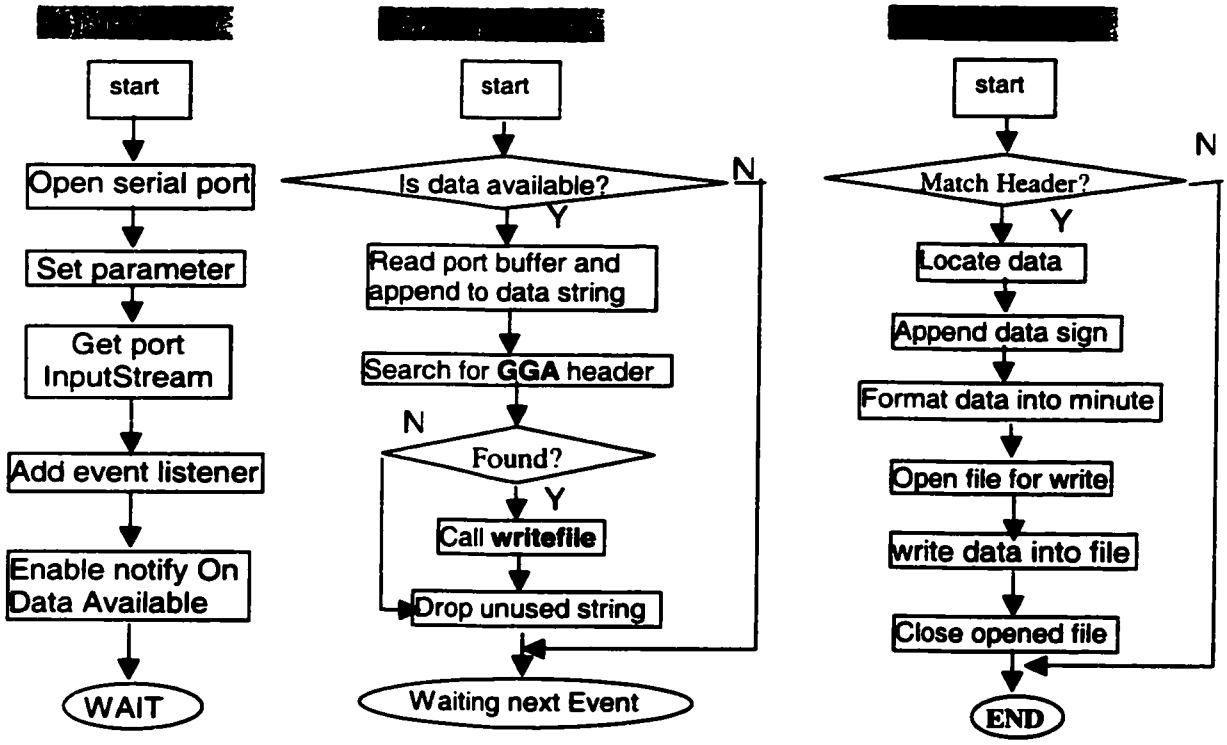


Figure 2-7 Flowchart of GPS reading program

c. *MYMAR MS Directories Structure*

The whole MYMAR programs and files should be installed in the MS (or Client), which has the structure shown in Figure 2-8. The *MYMAR_USER* is the main folder which contains all the sub-directories and the files the MS or the client needs in MYMAR system. It is further divided into 4 subfolders:

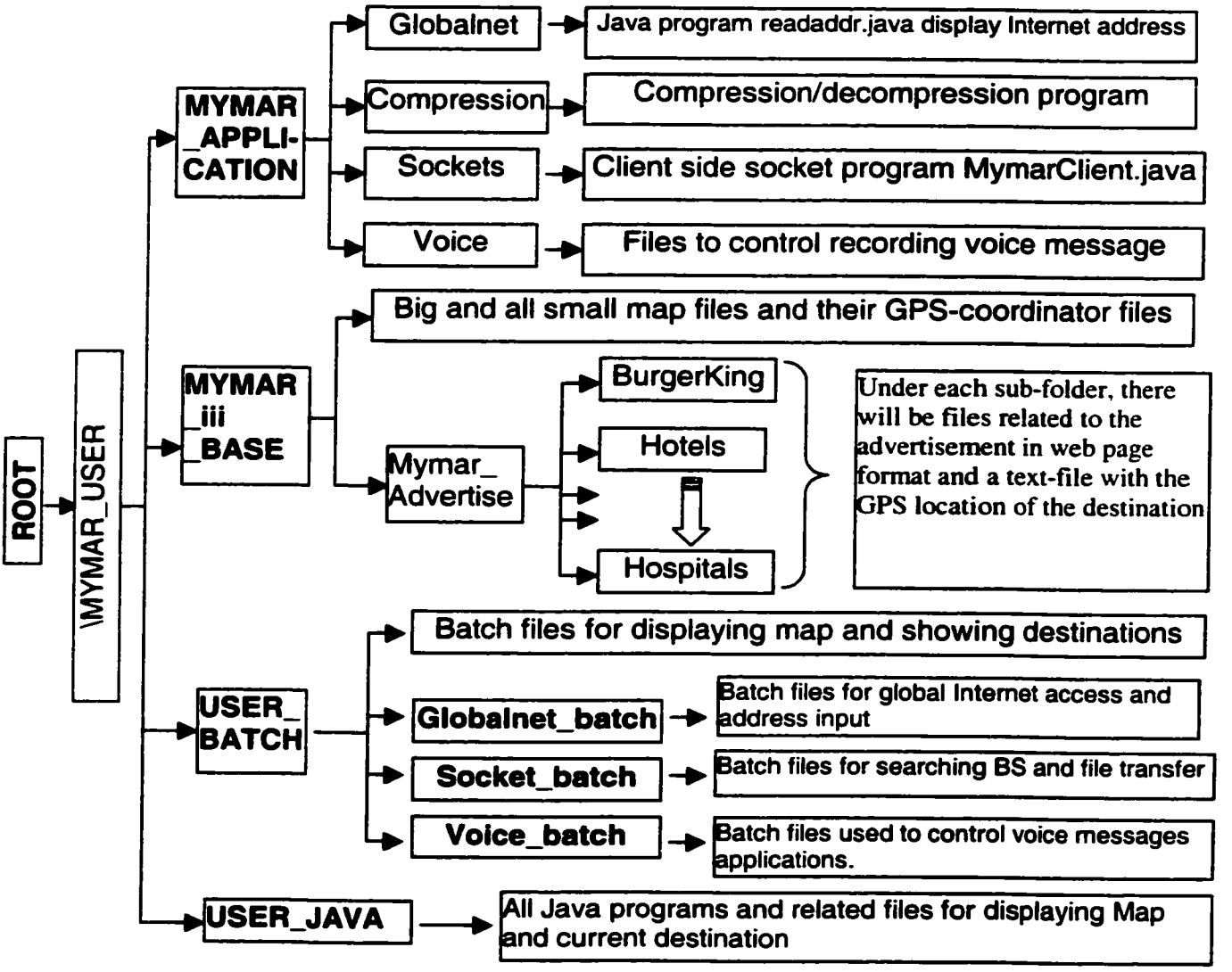


Figure 2-8 MS Directory structure and Files

* *MYMAR_APPLICATION folder*: The programs for each MYMAR application are under its sub-folders. *Globalnet* subfolder: program files ('readaddr.java/class'), used to read and show the Internet web-site address which the user spells using the Realize voice commands. *Sockets* subfolder: program files ('MymarClient.java/class'), used to search the IP-Address of nearby Base-Stations. *Voice* subfolder: voice message recording and playback programs, and their action control programs. *Compression* subfolder: a compression software (ARJ), which is used to de-compress all the compressed data retrieved from the BS.

* *MYMAR-iii_BASE folder*: This folder has the same files and structure as *MYMAR_Base_RegionName* folder in the BS; actually, it is decompressed from the *Mymar.arj* file got from the nearby BS. It contains local information such as map files and map GPS-coordinator files. The *Mymar_Advertise* subfolder under this folder has sub-folders for each advertisement catalogue.

* *USER_BATCH folder*: This is the main folder for the MYMAR system control, which holds all the important batch-files needed for the whole MYMAR applications to work. Some batch files included in this USER_BATCH folder are related to the data updating, map displaying, and advertisements viewing. Under this folder is subfolders containing some batch files for each applications: *Globalnet_batch*, the corresponding batch files used to access the global Internet by the Realize voice commands; *Socket_batch*, the corresponding batch files used to establish communication with the BS; *Voice_batch*, the corresponding batch files used to control voice messages applications.

* *USER_JAVA folder*: This folder has programs and the supporting files, needed to display maps and show the locations of the user and intended destination on the map. Some important files in this folder are: *gpsreading.txt*, containing the current GPS readings obtained from the GPS receiver by *GPSRead.java* program, and used to project the mobile user position on the map while he is traveling; *display.java/class*, displaying the requested map (*map*.gif &map*.txt*) to the user and marking the location of the intended destination (*destination.txt*) and current user position (*gpsreading.txt*).

Except for some minor differences in the business information/location data, almost all BSs belonging to the same MYMAR region are identical. For example, any BSs belonging to MYMAR Montreal region will have the same regional maps and similar business advertisements for Montreal. This means that once the MYMAR Yellow Page data is retrieved, the user at the MS end may not need to repeat the whole data transfer process again as he roams within the same MYMAR network region.

IV. MYMAR DATA Transfer

While a MS is roaming, it may enter any MYMAR WLAN regions. Before trying to communicate with the MYMAR network in the region, the MS must perform a BS search procedure to identify the nearest BS. Following this, the MS can transfer data between itself and the identified BS. In our present implementation, File transfer protocol (FTP) is used to transfer MYMAR data.

a. ***BS Identification***

All MYMAR stations (BSs or MSs) have a same domain name "Wiselab.com", and each BS has a unique IP address specified in its Network "TCP/IP for the LAN Adapter" configuration. However, unlike wired terminals, which remain stationary when operating on the network, one of the important features of WLAN terminals is the freedom of mobility. A MS may enter a region of different BSs, thus a MS cannot be programmed to direct all its requests to a fixed BS as the case for wired networks. This problem was solved by a kind of BS searching algorithm preceding every data transfer between a MS and a BS. Whenever a MS enters a MYMAR network in its current region, the first step is to search for an active and reachable BS within the region. The search mode is done by trying to establish connection one by one with a BS within a certain IP range of that region. If the MS was able to establish a connection with a specific address within a fixed time-out period, the search ends; in case of failure within the time-out period, the search continues to the next address in the range, until a BS is identified or the searing range is exceeded.

Each MYMAR network has a fixed IP address range for BSs in a specified geographical region, see Fig. 2-9, each field of a IP address for BSs is coupled to the geographical location, and the MS must know these ranges and the corresponding geographical regions. Let us take a scenario of an MS entering the Montreal region, so the user will specify the region name "Montreal".

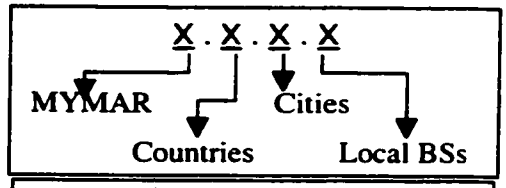


Figure 2-9 IP address range for MYMAR BSs

Using a lookup table in the MS, the communication software will get the BS IP search range for Montreal region. In the present prototype model, the BS search mode is developed using JAVA code and primarily the JAVA java.net socket packet [18]. Programs are developed for both MS sides and BS sides to support two-way communication based on a simple handshaking protocol.

All BSs have a server side program -- MymarMultiServer.java -- running all the time, waiting to accept a connection attempt from a MS TCP client at port 4444. This application listens to incoming client (MS) requests on its main socket and in turn instances a new socket to start a communication handshaking whenever an incoming request received. In this way, the application can handle requests from multiple MS users at the same socket (IP address, port number) in the same time. The assigned socket will establish a connection with the client and begin data communication (handshaking) as illustrated in Figure 2-10.

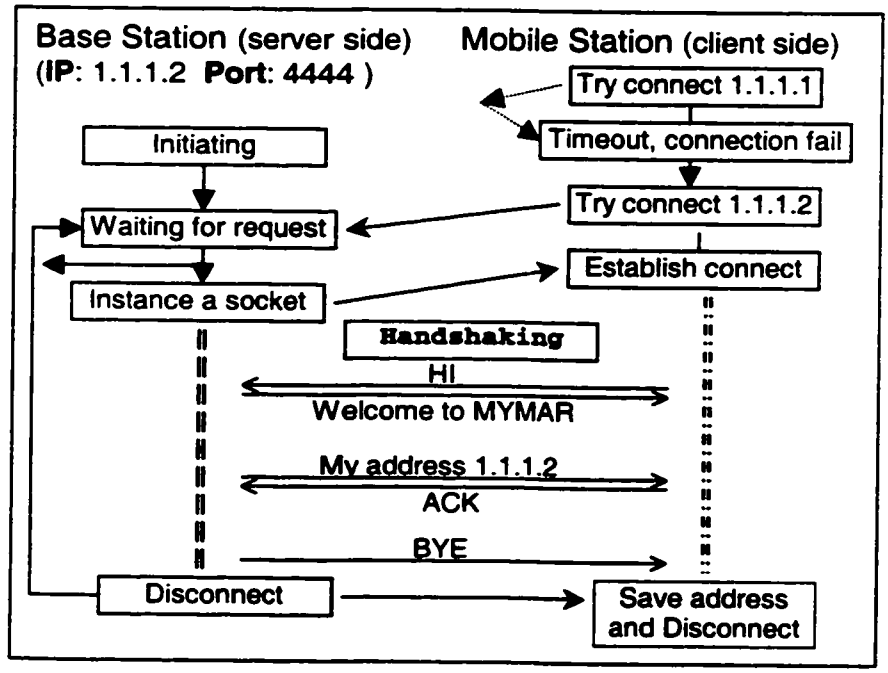


Figure 2-10 Example of BS Identification Procedure

For the MS client side, prior to any data communication with the BS, to create an association with the BS, the MS must initiate a connection with the specific BS IP address and port number. As a MS roams, it does not know exactly which BS (IP addresses) is reachable; only knows IP address range in its current area. Therefore, the MS can try to communicate with IP addresses within the certain range for the region one by one (all using port 4444), until it is able to establish a connection with one BS address within a timeout limit. Thus, finally the MS gets the identification of an accessible BS nearby. The JAVA application program for this task is MymarClient.java, which always runs in the MS before any data communication with the BS.

Once the socket is initialized with a certain BS server, the client program MymarClient.java in MS goes into a dialogue phase to confirm the BS address. In the handshaking procedure shown in Figure 2-10, after the HELLO message, the server (BS) sends its IP address again just to double check the fact that MS is communicating well with the right BS. The client program in MS compares the two IP addresses, the one obtained from the search phase and the one received from the BS. If the two IP addresses are the same, the program writes the IP address after a string "HOST" into a text file named as *IPaddress.txt*. For example, if the BS IP address was 1.1.1.102, the content in the file will be:

HOST 1.1.1.102.

This BS IP address in the file will be used as the host server name in the data transfer between the MS and this BS thereafter.

b. MYMAR File Transfer (FTP)

Once an active BS is identified, the MS can communicate with the BS for purposes such as retrieval of the local MYMAR information, sending voice message, etc. The main information in the MYMAR Base Station contains maps for the current region (Gif files), GPS coordinates for the maps and business destinations (text files), and business advertisements files (HTML files). As all the data is in the form of files (txt, gif, html), File transfer protocol (FTP) is used for data transfer. Whereas a BS is configured to accept FTP requests and to provide information files, refereed to as a FTP server or HOST, a MS initiates requests and retrieves files from a specific server (BS), configured as a FTP client.

In MYMAR application, all the BSs must contain FTP servers. Having an FTP server on the network does not mean everyone has access to such services. Some system administrators may choose to restrict access to their files to a local or selected group of users. When an FTP server has an access restriction, it requires a user identification code and password before allowing access to its files. In the case of no access restriction, the FTP server allows any and every user to access its files; this kind of server is called anonymous FTP server. For our MYMAR application, anonymous FTP server mode is used so that access is provided to any incoming requests and thus facilitating global adaptation of MYMAR application. Although the anonymous FTP server has unrestricted access, it still has a logon procedure. Most anonymous FTP servers are set up to accept "anonymous" as the user identification name.

For FTP server in MYMAR BS, we choose the ArGoSoft FTP Server program from ArGo Software Design [23]. This program is easy to install and simple to setup and it is powerful. This server supports all standard FTP commands and more, such as passive transfers, and file transfer restarts. ArGoSoft FTP Server is a freeware for home or educational use. After installation, we need to set program options (see Figure 2-11) as "DOS" type operating on normal FTP port -- port 21. Furthermore, to become Anonymous Server, a user account named "anonymous" with no password must be created and the permission of read, write, and list right on the folder where MYMAR system resided in the BS must be associated with it. It is also a good idea to check options to let the FTP server start and run automatically in the back ground every time BS is powered on.

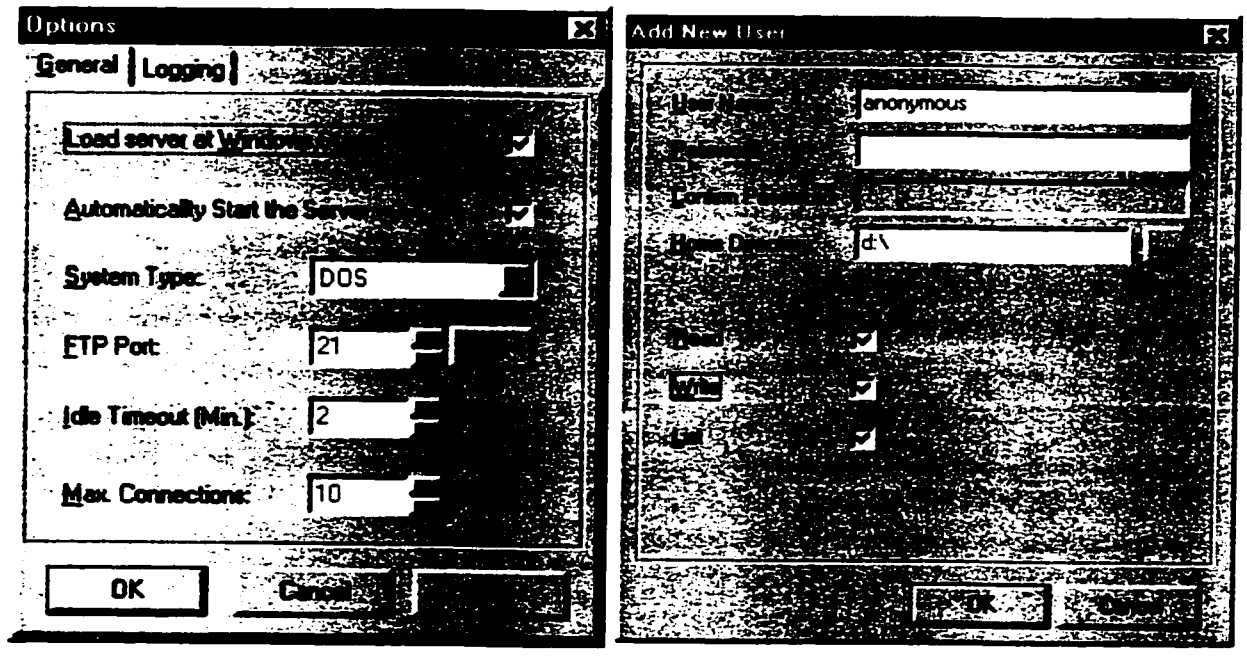


Figure 2-11 FTP Server Setup

With the FTP server program running in the BS, all MYMAR MSs need to be equipped with FTP client software, which orchestrates the file transfer communications between the client (MS) and the server (BS). There are many FTP client programs in the market, we choose CuteFTP [24] for our MYMAR MSs. It is a powerful FTP client program and supports macro scripting, which can be modified to control automatic file transfer sessions. Thus, we can preset file transfer actions using batch files so that data transfer can perform via few hand-free voice commands. This program can be executed from the DOS command line or batch file with a macro/script file in the following format:

[*path*\]**CuteFTP32.exe Macro=**[*path/file name of a saved macro/script file*]

```
HOST 1.1.1.102
Retry 5
Login Anonymous
Connect
Localclearselection
LocalCwd d:\MYMAR-USER\MYMAR-III_BASE
RemoteClearSelection
Remotecwd /MYMAR-III/MYMAR-III_BASE
remoteselect mymar.arj
Download
shutdown
```

Figure 2-12 A example of FTP macro/script file in data transfer

A macro or script file is a text file with an .scr extension describing and controlling FTP file transfer sessions. An example of scripting file is listed in Figure 2-12, which connects to FTP host at 1.1.1.102 as an anonymous user, downloads *mymar.arj* file from *\MYMAR-III\MYMAR-III _BASE* folder in the server (BS), puts the downloaded file into *d:\MYMAR-USER\MYMAR-III_BASE*

folder in the machine (MS). The frequently used macros in the MYMAR application are the followings:

HOST [*host name or IP address*] : the host FTP server name or IP address try to connect to.
RETRY [*number*] : if can't establish connection, how many times to retry.
CONNECT : try to establish connection.
LOGIN [*user name*]: the user name used to login into FTP server.
DOWNLOAD : transfer selected file from remote FTP server to local machine.
UPLOAD : send local selected file to remote machine.
LOCALCWD [*folder name*] : change current working directory of local machine.
REMOTECWD [*folder name*] : change current directory in the remote machine (FTP server).
LOCALSELECT : select file in the local machine.
REMOTESELECT : select file in the remote FTP server.
LOCALCLEARSELECTION : clear selection in the local machine.
REMOTECLEARSELECTION : clear selection in the remote FTP server machine.
SHUTDOWN : shut down the CuteFTP program.

c. File Compression

In MYMAR application, the data to be transferred from a BS to a MS consists of a large number of files - - gif files for maps, one text file per map containing the corner coordinates, text files for business location coordinates and HTML advertisement files. Moreover, most of these files are small size text files (about 50 bytes for a GPS coordinate file). Transfer of so many files using FTP takes quite a long time in each time MYMAR file transfer. Because FTP needs lots of communication overhead including control handshakes, FTP commands, FTP replies, data connections etc. Due to the large number of small-sized files in MYMAR, the communication overhead will occupy most of the data transfer time, so it is very inefficient. Because of the mobility factor of the MS, the data transfer time becomes even more critical. If the data transfer takes too long, the mobile

user may move out of the WLAN of the current BS, there are only 2~3 minutes for continuous transfer. A straightforward solution to above problems is the use of some file compression utility. File compression is used to reduce files in size, and to compress multiple files into a single file so that file transfer has fewer overheads, thus becomes faster and easier. The test results showed that 10 times less could be achieved compare to the transfer time of the normal uncompressed MYMAR yellow page data.

We have used a DOS file compression utility called ARJ. This program allows us to compress/decompress files using DOS commands; thus is compatible with the batch file format for voice commands of Realize voice recognition. We compress all local MYMAR information files in the folder named Mymar_Base_*RegionName* of the BSs, where *RegionName* is the corresponding name for the region or municipal city where the BS resides. Whatever the region it is, all local data are compressed into a same single file named mymar.arj. This compressed file is then copied to the directory MYMAR-III-BASE. Upon MS requests, the contents of the folder MYMAR-III-BASE (currently only mymar.arj file) in the BS are retrieved by the MS FTP program. Once the compressed file is transferred to the MS, it can be decompressed to recover the MYMAR data files.

The command to compress all MYMAR data files in the BS is

arj *a -r* mymar.arj

This command will archive all files of current folder including recursive subdirectories into a file named mymar.arj. The command to decompress the above archived file in the MS is

arj *x -y* mymar.arj

d. Data Transfer

Now, we put all the above-mentioned processes together for a Mobile MYMAR user to retrieve local MYAMR data from a nearby BS. This file retrieving procedure in the MS will include BS searching, FTP file transferring, and file decompressing. In the mean time, the BS is running its server side programs all the time waiting for requests. As all system commands for MYMAR applications are voice activated, the voice command for a MS to begin Yellow Page data transfer is the "Update" voice-phase. The corresponding batch file for this voice command is *ftp.bat*, which is given in Figure 2-13.

```
1.    echo off
2.    java -classpath "\mymar-user\mymar_applications\sockets" MymarClient
3.    if errorlevel -1 goto end:
4.    copy ipaddress.txt+mymar.scr c:\ftp.scr
5.    del D:\MYMAR-USER\Mymar-iii_BASE\finish
6.    start /wait c:\Progra~1\cuteftp\cutftp32 macro=c:\ftp.scr
7.    d:
8.    cd \MYMAR-USER\Mymar-iii_BASE
9.    if not exist finish goto end:
10.   del map*.*
11.   deltree /y mymar~~1
12.   D:\MYMAR-USER\mymar_applications\compression\arj x -y mymar.arj
13.   :end
```

Figure 2-13 MS Data Transfer Batch File for "update" Voice Command

Line 2 runs the client side JAVA application program first for BS searching. Line 3 checks whether the search process finds a BS within the specified IP range. If it cannot find a BS IP address, MYMAR will keep using last transferred information instead of retrieving new data. If the search successfully finds a BS, its IP address is written to a text file (*IPaddress.txt*), which will become the first line of file *ftp.scr* (see Figure 2-12) to indicate FTP host address. Line 4 combines obtained BS address (*IPaddress.txt*) with a script file *mymar.scr* to

form a completed FTP session-control file *ftp.scr*, which downloads MYMAR data *mymar.arj* file from the found BS to the MS. Controlled by formed FTP scripting file *ftp.scr*, Line 6 starts FTP client program to transfer MYMAR data. Then, Line 9 checks whether data transfer is successfully finish. If succeed, after deleted old MYMAR files in Line 10 and 11, Line 12 calls the decompression utility to decompress the file retrieved from the BS to get local MYMAR Yellow Page data (digital maps, coordinate files and advertisement files). At this point, the MS user is ready to use the MYMAR Yellow Page utility as described in detail in the next section.

V. MYMAR Yellow Pages

Upon successful completion of the "Update" command, the MS is ready to use MYMAR Yellow Page application. This Yellow Page application includes map displays, driving directions and business advertisements. The key to this application is that MYMAR Yellow Page data in any BS (then retrieved by MS) must be arranged in a specific predefined directory structure, so MYMAR program can get related data from corresponding files and locations. As in the present prototype model, we have subdivided the businesses into certain catalogues as Restaurants, Hotels, Car Rentals etc., called each subfolder with a specific name for each catalogue containing the related information (see Figure 2-4 and Figure 2-8). For example, the directory name for Restaurant catalogue is "Restaurants" and it contains a text file having longitude/latitude position of all the

restaurants in that region and an HTML file containing Restaurant advertising web page.

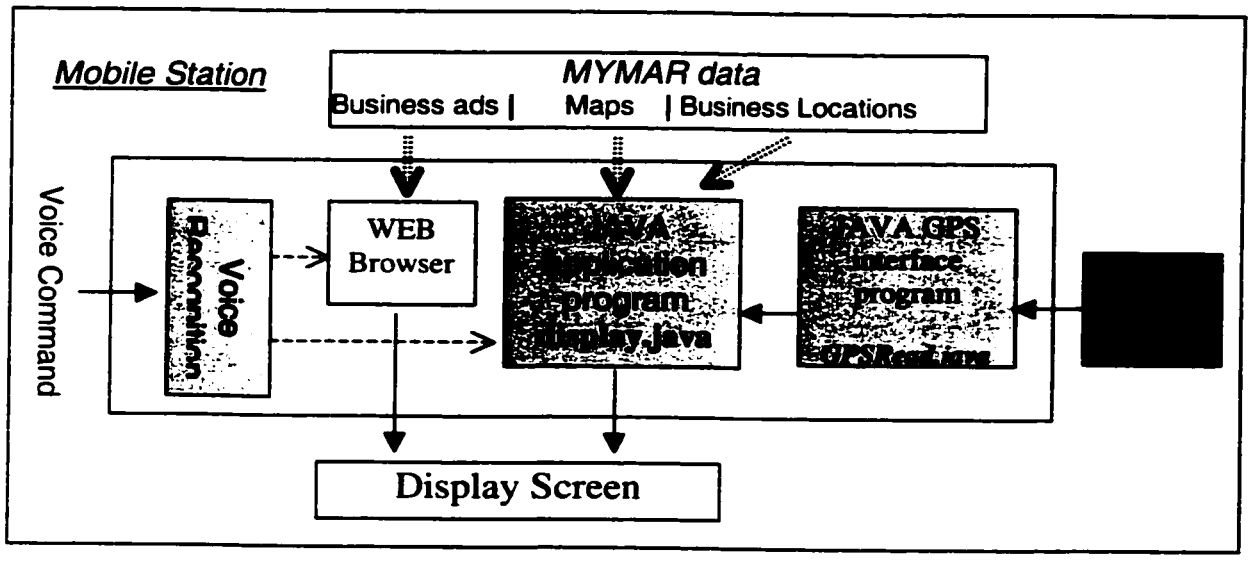
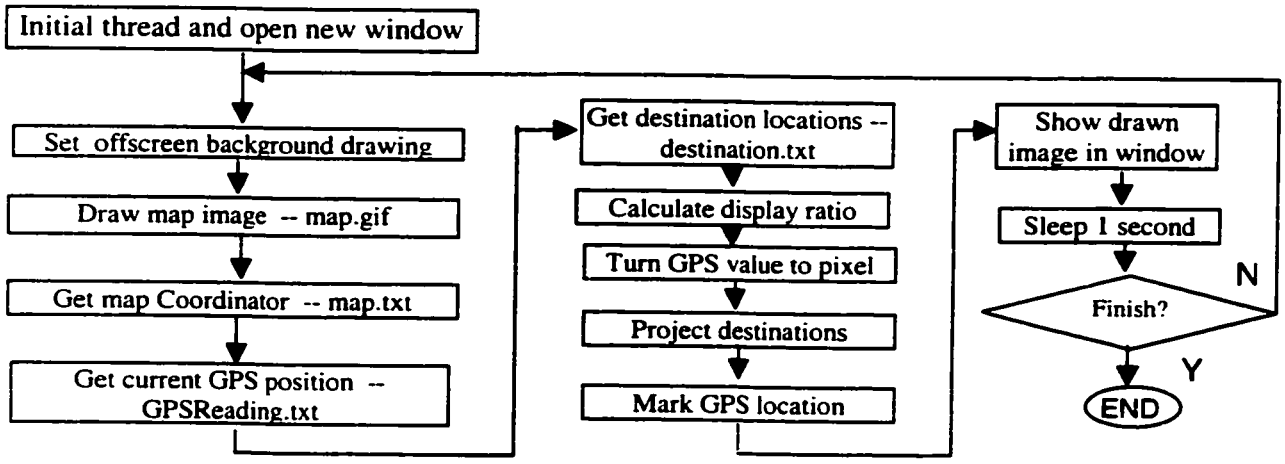


Figure 2-14 MYMAR yellow pages programs

Figure 2-15 Flowchart of display.java program



The operation of MYMAR Yellow Pages is illustrated in Figure 2-14. Whereas the Web browser shows business information, the Java program *display.java* (see Figure 2-15) displays a map contained in the *Map.gif* file in the "USER_JAVA" folder. The *display.java* program uses 'frame' to display the map

and uses 'double buffering' to prevent the map from flickering when image is drawing. The program also marks the destination locations (GPS value in file *destinaton.txt*) and the current user position (GPS reading in file *gpsreading.txt*) on the displayed map, according to their GPS values related to the GPS coordinators of the displayed map (in file *Map.txt*). As the map image and position marks are updated regularly (every second) according to related files, if we copy another map file to overwrite displayed file *Map.gif*, the displayed map will change instantly. If we overwrite the file *destination.txt* with other destinations, for example *hospital.txt* for "Hospitals" GPS locations, the marking destinations will change accordingly. If the MS user has the GPS receiver mounted on his vehicle, as GPS readings are updated in the twice-a-second frequency by Java program *GPSRead.java* (see section 2-III.b), the user will see his position, a blue moving point with intended destinations around him, moving as he is driving on the way.

a. Map Display

The user voice command "where am I" or "big map", will run *GPSRead.java* and *display.java* programs and bring the main regional map on the screen. This big-map shows the general overview of the city in a large scale (see Figure 2-16). To provide zooming capability to street level, each regional map is divided into a number of small regions according to the map scale. The main map has some highlighted flashing numbers corresponding to each sub-region. Each sub-region has a corresponding map file and a corresponding GPS coordinate text files (all maps and their GPS coordinators are obtained from

MapBlast [25]), thus one can zoom in a particular region on the main map by switching to the corresponding map of the highlighted region. Figure 2-16 shows a main map with highlighted regions for Montreal city. The MS user sees himself moving as a blue oval on the display screen; the nearest flashing number gives the identification of the local small area around the MS. In our example, the Montreal region is divided into 36 regions. By the voice command "Map *region number*", for example "map 17", the detailed region map around the user will be magnified. As this map covers a relatively smaller area, detail street information can be viewed.

Figure 2-16 Displayed Big-map of Montreal in MYMAR MS

b. Driving Direction

At the same time of the map display, business locations and current user position are also being projected on the currently displayed map (see Figure 2-16). As the destinations are displayed in 3D red-color square with a "D" in the map, and the user position is a blue moving point as he is roaming, it is easy to find the nearest intended destination and a fast route to reach it. By simply using the corresponding voice command phrase, for example the command phrase " Holiday Inn", the map will be updated with another local destinations. This also gives a detailed driving direction to MS users who roam away from home.

c. Business Advertisement

MYMAR Yellow Page application also includes readable messages for the surrounding businesses, which will give the user (MS) more help and detail information for the right destination he wants to reach. The information, downloaded from the BS (usually advertiser), is designed in web page format; it may contain advertises and promotional offers in an attractive way or important information about businesses such as telephone numbers or street addresses. This detailed information can be viewed in a web browser by using the appropriate voice command phrases like "info *business name*" and they in turn will open the corresponding HTML files from the respective directory.

VI. Voice Messaging

MYMAR includes voice-messaging facility to provide users a non-realtime two-way communication via short voice messages. It works just like a normal voice email service with BSs as servers and MSs as users. Similar to a voice mail server, a MS user has to register to a BS where he can check mails. However, he does not need to locate in the range of his registered BS to send and receive voice mail message (see Fig. 2-17). First, after a user call voice recording application to record his message, the message is always sent to the nearest Base station. Then, the nearest Base station will route the voice message to the registered BS of the receiver. The voice message is stored in the Inbox of the registered BS waiting for the receiver to retrieve. While a user wants to check voice message, the request is routed back to its registered BS. If there are new messages for the user, the messages are transferred to the receiver via routing between its registered BS and its current BS.

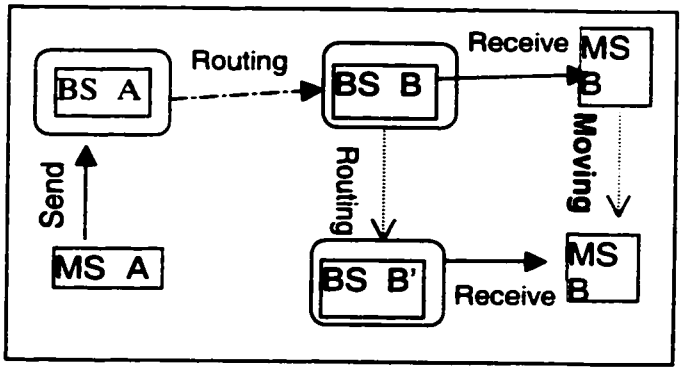


Fig 2-17 Routes of Voice Message

In our experiments, so far, we have only demonstrated voice-messaging facility by sending voice messages from the MS to a BS and receiving messages from the BS to the MS. It can be seen as if the MS is always located near its registered BS or the voice messages have already been routed to the nearest BS

of a moving MS. The above-mentioned FTP programs are used to transfer voice-message files between BSs and MSs. In the complete version of MYMAR, the BS should have the capability to route the arriving voice messages to their destinations using wireless ad-hoc routing algorithms. If implemented high-speed wired interconnection between some MYMAR BSs, the routing will become much faster and MYMAR system will perform much well.

All arriving voice messages are stored in *Voice folder* of the MYMAR BS (see Figure 2-4). This folder has the voice message file (*Voice_Message.wav*) which is sent from the mobile-station to the nearest BS. The file is supposed to be delivered to the registered BS of the receiver (which will be implemented in future). However, at present, we can see this nearest BS as the user's registered BS, so the user can send and retrieve short voice messages to the BS (point to point communication from MS to BS or vice versa).

a. Sound Program

To record or playback voice messages, an application called RecordAll-Pro from Sagebrush Systems Inc. [26] is used. Again, the reason of using this software is its auto-controllability and its DOS compatible features, thus making Voice messaging application co-operate with Realize voice commands. The program format is:

[path]\RecAllPro /l [filename] : To start Recording.

[path]\RecAllPro /p filename : To play a sound file.

It is desirable to make the size of recorded message files as small as possible, because the files are supposed to travel across many WLANs on

wireless channels to reach the end user. Thus, if we record sound file as the compression 8-bit mono format at 8kHz sample rate, the size of one-minute voice message will be less than 100k-byte long, more than 10 times smaller than the normal wave sound format.

b. Control Program

Usually each computer system has only one sound device, so two applications can not use the same device simultaneously. As the MYMAR is a voice-activated application equipped with Realize Voice, the program has to run in the background all the time to accept voice commands. Now any recording software also needs the sound card, which is being used by Realize voice recognition package, hence for the MYMAR voice application to record or to playback a voice message at the MS, it will cause a sound device conflict.

The straightforward solution to this problem is that while recording a voice message, we disable Realize Voice; and after that switch back to Realize Voice for using other voice system commands. Similarly, before playback a voice message, we disable Realize Voice and after that reactivate voice recognition. However, this switching control between the applications must be hand-free and automatic so that the MYMAR user does not need to take an action. As both Realize Voice and the recording software are window-based applications, to control their enabling, we used Microsoft Window C++ programming techniques to simulate the keystrokes to enable/disable the Realize voice recognition program.

Realize Voice package includes a "Stop Listening" mode in which it no longer uses the sound card resources for Listening. Thus, whenever voice-messaging utility is activated, we switch the program into "Stop Listening" mode by activating "Speech" --> "Stop Listening" menu or combining of 'Alt_S' and 't' key strokes. This is done automatically by the use of the program coded in *StopListening.cpp* (see Appendix A). Likewise, The program *StartListening.cpp* (see Appendix A), implementing keyboard commands 'Alt_S' and then 's', brings Realize Voice back into "Start Listening" mode again to enable the users to use voice commands for other MYMAR features.

Additionally, after sound recording is started, there is no way to stop it without manual control. Thus, a MYMAR C++ program *Delay2.cpp* (see Appendix A) implements a fixed duration to stop voice messaging system. Message Recording continues for 2 minutes unless stopped manually using a mouse or keyboard action.

c. Voice Commands

The above-mentioned progresses are associated with three voice commands of MYMAR voice messaging application in the present prototype:

- "*Voice message*": Begin recording a voice message

The corresponding batch file *record.bat* (see Figure 2-18) put all controls we have mentioned together. Line 3 first stops voice recognition to avoid conflict. After start 2 minutes delay counting in Line 7, Line 8 begins voice message recording. After two minutes, the recording stops automatically and Realize Voice will be reactivated in Line 9 to take voice commands again.

```
1.  d:
2.  cd \mymar-user\user_batch\voice_batch
3.  \mymar-user\mymar_applications\voice\stopListening
4.  if not exist voice_message.wav goto conti:
5.  del voice_message.wav
6.  :conti
7.  \mymar-user\mymar_applications\voice\delay2
8.  start /w "c:\program files\recallpro\recallpro" /l d:\mymar
                              _user\user_batch\voice_batch\voice_message.wav
9.  \mymar-user\mymar_applications\voice\startListening
```

Figure 2-18 Batch File for Recording Voice Message

- "*Send Message*": Send recorded voice message to the nearest BS

Once the voice message is recorded, the next step is to send it to the BS. As the message is in the form of a wave sound file, FTP is used to transfer this file from the MS to the BS (Uploading to the FTP server). The principles used are just as the one used for MYMAR data transfer from a BS to a MS.

- "*Receive Message*": Retrieve a voice message from a BS and playback the message

This is a reversed action of the above two steps. It first calls FTP program to download message file *voice_message.wav,* then playbacks the message to the receiver. Similarly, to avoid sound conflict, it needs to stop and then reactivate the voice recognition program.

VII. Global Internet

Because of the 1 Mbps or higher available rate for wireless UL bands, one of the most attractive service for MYMAR users is using this high speed

communication as wireless Global Internet access. This service can be provided if several MSs and its nearby BS are constructed as a small local WLAN (a virtual LAN with IEEE 802.11), with the BS connected to the global Internet and working as a proxy server for the MSs. The proxy server works just like a gateway to the World Wide Web; the BS listens for global access requests from clients (MSs) within the WLAN and forwards these requests to the remote global Internet servers. After reading responses from the external Internet servers, the proxy server delivers those responses back to the internal clients (MSs).

a. Web Proxy Server

A Web proxy server is a specialized HTTP server. The proxy server acts as both a server system and a client system. It is a server when accepting HTTP requests from browsers, and acts as a client system when forwarding the requests to remote servers to retrieve web page documents. Directing their requests to a proxy server, users can simply open their browser or e-mail application, and are automatically connected to the Internet as if they had their own dedicated connection.

The proxy server program in our MYMAR, WinProxy 3.0 [27], offers the latest technology in the Internet connection-sharing software, which allows users to connect multiple computers on a network to the Internet through a single point, using its existing service provider and user account. By installing WinProxy program, the BS is designated as the only computer authorized to make a physical connection to the Internet. WinProxy then acts as the network's intelligent Internet gateway by accepting authorized requests for Internet access

from all other MSs and routing the connection through WinProxy on behalf of the individual users, hence the term proxy. WinProxy runs unattended as a service in the background, automatically establishing connection to global Internet when required and logging off when service is no longer required. To set up the WinProxy server in the MYMAR BS, in "Protocol -->DNS Setup" (see Figure 2-19) enter the ISP's DNS server addresses (can obtained by clicking on "Find My Name Server"), and enter "Wiselab.com" in Domain name for current MYMAR system.

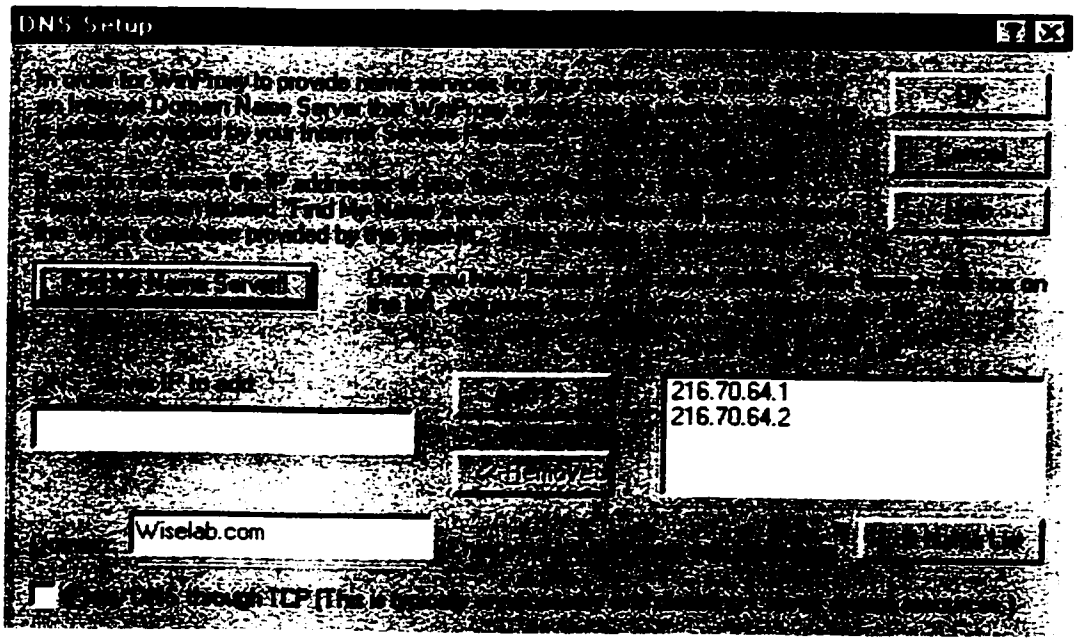


Figure 2-19 BS Proxy Server Setup

For the client (MS) side, ordinary web browsers and settings can be configured to direct global Internet requests to the proxy server. For Global Internet, the MS does not enter a BS search mode and neither it uses the IP address obtained earlier, instead it broadcasts its request and then the nearest BS (with proxy server running) responds and assigns an IP address to the MS for

further identification purposes. This mode of communication is called Dynamic Host Configuration Protocol (DHCP), which provides a means for a central computer to dynamic assign network addresses and information to individual computers as needed. Thus, set "Obtain an IP address automatically" in Network Configuration on "TCP/IP for the LAN Adapter" in the MS to get the IP address automatically.

b. Voice Activated Browsing

Voice activated browsing is one of the attractive features of the present MYMAR prototype model and thus makes internet browsing to be compatible to the rest of MYMAR voice activated command structure. As mentioned earlier, Microsoft Realize voice package is used for voice recognition, system voice commands includes 'Press Tab' to advance between the links, and 'Press Enter' to retrieve a web page corresponding to highlighted link.

To go to a specified web site, a good idea is using "GO TO" voice command to surf predefined 'bookmark' or 'favorites' web pages, otherwise the address of a web site or Uniform Resource Locator (URL) is supposed to be typed in by the keyboard. A hot research area, in the field of voice activated browsing is to get a specific URL address using voice. MYMAR introduces a new and simple approach in this regard. MYMAR users can spell the URL letters one by one to get an address inputted just like typing. This approach overcomes the limitations of the predefined and fix-booked web page accesses. Users can

directly go to any desired web sites, just the way a keyboard works, instead of typing the URL, users will just spell them using voice commands.

To simulate keyboard typing, voice commands for all the alphabets 'a' - 'z', digits '0' - '9' and some frequently used characters in URL's such as '.', ' /', '-' and '_' are defined in MYMAR system. The associated batch files are written in such a manner that whenever they are executed they append the corresponding character into the end of a same text file "*URLaddress.txt*". For example if the voice command for alphabet 'a' is voice 'a', this will run a corresponding batch file "a.bat" which will append file "*URLaddress.txt*" with the letter 'a'. By this way, the whole URL address can be formed finally in the file "*URLaddress.txt*". Once the address is ready, starting an installed web browser with the URL address from file "*URLaddress.txt*" as an argument can retrieve the Web Page.

Figure 2-20 Display URL Address Frame for Global Internet application

To let a user know what he has just spelled, a JAVA program readaddr.java (see Appendix I) runs whenever Global Internet application is initiated, allowing the user to view the contents of" *URLaddress.txt*" file in a large font as he spells the URL address (see Figure 2-20).

The voice commands for Global Internet application in the present prototype are as follows:

"*Internet*": This voice command associates the Global Internet application and runs the *readaddr.java* program, which displays the contents of URLaddress.txt file in a large font. Now the user can begin spelling the intended URL address. As his spelling goes on, letters of the address are appended into file "*URLaddress.txt*" one by one and are displayed on the frame window at the same time.

"*Submit*": When the URL is complete and is ready for the web page request, the *'Submit'* command finishes above address spelling process, and initiates the web browsing. The corresponding batch file runs the default web browser installed on the system, taking the URL address from the file "*URLaddress.txt*" as an argument. If the user is in any of the MYMAR network the nearest DHCP proxy server will respond and get the WebPages for the user.

VIII. Field Tests

The concept of our MYMAR system has been analyzed and designed; related application programs and system control programs are developed in HTML, Java, and C++ language. Simple voice commands of voice recognition technology are integrated into MYMAR system for hand-free operations and controls. Except Window system command, the voice command short-cuts and their associated batch files are developed to access different application of MYMAR (see Table 2-1).

Table 2-1 MYMAR voice command phrases

Voice Phrase	Path where it points to...	Voice Phrase	Path where it points to...
Update	\user_batch\socket_batch\ftp.bat	**Restaurants**	\user_batch\restaurants.bat
Big map	\user_batch\bigmap.bat	**Burgerking**	\user_batch\burgerking.bat
Where am I	\user_batch\map.bat	**Mcdonalds**	\user_batch\mcdonalds.bat
Map1...map40	\user_batch\map1.bat...\map40.bat	**Hospitals**	\user_batch\HOSPITALS.BAT
Zero	\user_batch\globalnet_batch\zero.bat	**Hotels**	\user_batch\hotels.bat
Dash	\user_batch\globalnet_batch\dash.bat	**Police**	\user_batch\police.bat
Slash	\user_batch\globalnet_batch\slash.bat	**Postoffice**	\user_batch\POSTOFFICE.BAT
Dot	\user_batch\globalnet_batch\dot.bat	**Shopping**	\user_batch\shopping.bat
Com	\user_batch\globalnet_batch\com.bat	**Gasstation**	\user_batch\gasstation.bat
Submit	\user_batch\globalnet_batch\end.bat	**Inforestaurants**	\mymar-iii_base\mymar_~1 \restau~1\Restau~1.htm
Internet	\user_batch\globalnet_batch \globalwebpage.bat	**Infoburgerking**	\mymar-iii_base\mymar_~1 \burger~1\Burger~1.htm
a...z	\USER_BATCH\globalnet_batch \a.bat...\z.bat	**Infomac**	\mymar-iii_base\mymar_~1 \mcdona~1\mcdonald.htm
One...nine	\USER_BATCH\globalnet_batch \one.bat...\nine.bat	**Infohospital**	\mymar-iii_base\mymar_~1 \hospital\Hospital.htm
Underscore	\USER_BATCH\globalnet_batch \underscore.bat	**Infohotel**	\mymar-iii_base\mymar_~1 \hotels\Hotels.htm
Voice message	\USER_BATCH\voice_batch \record.bat	**Infopolice**	\mymar-iii_base\mymar_~1 \police\Police.htm
Send message	\USER_BATCH\voice_batch \send_message.bat	**Infopost**	\mymar-iii_base\mymar_~1 \postof~1\Postof~1.htm
Receive message	\USER_BATCH\voice_batch \receive_message.bat	**Infoshop**	\mymar-iii_base\mymar_~1 \shopping\shopping.htm
		Infogas	\mymar-iii_base\mymar_~1 \gassta~1\Gassta~1.htm

MYAMR system is verified both theoretically and practically. Indoors and outdoors testing proved the practicality, usefulness, and versatility of the multi-application multimedia service obtained. Numerous field trials were conducted with the above prototype model of MYMAR in the streets of cities of Montreal, Brossard, St Hubert, and around Loyola Campus of Concordia University located in Quebec, Canada. Most were successful and achieved the expected partial or full results. Excellent communication ranges were obtained in the tests. For example, a reliable link is maintained at 4KM distance with a 5dB car magnet mount small antenna and a 12dB BS transmission antenna placed over Mont Royal in Montreal, a 100-meter height above street level. What a surprise, given

the nature of propagation loss the low power (800 mw) and the LOS characterization of the IEEE 802.11 equipment used. The three main applications of MYMAR namely, MYMAR Yellow Page, short voice messaging, and global web access, all worked at the above range. Numerous field trials were also conducted on straight section of roads in suburban areas of the city of Brossard, Quebec. The above antenna gave a 2KM range only, mainly due to the low elevation of the BS antenna (about 4m with respect to street level where the MS roamed). The worst range obtained was 600m around Loyola campus of Concordia University, Montreal, mainly due to the high foliage, and the presence of higher buildings around the place where the antenna was placed. The details of the field trials are listed in Table 2-2.

Table 2-2 Field Test Results

Location	**Loyola Campus, Concordia University**	**Brossard, Quebec**	**St Hubert, Quebec**	**Mount Royal and Montreal Downtown**
Date conducted	February 2000	March 2000	July 2000	July 2000
Trans. Antenna	12 db unipole			
Antenna Elevation	15.6 m	2.6 m	3.6 m	100 m
Receiving Antenna	5 db magnet mounted on a moving car (speed 50 km/hr) with an elevation of 1.5 m			
Transmitter Power	800 mW transmission power GINA unit with PCMCIA adapters			
Range	600 m radius. The antenna was mounted on a building surrounded by other higher buildings.	2 km at a straight section of the street with fewer trees.	1.2 km due to the curving of the street.	4 km. Intermittent communication was still possible at a distance of 8 km.

We can conclude that our design of the MYMAR system is practical, user-friendly, and hand-free application for wireless devices. It shows a brilliant future of wireless communication. Actually, major car manufacturing companies, such

as GM, Ford, Mercedes etc. are coming with similar products in 2001 but their operation is in the lower bit rate of the cellular or licensed bands. Our MYMAR system approach makes an effective and efficient use of the unlicensed free bands within the interconnected wireless networks. Future research and development in this area can prove to be very useful and can bring about a new revolution in wearable computing devices. Although the complexity of such devices will not be too great, their utility has the potential to really speed up many ordinary repetitive tasks that we do by hand today.

Chapter Three Zone Routing Protocol (ZRP)

I. Introduction to ZRP

The unguided wireless medium and surrounding physical environment significantly decline and distort radio transmission, result in a limited transmission range and unreliability of wireless communication of a WLAN segment. While mobile users free roam around nested WLAN segments, existing IEEE 802.11 based WLAN standard has minimal capabilities when it comes providing wireless connectivity among neighboring WLANs. In wireless client/server networks, a central station always reaches all mobile nodes in its service area, traffic between BSSs usually is routed by APs via infrastructure network, but this is not always the case in the ad hoc networks. In ad hoc network mode, which is self-organizing and does not rely on any base stations and existing infrastructure, an underlying network routing through multihop to distant nodes must be provided for wireless applications in this distributed network. However, unlike wired network routing protocols, for large-scale wireless ad hoc network, multihopping, mobility, and large node size (each node is a hop) make the development of efficient routing protocols a major challenge. On one hand, the effectiveness of a routing protocol increases as network topology information becomes a more detailed and up-to-date. On the other hand, in a WLAN, the topology may change quite often, requiring large and frequent exchanges of data among network

nodes. In this chapter, a routing protocol, Zone Routing Protocol (ZRP), is analyzed and implemented. The evaluation results showed that a simple and efficient ad hoc routing protocol is possible.

a. Routing in ad hoc network

IEEE 802.11, however, only defines the ad hoc network standard for wireless nodes to communicate between each other if they are within the same BSS. Similar to wired networks, routing topologies in ad hoc networks must be provided for wireless connectivity among neighboring wireless LAN segments. However, due to the features of wireless radio communication and the free mobility of the nodes, conventional wired routing protocols may not suitable for WLANs. In recent years, various style routing protocols for wireless ad hoc network have been proposed, which can be classified into following four kinds of schemes: flooding, proactive routing, on-demand routing, and location assisting routing [28]

Flooding

The least efficient routing protocol is flooding: the source simply broadcasts a packet to its neighbor nodes, neighbor nodes re-broadcast received packet to their neighbors; in this way, the packet is flooded all over the network, expecting that at least one copy of the packet can reach the intended destination. No routes are computed on demand and minimal or no priori knowledge of network structure is assumed. Frequently, scoping (time-to-living) may be used

to limit the hops of the flooding range.

Proactive Routing

The modification of some traditional wired proactive routing, including distance vector and link state, can be applied in wireless networks. For example, destination sequence distance vector routing (DSDV) [29] based on shortest-path algorithm. Each node maintains a routing table with entries for all nodes in the network. Due to frequent changing mobile topology, sequence number or time stamp may be used to keep routing table up-to-date. This type algorithm need periodically and globally update messages; thus, only performs well on small population ad hoc network.

To reduce update message, [30] purposed a routing protocol using "fish eye" algorithm. The Fisheye State Routing (FSR) maintains accurate distance and path information about the immediate neighborhood of a node, with progressively less detail as the distance increase. Updates from further away will be sent out less frequently compare to local updates, this is accomplished by using different exchange periods for different entries in the table. The imprecise routing path to a distant destination is compensated by the fact that the routing information becomes more accurate as the packets get closer to the destination.

To reduce the size of routing information table and processing overhead, hierarchical techniques can be applied by organizing nearby nodes into clusters and combining nearby clusters into super-clusters to build up a large hierarchy [30]. Using this approach, one or more nodes can act as cluster-heads, representing a router for all traffic to or from the cluster. Only the cluster-head

nodes keep and update the routing information about low-level cluster nodes. Nodes of other clusters only need to know how to reach the cluster. As nodes move, clusters may split or merge, altering cluster membership. This approach hides all the small details in clusters, so routing information is relatively stable as nodes typically remain within a cluster.

On Demand Routing

In contrast to above precomputed routing protocols, on demand routing scheme invoke a route discovery only on necessary. On-demand routing is based on a query/reply approach; an example is the Dynamic Source Routing (DSR) protocol [31]. The source node with data to send but no route to destination floods a query into network. The intended destination eventually receives the query and responds with a reply message sent back to the source. The intermediate nodes append their own addresses into the query/reply message to accumulate the route. Upon received the reply message, the source accumulates a whole path to the destination. In addition, the intermediate nodes may also learn their own paths to the source (or destination) from the query (or reply) message.

There are no periodic updates and huge table storage are required in the on-demand routing. The on-demand routing discovery may result in less control traffic, especially when innovative route maintenance mechanisms are applied. However, this reactive scheme will introduce a larger initial latency, and the reliance on packet flooding may still lead to considerable control traffic, especially when the scale of network becomes large or when the nodes move very

frequently.

Location Assisting Routing

Another type scheme is location assisting routing, route computation is assisted by the knowledge of geographical location of the destination, usually provided by GPS devices. Location-Aided Routing (LAR) [32] is a location based on-demand routing protocol. Using location information obtained from GPS, LAR restricts the flooded area of route request messages to a circular area where the destination may be located, or LAR guides route request messages forwarded only to the directions that the distance will be reduced.

Distance Routing Effect Algorithm for Mobility (DREAM) [33] is another proactive location based routing protocol. In the routing table, coordinates of each node are recorded instead of route vector. Each node in the network periodically exchanges control messages to inform all other nodes of its location. The source selects a set of neighbors that are located in the direction to the destination as the next hop and multicasts the data packet to these nodes. Once those nodes receive the packet, they select their own list of possible next hops and forward the data with the updated list unless the packet has reached the destination.

[34] presents a GPS based location assisting routing incorporating within the MAC layers. It limits the flooding (broadcast) packet only forwarded by neighbor nodes that are within a specified routing angle from the source to the destination. In case all or some of the node's GPS locations are not known, the flooding can be further reduced by a random forwarding possibility among

neighbors.

b. Zone Routing Protocol (ZRP)

As mentioned above, the traditional routing protocols can be classified as either proactive or reactive schemes. Proactive protocols attempt to continuously determine the network connectivity so that a route is available whenever a packet needs to be transferred. The families of distance vector and link state protocols are examples of proactive schemes. The advantage of the proactive schemes is that when a route is needed, there is little delay until the route is determined. However, pure proactive schemes continuously use a large portion of the network capacity to keep the routing information current. Proactive protocols are widely used in small population, high density and heavy-traffic networks.

In contrast, reactive protocols invoke a route determination procedure only on demand. That is, only when a route is needed, some sort of global searching procedure is employed. The classical flood search algorithms are reactive protocols. However, in reactive protocols, the route information may not be available at the time a route request is received, the delay to determine a route can be quite significant. Furthermore, the global search procedure of the pure reactive protocols requires excessive control traffics. Reactive protocols are more appropriate for large population, light traffic networks.

A recent proposal of Zone Routing Protocol (ZRP) [35] is a hybrid reactive/proactive routing protocol, which provides efficient and fast discovery of

routes by integrating the two completely different classes of traditional routing protocols. On one hand, it limits the scope of the proactive procedure only to the node's local neighborhood. As the local routing information is referred quite often in the operation, ZRP protocol reduces the waste associated with routing update traffic of the purely proactive schemes to the limited number of zone members. On the other hand, the search throughout the network, although it is global, is performed by efficiently querying selected nodes in the network, as opposed to flooding queries all over the network.

Each node has its routing zone. A routing zone of radius ρ is defined the nodes whose minimum distance in hops from the node in question is at most ρ hops. As an example of a routing zone of radius two hops is illustrated in Fig 3-1. Dash line shows the region of each zone: zone A for central node A, etc. The nodes whose minimum distance to the node in question is exactly equal to zone radius are called peripheral nodes (or border nodes); all other nodes (minimum distance is 1 to ρ-1) are interior nodes. For instance, Node J is border nodes of zone A; also, it is an interior node of zone B. Node H is a border node of both zone A and zone B. Noted that if two zones define the same radius ρ, while Node H is a border node of zone A, Node A is one of border nodes of central node H.

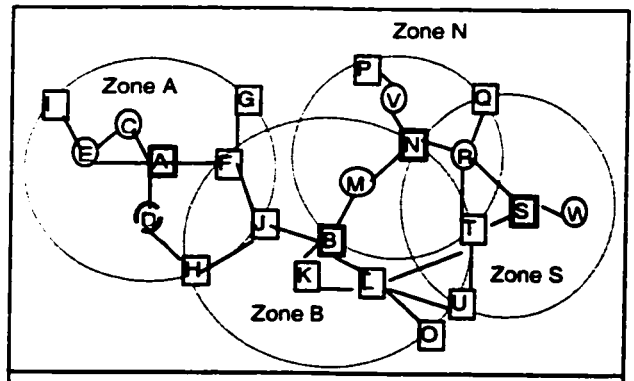


Fig 3-1 Routing Zone

Each node maintains routing information only to those nodes that are within its own routing zone. Therefore, the routing updates are only propagated locally, the amount of update traffic required to maintain a routing zone does not depend on total number of network nodes instead of a small number of zone members. A node learns its zone through some sort of proactive schemes, which we refer to as the intra-zone routing protocol (IARP). Each node keeps the route information to all nodes within its zone in a local routing table. Because each node maintains its own routing zone, the zones of neighboring nodes are heavily overlapped. In this implementation, we use a modified version of the distance vector algorithm for IARP.

The IARP maintains routes only for those nodes that are within the coverage of a routing zone, which is relatively small compared to the size of the network. Thus, most destinations lie outside a node's routing zone, and the desired routing information cannot be immediately provided by the IARP. The interzone routing protocol (IERP) is responsible for reactively discovering routes to destinations located beyond a node's routing zone. If the destination is not within the source's local routing zone, the source broadcasts a route request (query) to its all border nodes. Once its border nodes receive the request, every one checks whether the destination is within its local zone. If so, a route reply is sent back to the source indicating the route to the destination. If not, the border nodes, also central nodes of their own zone, forward the query to their own border nodes, which in turn execute the same flooding like procedure unless a preset zone limit is reach.

As an example of this route discovery procedure, suppose the source node S needs to send a packet to the destination D in Fig 3-1. To find a route to the destination, node S first checks whether D is within its local routing zone. If so, S knows how to reach D from its IARP routing table. However, D is not local to zone S in this example, so node S bordercasts a query to its border nodes Q, N and U. Now, after verifying that D is not in its own zone, each one of three nodes forwards the query to its border nodes. In this procedure, one of the border nodes - - node B forwards the query to node H. When H recognizes that destination D is within its local zone, a reply is sent back to the source S. If we accumulate this query trail, it is easy to find one of the routing paths from source S to destination D is: S--N--B--H--D.

c. Implementation of ZRP Protocol

We implement the ZRP protocol as an additional layer between the Network Layer and Data Link Layer (MAC Layer). Or we also can see it as an upper sublayer part of the MAC layer. The ZRP layer includes two sublayers itself (see Fig 3-2), *intrazone routing protocol* (IARP) sublayer and *interzone routing protocol* (IERP) sublayer. Whenever the higher layer will send a packet to destination, it invokes ZRP IARP sublayer first. IARP will handle all packets transferring within local zone. If IARP found the destination

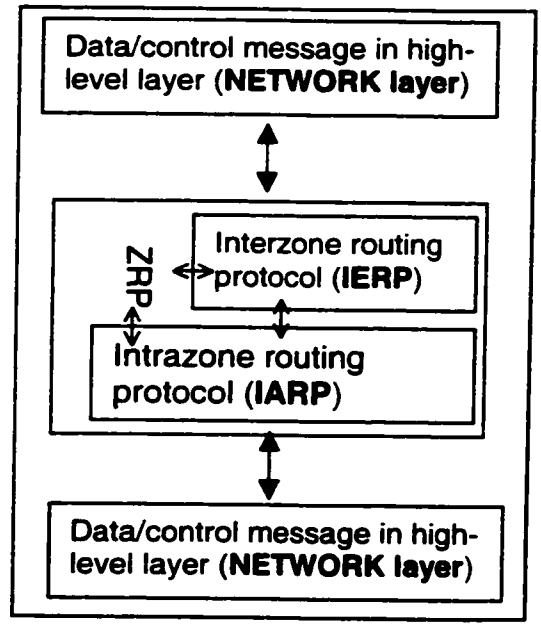


Fig 3-2 ZRP Layer

cannot reach in the local zone, it then invokes IERP sublayer to control the packet transformation across zones.

While IARP handles all local zone packets transfer and keeps local routing information up-to-date, IERP controls query and reply procedures to discover a route path to the destination outside of the local zone and handles packets transferred between zones. To simplify protocol design, we treat IERP sublayer as an upper layer of IARP sublayer. An IERP packet transferring in its route path from a border node to the next border node is using IARP protocol within each local zone. That is, the source node IERP layer forms an IERP packet and then using IARP, to transfer the packet to the first node in the route path, which is a border node within local zone of the source node. After the first border node receives the IERP packet and gets the next node address in IERP route path, it forwards the IERP packet to this next border node using IARP local zone routing information, treating the IERP packet as an IARP data packet. The same procedure repeats in each border node in the route path, until this IERP packet reaches the destination, which is the last node entry in the route path. All of interior nodes within each local zone forward the IERP packet just as an ordinary local data packet using IARP protocol and local zone routing information. As such, interzone packets transfer from one zone to next zone and lastly reach the destination in the network.

Only IARP has the interface to the MAC layer below it, which includes send packet and receive packet sessions. The maximum packet size for 802.11 protocol in the MAC layer is 2312 bytes, as our ZRP does not implement packet

fragmentation and assembly, we limit data packet size for the ZRP is 0~2047 bytes. Usually after adding the ZRP header (IARP and/or IERP), the size for data is below the MAC layer 2312 bytes limitation (16 bytes for IARP header and 20 bytes for IERP header). Occasionally, if the IERP routing path length is longer than (2312 – 16 – 20 – 2047) = 229 bytes (57 nodes), that is, transferring over 56 zones using IERP, the higher layer must lower the packet size limitation.

II. Intrazone Routing Protocol (IARP)

A. IARP Routing Algorithm

IARP can use any traditional proactive routing algorithm like distance vector and link state protocols. The most popular routing algorithm used nowadays is the *Routing Informaiotn Protocol* (RIP) - - shortest distance vector algorithm [36][37]. We modify the RIP standard to fit our case in IARP and to accommodate special features in the mobile wireless communication. Periodically, each node broadcasts a routing update message, which contains its own information and all routing information he knows. The neighbor nodes receive the update message, extract routing information from the message, and update or add the up-to-date routing information accordingly with the sender of update message becoming next hop to reach destinations. Then soon, these neighbors would further broadcast the routing information they just learned. Like

this, destination information will be propagated from one node to others, until it reaches its border nodes in its zone.

Intrazone Routing Table

Each node keeps a list of routing information in an intrazone routing table (RT table) for all destination nodes in the local zone. Each entry in the table identifies the route information to a destination node; one destination node has only one entry in the table of a zone, which can reach the destination locally. The routing information format used in routing table is defined in Table 3-1. *Node address* is the destination node address; *Next node* is the next node address in the route to the destination (if directly reachable, it is the destination address); *Distance* is the distance to the destination measured in hops (nodes), 4-bit for a maximum distance (zone radius) of 15; *Life time* is the time for a route information entry supposed to expire unless it is renewed before this expiring time, 12-bit for a maximum of 4095 time units. Each entry needs 10 bytes in the memory, so supposed one zone of 100 nodes, need just a little more than 1.2k-byte memory to store routing information.

Table 3-1 Intrazone Routing Table Format

Size (bits): 32	32	4(16)	12(4.096s)
Node address	Next node	Dist-ance	Life Time
...	...	...	...
Node address	Next node	Dist-ance	Life Time

Because of the mobility of the wireless network, a node may shutdown or move out of zone, we add this *Life time* field to limit the validity of each route entry, also introduce *timestamp* in the routing update message. Just like HELLO message in a normal wired network, each node broadcasts an update message in a fixed time interval. The update message includes the next supposed update time count in *timestamp* field (5~10 times longer than the update interval) to indicate how long this entry is valid. As the computer timing is not synchronized among each node, this time to live count plus the different local clock becomes the expiry clock (*Life time)* of the routing entry in the receiving nodes. When this routing information is forwarded from one node to another, this time to live count continues to associate with this entry, and the remaining time to live is recalculated (deducted) into the *timestamp* field in update message. In addition, the biggest *timestamp,* the longest remaining time to live count, means the most recent update. If one node moves out of the previous zone, or it is powered off or failed, as no further update message from this node will be received, *Life time* will time out, then the entry to this destination and all routes passed through this node will be dropped from the routing table. Moreover, each node can set a different count according to its mobility and traffic loads. For example, a fixed-place base station can have big *time to live*, and update less frequently.

Send routing update message

Initially, upon power on, a node's local routing table is empty. Periodically, each node broadcasts all routing information in its own RT table, even though the RT table is empty. This procedure is called route updating. We define the format

of the update message as in Table 3-2. The message header contains *source address*, *number of routing entries* which will be send in the message, and *timestamp* which indicates the initial value of valid time to live count for this source routing information. After the header, the message lists the destination information that can be reach by this node. The destination information come from the RT table in this source node; each only contains the destination *node address*, the minimum *distance* if route through this source, and *timestamp* that is the remaining time to live for the entry. If the routing table has too many entries, they cannot fit into one packet (2047 octets, 340 entries), we need to divide the updating into several messages, but each with the same source information.

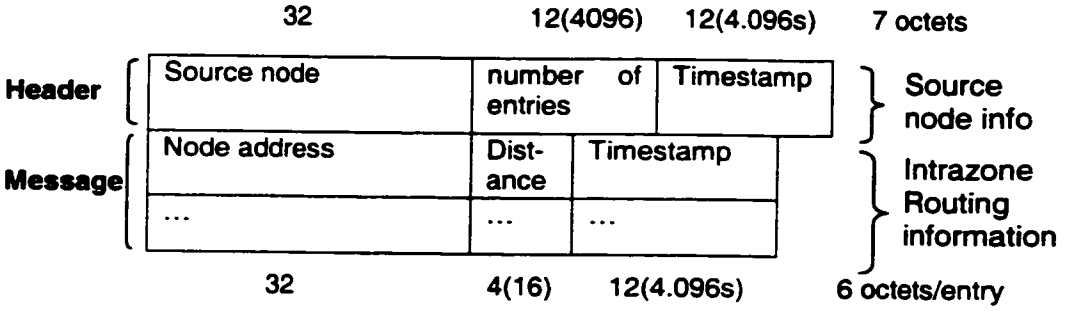
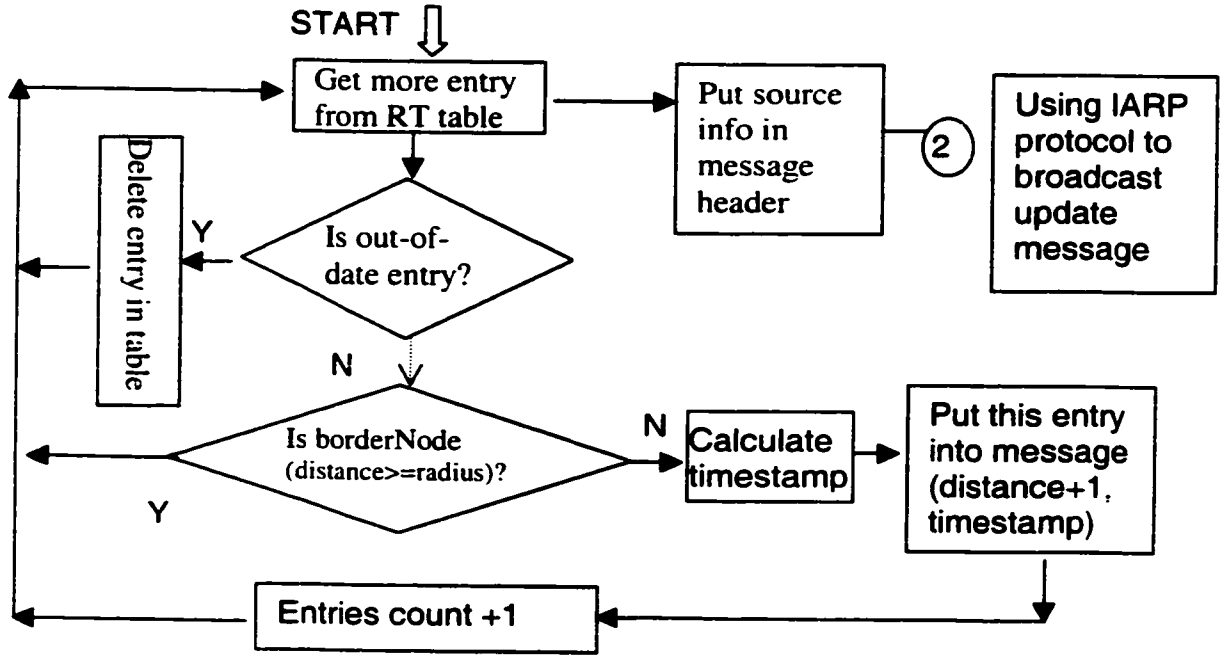


Table 3-2 IARP Update Message Format

The flowchart Figure 3-3 shows the procedure how to form update message (see program IARP::sendUpdate()). For each entry in local routing table, if the entry is out-of-date, i.e. *Life time* is expired, the entry does not need to be sent; this entry will be deleted from RT table. Then, *distance* (hops count) is increased by one in the update message to show the distance if the receiver will reach the destination through this node. As the distance to border nodes of the update source increases one by receivers, these border nodes will not be in the local zone range of the receivers anymore (distance > zone radius). Thus, the

update source does not need to send entries of its border nodes; only needs to send the update message about its interior nodes (distance < zone radius). In addition, the source node needs to recalculate the remaining time to live count from *Life time* in RT table, and carry it in the *timestamp* field with each entry. After getting all entries from the RT table, add the update message header, which contains the source information and the total entry count. This update message will be broadcasted using IARP protocol.

Figure 3-3 **IARP Create Update Message -- -- IARP::sendUpdate()**



Receive Update Message

All nodes within the power range of the updating source will receive the update message, and their distances to the source will be count as one. The procedure of processing received update messages (see program IARP::processUpdate()) is shown as Figure 3-4. First, it extracts the source

information from the message header. Then, it updates or adds the routing entry to the source in its RT table. As the source is directly reachable, *next node* in the entry should be the source address and *distance* is one hop. Furthermore, this source address will become *next node* in the path if routing information to the destination is updated according to the entry in this received update message. Then, extracts each entry from the update message, and updates/adds the entry in the RT table accordingly. However, the update message may contain a routing information about this receiver itself, the entry must be discarded otherwise a routing loop may occur.

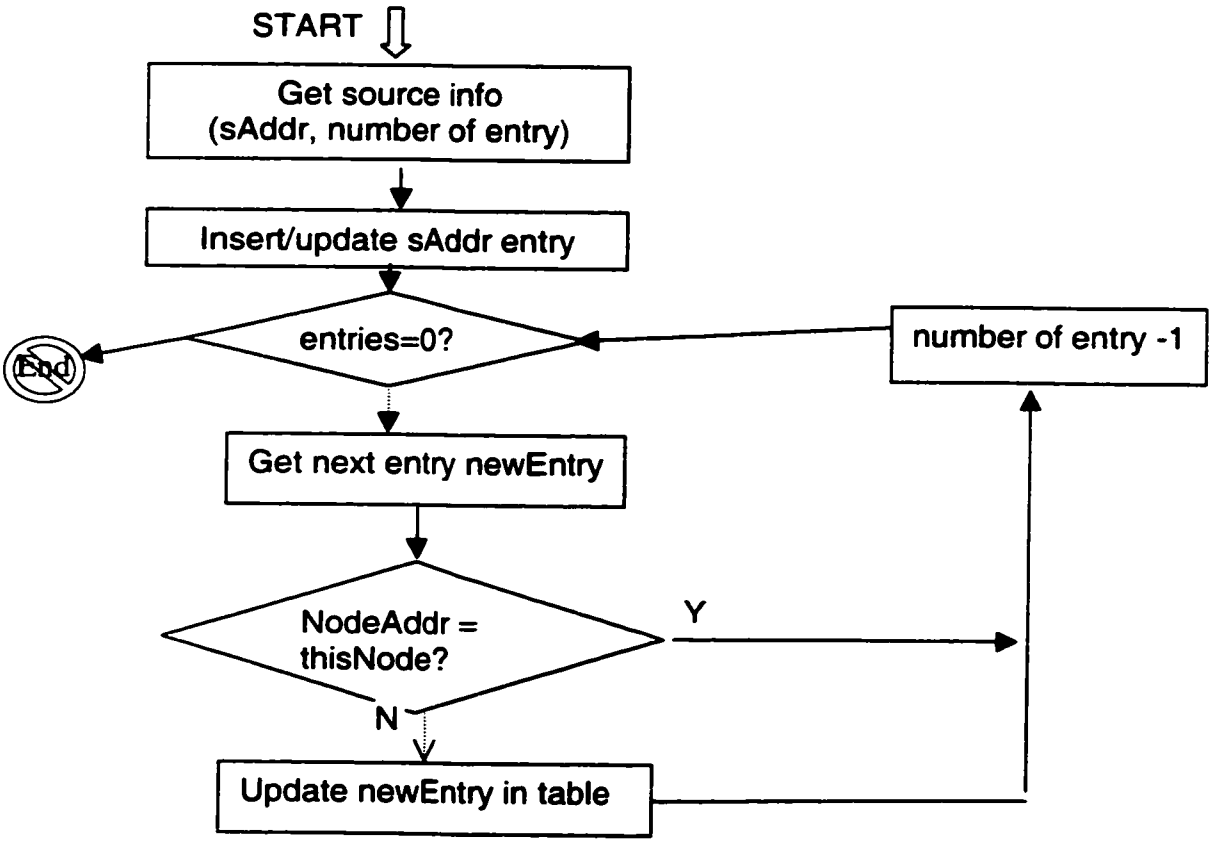


Figure 3-4 Receive update message -- -- IARP::processUpdate()

Update Routing Table

The RT table updating algorithm is shown as in Figure 3-5 (see program IARP::updateTable()). First, the node searches for the destination address of the new entry in the table. If not found, the new routing information should be added into the table. If there is routing information in regard to this destination already in RT table, the node needs to replace that entry in the table while the received routing is the newest entry (longer *timestamp,* also means less propagation) or while it has the shortest distance to the destination (the increment of the distance has already been done while forming the update message).

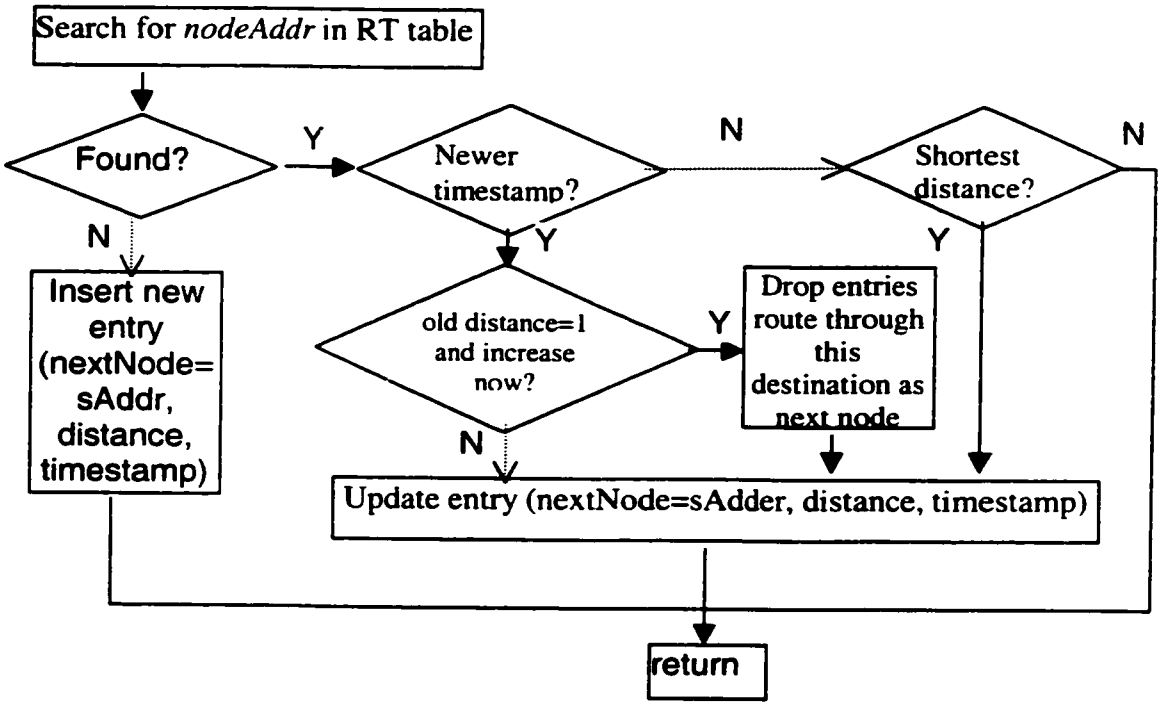
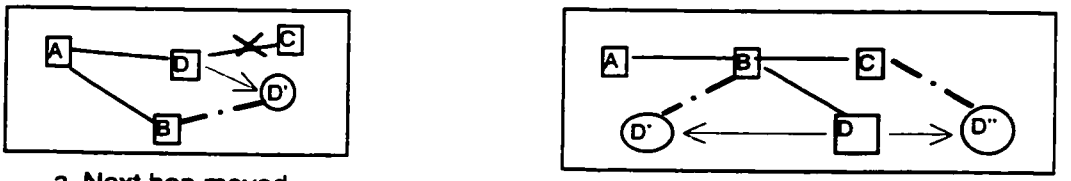


Figure 3-5 **Routing Table Update -- -- IARP::updateTable()**

A problem of this shortest-distance like routing algorithm is that if a node move outward, the distance in update message is greater than old routing

information, so the routing entry will not be updated according to shortest distance algorithm. However, as our modified algorithm contains *timestamp* information, the above problem is overcome while the entry will be first updated according to a newer timestamp. In a special case, see Figure 3-6a, supposed an original neighbor D moves outward from a receiver A (D->D'), this original neighbor may work as next node in the routes to some destinations like Node C. As routing information to this original neighbor updated (update message from Node B shows that distance to D increase from 1 to greater than 1), Node D cannot work as next node any more. Therefore, those routes entries originally pass through Node D are not valid now, so must be dropped.



a. Next hop moved

b. Dynamic Routing Correction

Figure 3-6 Mobility of nodes in the IARP routing

The disadvantage of the distance-vector algorithm is long propagation delay in the newer routing information updated from one node to others. As nodes move, the relative positions of nodes change frequently in any directions, the new routing information may not have been updated in the source while it send out a packet. We may shorten the update interval to let the new routing information be forwarded quicker, but this also increases the network traffic. Fortunately, IARP routing policy can dynamically correct the routing information in the middle way of the packet forwarding. As one node only knows the next node address in the path to reach the destination, all the next nodes in the path

will search the destination again in its local RT table as a data packet forwards through the path, this procedure will redirect the packet to the destination. As an example showed in Figure 3-6b, Source A send out a packet to Destination D using its current routing information, so the next node B will forward the packet. If the destination D moves to D', it is still directly reachable by Node B. If D moves to D", the new routing information in B will direct the packet through next node C. Even though when a packet is sent out of the source, the destination is moving out of one's local zone, the packet can still be forwarded to the destination by one of the border nodes. This routing algorithm has the big advantage in wireless network; it is compatible to the mobility of the mobile wireless communication.

B. IARP Packet deliver

As described above, IARP is a sublayer between higher layer and MAC layer, all higher layer packets and IERP packets are higher layer data packet for IARP, and the local routing information update message is the only IARP control packet. Each IARP packet is forwarded one next hop to another next hop until it reaches its destination. The routing information about next hop is obtained from each intermediate node's own routing table, which is maintained by above mentioned routing algorithm.

IARP header format

An IARP packet must include packet information and routing information in the packet header to let other nodes know how to process the packet (ignore,

forward, etc.). We define the IARP header format as in Table 3-3, which is a fixed-size header with 16 bytes long. Besides the *source address* and *destination address*, it needs only *next node address* field to specify *next hop* of the route path for a packet to deliver to the destination. The 11-bit *payload* can send the maximum data length is 2047 bytes. The 2-bit *T* field defines the receiver protocol type or message type, which is either control/error message, or IARP update message, or data packet (from IERP or higher layer). The 3-bit *P* field can specify eight level priorities. The user of IARP can specify the priority level of the data packet. Usually control massage and IARP update message have higher priority.

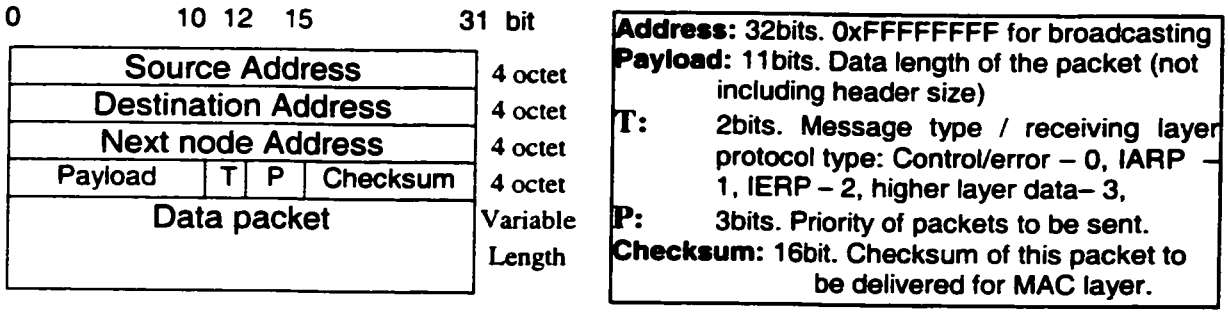


Table 3-3 IARP header format

Send IARP Packet

When IARP has a packet to send, it adds 16-byte IARP header to become an IARP packet according to the packet information (source, destination, payload, protocol type, and priority). The flowchart in Figure 3-7 illustrates the procedure of IARP sending data packet (see detail in program IARP::processSend()). There are three sources of data packet for IARP: Higher layer, IERP packet, and IARP

routing information update message. After adding IARP header, calls IARP::sendToNext() to deliver the packet to the destination via next node in the route. IARP::sendToNext() will get routing information *next node address* from its local routing RT table. After putting *next node address* into IARP header, the packet will be sent by MAC layer to the next node, which is the first forwarding receiver in the route to the destination.

Figure 3-7 **IARP sending packet -- IARP::processSend()**

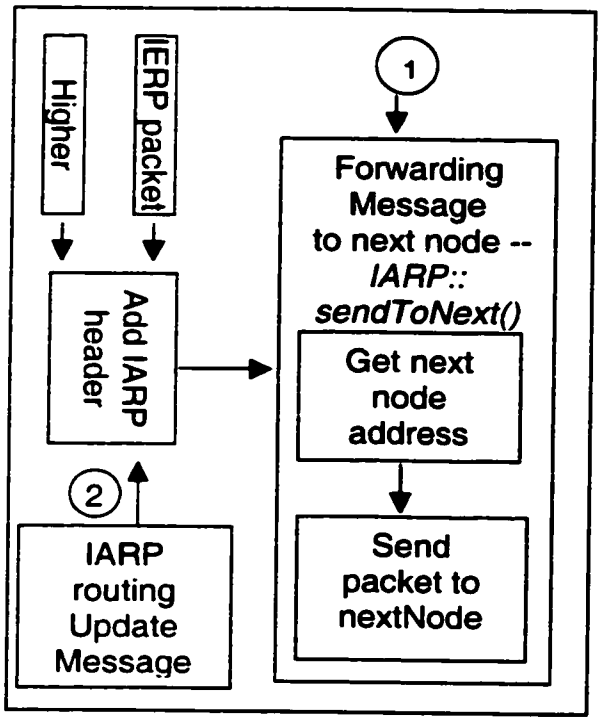
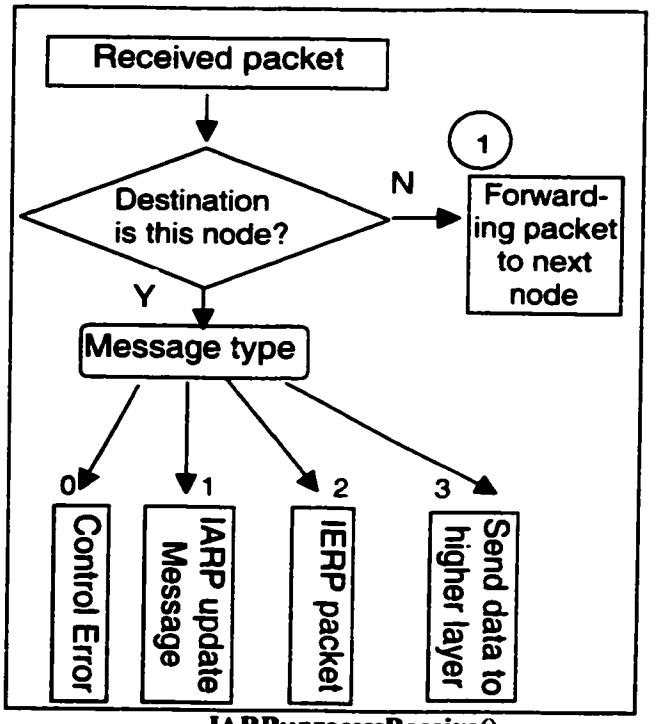


Figure 3-8 **IARP receiving packet -- --**



IARP::processReceive()

Receive IARP Packet

When a node receives an IARP packet from the MAC layer, it follows the procedure in the flowchart Figure 3-8 (detail in the program

IARP::processReceive()). First, the program extracts information about the packet from its IARP header. Then checks if this node (receiver) is the destination of the packet. If the packet has not reached the destination yet (this node is an intermediate node in the route to the destination), calls IARP::sendToNext() to continue forwarding the packet toward the destination. As mentioned above, IARP::sendToNext() will get current routing information to the destination from the local routing table of this node. After changing *next node address* in the IARP header, the packet will be sent by MAC layer to the new next node in the route. The new next node will follow the same procedure to forward the packet. Until the receiving node finds that the packet destination is its own (reach the destination), the program will deliver the data packet to the receive layer according to the message type after discarding IARP header.

III. Interzone Routing Protocol (IERP)

A. IERP Route Discovery Algorithm

Whereas the IARP maintains route information only for those nodes that are within the coverage of a node's local routing zone, the Interzone Routing Protocol (IERP) is responsible for reactively discovering routes to destinations located beyond a node's routing zone. As described in the introduction section, the principle of IERP is to transfer data packets, including IERP control packets, from a source node's local zone to its border node's local zone, to next border's local zone, repeating this procedure in every border node until packets reach the

destination. Thus, the route path from the source to the destination consists of a sequence of border nodes' addresses. However, IARP protocol is used while packets are forwarded from one node to its border node within its own zone, so we can treat IERP as an upper sublayer of IARP.

IERP Routing Query

When a destination is not within the source local zone, the source node need to find a route path to the destination, the source IERP broadcasts a query message to begin a global searching for the destination. This query packet is spread out from one zone to neighboring zones as all received nodes forward the query to their border nodes. All received nodes follow the same procedure: when the query packet reaches border nodes of each zone, it checks if the destination is within local zone; If not, it changes the last query *sender* in query message and works as a new query *sender* to spread out the query packet again. Finally, some border nodes will find that the destination is within local zone and send a reply message back to the source. While we accumulate the query forwarded trail (the last query *senders)*, a route path can obtain.

Because the source may request another route discovery to the same destination before the first query get reply, or normally, there are several packets needed to send to the same destination after a route discovery. In both case, it is not necessary to discover the same routing path again in a short period. Thus, every node's IERP keeps a small size Query Table for information about current querying routes and recent discovered route paths. Define Query Table entry

format as (see Table 3-4): ***destination address***, the queried destination or the route destination; ***route pointer***, pointing to the memory where the route path is stored; ***route length***, indicating the length of route path in number of nodes; and ***timeout***, the timer count to wait for the query reply or the expired timer of a valid route. It needs 12 bytes memory for each query entry and additional memory to store a routing path. When a query is sent out, one entry in the table records the query destination and sets the maximum *timeout* for the source to wait for the reply message. At this time, the route is not discovered yet, *route pointer* is empty and *route length* is zero. While a query gets its reply message, and accumulates the route path accordingly, *route pointer* will point to the location stored the route path, *route length* has the length of the route path, and *timeout* is reset to indicate the expiry time of this route path.

Table 3-4 IERP Query Table Format

Size:	32	32	16	16 Bits
Destination Address	Route Pointer	Route Length	Time-out	
~	~	~	~	

Early Detection

However, problems can arise in the global searching procedure. For instance, an example showed in Fig 3-9 , because the routing zone is heavily overlapping, while the query will be spread out by zone A to its neighbor zones, the query message is border-cast by central node A to all its border nodes B, C, F and G. These border nodes follow the same procedure, so node A will receive 4 loop-back queries from B, C, F and G. Similarly, node F will receive two same

query messages from A and B. To solve this loop-back and duplicate querying problem, we need introduce appropriate mechanisms which have the abilities to detect repeat queries and to terminate the queries early if necessary.

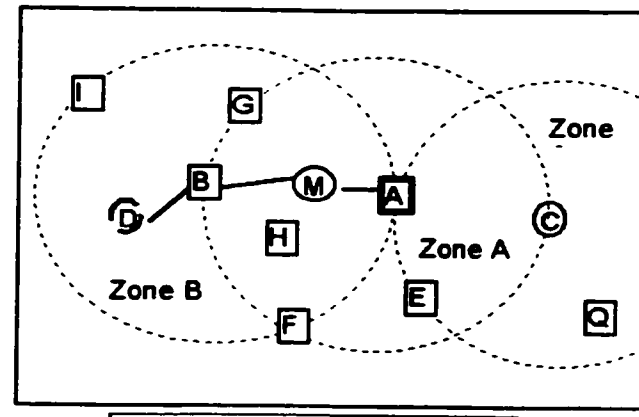


Fig 3-9 Early Detection

Because an IERP query is forwarded to border nodes via IARP by interior nodes, if we extend the early detection mechanisms to these interior nodes, the repeated queries can be terminated even earlier. For the example in Fig 3-9, while an interior node M previously receives a query passed along from A to B, it will instantly terminate duplicate query massages from B and G, which is supposed to loop back to node A. Actually, the early detection will terminate a query if it re-enters a previous queried area, in this way, the query is steered outward from the source only. While every node now has the ability of early detection, we can send query messages by broadcasting to all nodes instead of border-casting to border nodes. Nevertheless, interior nodes just record query messages for the early detection purpose, only border nodes need to process query messages.

For the early detection purpose, each node has to keep the information about all previous received queries. After received a query packet, a node record the query information into an Early Detected Query Table (ET table) (see Table 3-5), which records the query *source address*, *destination address*, query *sender*,

the last border node where the query is forwarded from, and *timeout*, indicating the expiry time for the entry to stay in the ET table. The ET table only records information about the first received query, which always represents the shortest accumulated delay the query has traveled. Because a source may have queried several destinations and a destination may be queried by many sources at the same time, each entry in the ET table is identical with *source address* and *destination address* fields together.

Table 3-5 IERP Early Detected Query Table Format 14 octets/entry

Size (bit): 32	32	32	16
Source address	Destination address	Sender	Timeout
~	~	~	~

Sending IERP Query

In conclusion, Figure 3-10 illustrate the procedure for a source to discover a route path to an out-zone destination (details see program IERP::query()). Whenever a destination is not in one's local routing zone, an IERP route discovery begins. First, the program searches the query table to see whether the destination query exists. If the query exists and a route path is known, a "ready to send packets" message is return; if the query exists but no route path is available, the procedure must wait until the reply message of previous query is returned or is timeout; otherwise, a route to the destination must be queried. IERP will form a query message as per the IERP protocol defined in next section. After recording the query information in Query table and ET table, and setting the timer of the maximum waiting timeout for getting the query reply, the query message is

broadcasted via the IARP protocol. The returned reply message will include an accumulated route path to the destination. This route path will be stored in memory, and the state of this query in Query table is updated (route information and new timeout). A single query may return multiple route path replies, we select the best route based on the shortest accumulate delay, which is simply the first returned reply path. If the query waiting time count is timeout, which means the destination cannot reach or delay is too long, a "destination cannot be found" error message is produced after the query entry in Query table is erased.

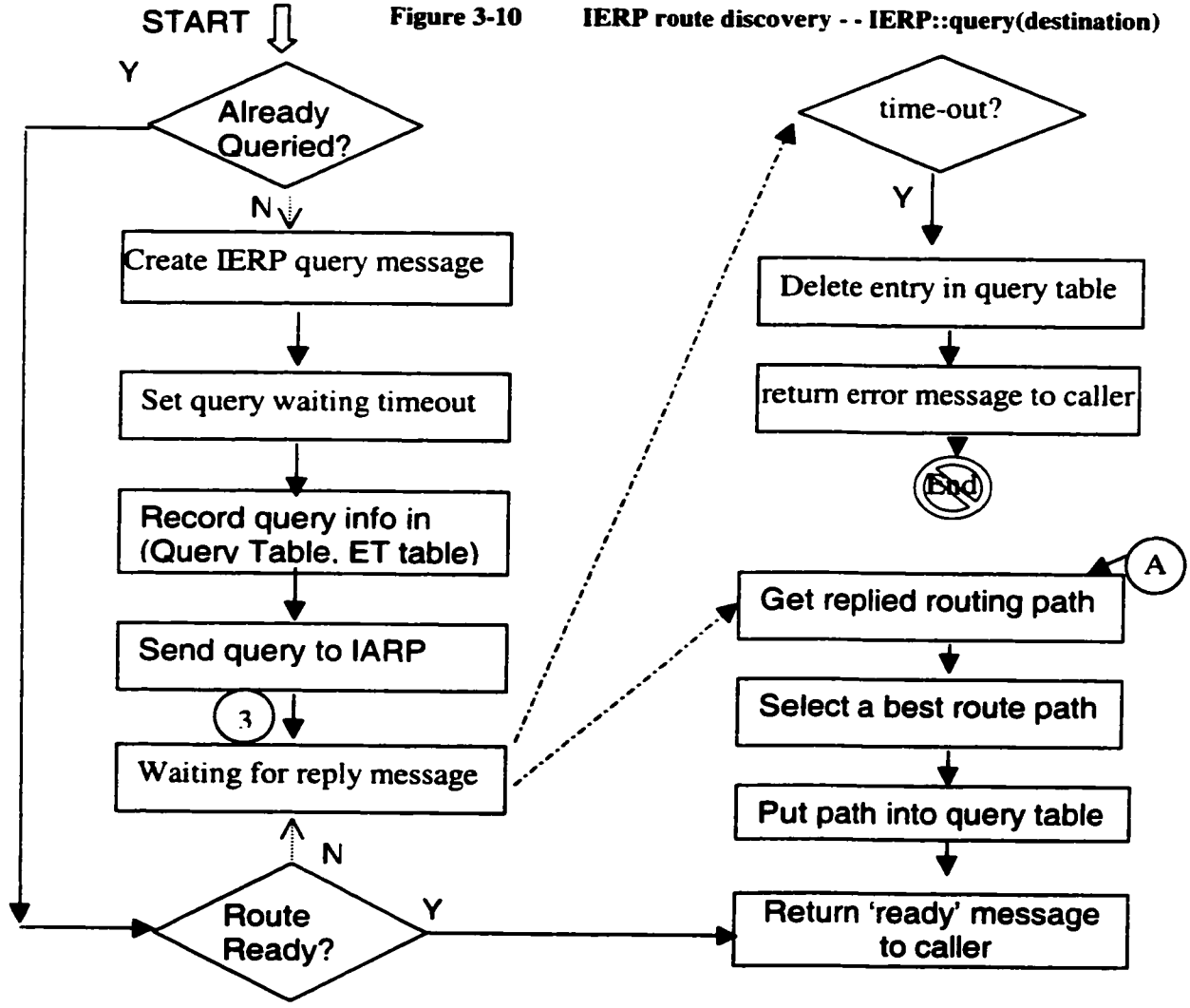


Figure 3-10 IERP route discovery - - IERP::query(destination)

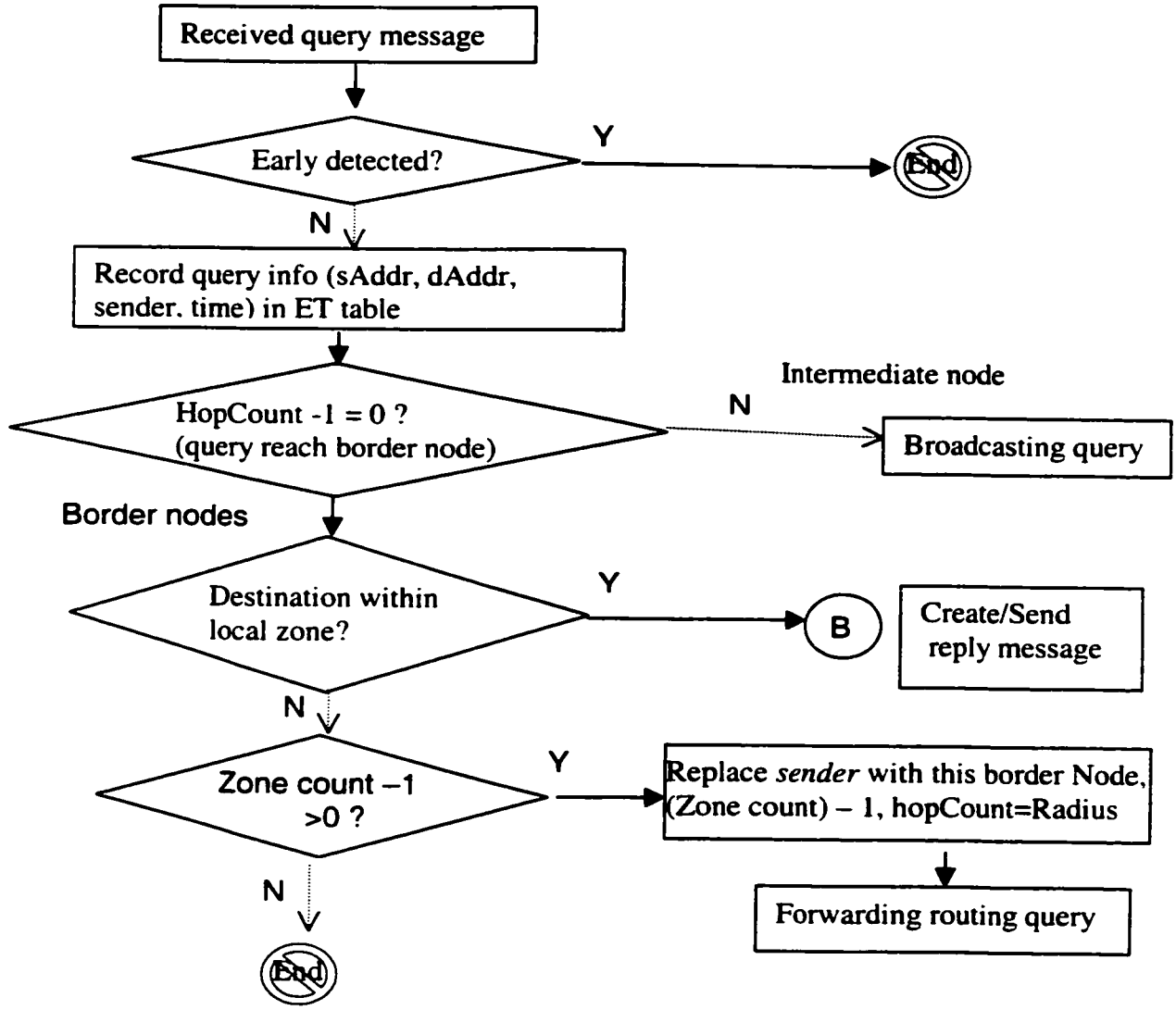


Figure 3-11 IERP Process Query -- IERP::processQuery()

When a node receives a query message, it will enter a query processing procedure as shown in Figure 3-11 (see the program IERP::processQuery()). First, the node needs to check the query information in its ET table, if the query has been early detected, early termination will occur to end the query forwarding. Otherwise, if the query is first time received, it will record the query information in ET table. Next, after decreasing the *hop count* by one, the node checks if the query reaches a border node. If not, this interior node will re-broadcast the query

message to its neighboring nodes. If the query reaches a border node, the border node will check if the destination is reachable within its local zone. If the border node cannot find the destination in its local routing table (unreachable), it resets *hop count* to the zone *radius* and overwrites *sender* with this border node address, then continue to broadcast this query. If the destination is within the local zone of this border node, it will invoke the procedure to create and to send a reply message back to the querying source. Certainly, a query cannot spread out all network forever, it must be dropped when it reaches a zone count limit setting in the source query message.

Route Accumulation

Tracking a query trailed from the source to the destination, we can find a route path. In the basic route accumulation procedure, a node (border node) appends its ID (address) to a received query packet and continues to query its neighboring zones. Until a border node finds that the destination is within its local zone, the accumulated sequence of IDs specifies a route between the querying source and the queried destination. Returning this accumulated sequence of border nodes to the source, the source will complete a route path discovery. In addition, by reversing the accumulated route, a path is provided to return the reply packet of the query to the source. The disadvantage of this procedure is that while accumulating, the query packet gets bigger and bigger, and as the query packet spreads outward, most of the querying directions cannot reach the

destination, these useless large query packets will create heavy network traffic, even though the query is at last terminated when time to live count reach zero.

In the ET table of the queried nodes, we already have a route about the query stored in the form of query *sender* (previous border node), which points a 'next hop' route back to the queried source eventually. Thus, we can reverse the above route accumulation procedure in the reply session instead of in the query session. In the query session, rather than append border node addresses to accumulate route information, every border node can overwrite *sender* field in the query packet with its own ID before forwarding the query, to specify that it knows route information back to the source. A node receiving the query will know that the source can be reached through the most recently queried *sender* and store this route information in its ET table. The route accumulation procedure begins while the reply packet is forwarded back to the queried source via the reverse route the query packet travels. By appending the query *sender* in the reply packet, a route path will be accumulated during this route reply phase. In this scheme, all query packets have the same format and a small size, resulting in less IERP control traffic.

IERP Reply

If a border node finds the destination is within its local zone, it invokes the procedure to create a reply message and to send it back to the querying source (see Figure 3-12 and the program IERP::reply())). The reply message will begin to accumulate the route path, but at this time, there are only two node addresses:

this border node and the destination. The reply message will be sent back to the querying source routing through the previous query *sender*, which is recorded in the ET table. This sending procedure is just an ordinary IARP data packet forwarding within one's local zone.

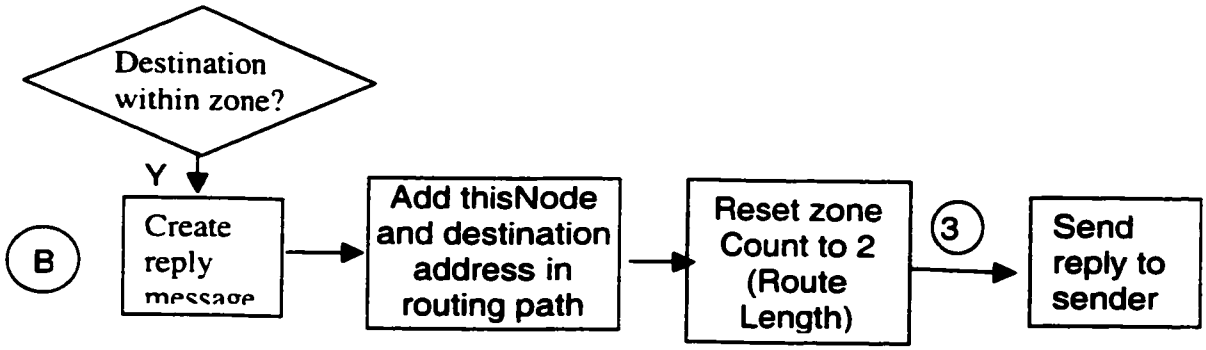
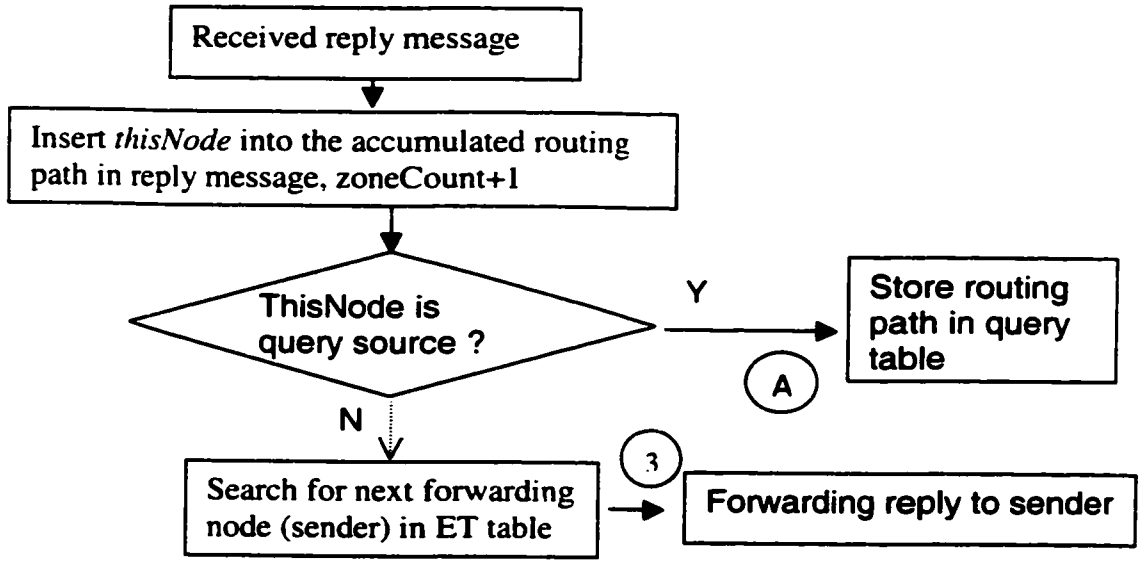


Figure 3-12 IERP Create Reply -- IERP::reply()

Figure 3-13 IERP Accumulating Routing Path: - - IERP::processReply()



After receiving the reply packet, a border node will process the reply message and will accumulate the route path as the procedure shown in Figure 3-13. First, to accumulate a route path, the node inserts its address at the

beginning of the accumulated route sequence contained in the reply message and increases the route length by one. So far at this node, the reply message has accumulated the route from this node to the destination. If the reply has already returned back to the querying source, then the route in reply message has accumulated the whole route path from the source to the destination. If the reply has not reached the querying source yet, the node searches its ET table to get previous query *sender*, and further sends this accumulated reply message to that node, which will follow the same process. Until the reply is routed back to the querying source, a route discovery procedure is finished after the accumulated route information is stored in query table.

B. IERP Packet deliver

There are four types of the IERP packets: data packets from higher layer, query/reply routing discovery packets, and error control packets. We define a unified IERP message format as Table 3-6: *query message* has only header (always 16 bytes), *reply message* has the accumulating route path, *data message* has data and the route path to the destination, and *error/control message* has the data field for error information and a route path back to the source.

Each fields are explained in the following, noted that some fields might have different meanings in specific message types:

Table 3-6 IERP packet format

Unified IERP format:

32	32	32	7	11	2	8	4
Source	Destination	Forwarding sender	Zone count	Total length	Msg type	Header length	Hop count
Routing path + data packet (optional and various length)							

Query message format should be:

Querying source	Destination	Last query sender	Zone limit	length = 16	**Query**	header =4	Hop count

Reply message format should be:

Querying source	Destination	Next border node	Route length	Total length	**Reply**	Header length	Un-used
Having Accumulated Routing Path (various length)							

Data message format should be:

Querying source	destination	Next border node	Path pointer	Total length	**Data**	Header length	Un-used
Routing Path (variable length)							
Data packet (variable size)							

Error/control message format should be:

Querying source	destination	Next border node	Path pointer	Total length	**Error**	Header length	Un-used
Routing Path (variable length)							
Error message (variable size)							

- **Msg type**: 2-bit Message type. 00 - - error/control, 01 - - query message
  10 - - reply message, 11 - - data packet
- **Forwarding sender**: 32-bit address. For query, it is the previously queried border node address. A node receiving this query would know that the source could be reach through this sender. For other message types, it is the next receiving border node.
- **Zone count**: 7 bits (Maximum value 127). For IERP query, it is time to live (TTL) of the query, i.e., the maximum limit number of zones to query. For the reply message, it is the length for the sequence of the accumulating route path (number of border node). For the data message, it is the pointer of next border node in the included routing path.
- **Total length**: 11 bits, Max 2047 octets. The total length of the IERP packet in octets (including header, route and data).
- **Header length**: 8 bits. The length of header in word, including the length of routing path. Maximum value 2^8-1=255 words. So, except 16 octets (4 words) for the fixed

header, the remaining value indicates the length of the included route path, maximum length of 251 nodes in the routing path (251*4=1004 octets).

- **Hop Count**: 4 bits (Max radius 15 hops). Node count used to determine if IERP query reach border node of a zone. Preset to zone radius.

IERP layer will create different message types as request: forming *query message* to begin discovering a route to a destination, forming *reply message* and sending it back to the querying source while the queried destination is found, and forming *data message* when an out-zone data packet needs deliver. While the use of *query message* and *reply message* is already described clearly in last section, the procedure of forming a data message is shown in Figure 3-14 (see program IERP::processSend()). After the query and reply procedures, knowing the route path to the destination, the source IERP will begin to send data packets to that destination. After checking the route path to the destination is known from the query table, IERP data message is created. As only the source has accumulated a completed route path, the route information must be included in each IERP data packet. Then, the IERP data packet is send to the second entry (1st entry is the source) in the route via IARP layer.

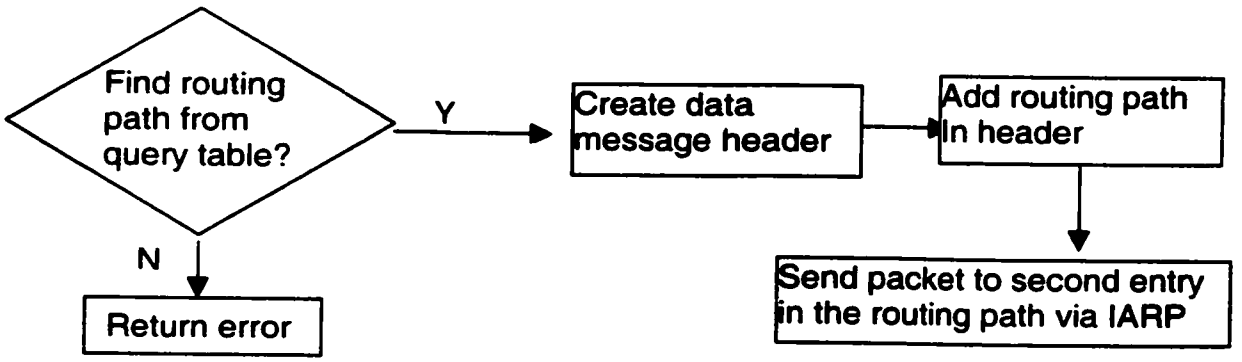
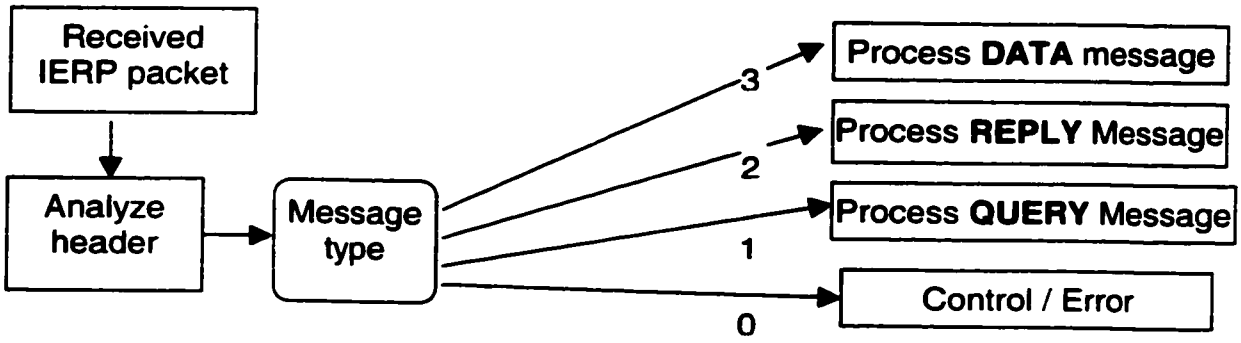


Figure 3-14 IERP sending data from source - - IERP::processSend()

Figure 3-15 IERP process received packet - - IERP::process(packet)



Accordingly, a node in the network may receive any types of above four IERP messages. Figure 3-15 shows the process when IERP receive a packet. After analyzing the message type included in IERP header, IERP protocol will invoke different processing procedures accordingly. We have discussed the Query/Reply message processing procedure in last section. Figure 3-16 illustrates the procedure of processing a received data packet.

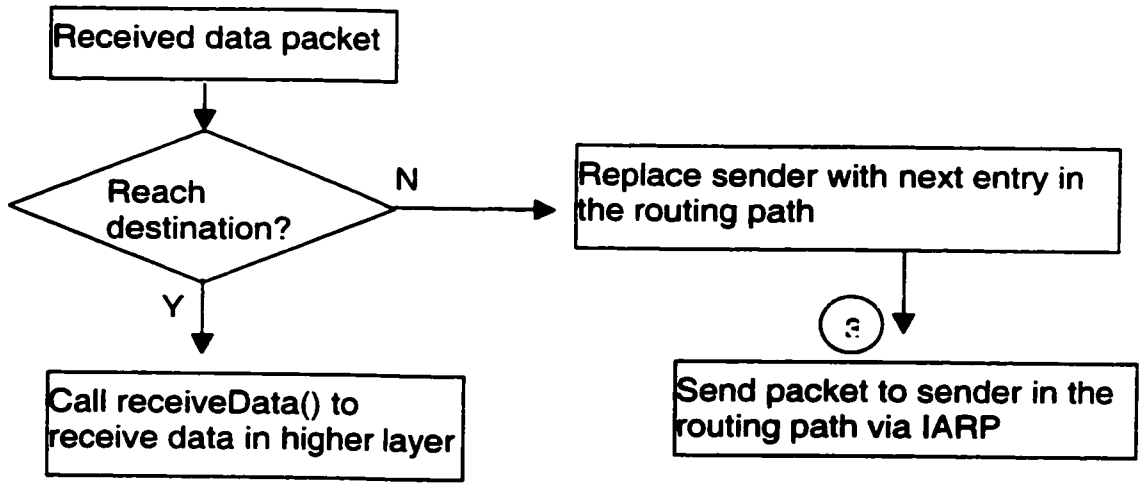


Figure 3-16 IERP receive IERP data packet - - IERP::processData()

IERP data packets will be forwarded from one zone to next along the included route path. While packets are transferred from a node to its border node

in the route, they are normal IARP data packets within the zone and they will not enter the IERP layer. When a border node receives an IERP data packet, it first checks if the packet has reached the destination. If not, it replaces *forwarding sender* (next border node) with the next entry from the included route, then continue to forward the data packet to this next border node address in route. If the packet reaches the destination, IERP has finished the sending procedure. The higher layer can receive the data included in the IERP packet.

IV. ZRP Layer Interface

We adopt network socket style interface between the layers. Wherever the higher layer has a packet to send, it calls OpenConnect() function to open a connection to the destination (see Figure 3-17). OpenConnect() checks if the destination is reachable in its local zone by searching the destination routing information in the local routing table. If it finds the routing information, it will return connection type "IARP", indicating the destination is within its local zone and IARP protocol is supposed to be used for the data deliver to this destination. Otherwise, it invokes IERP protocol to begin a route discovery scheme, queryiing for a route path to the destination. After getting the reply message before timeout limit, the scheme will accumulate and store the discovered route path. Then it returns connection type "IERP" to the caller, which indicates that the packet to the destination must be sent using IERP protocol. If timeout limit is reached

without getting a reply message, it will generate a "destination unreachable" error message.

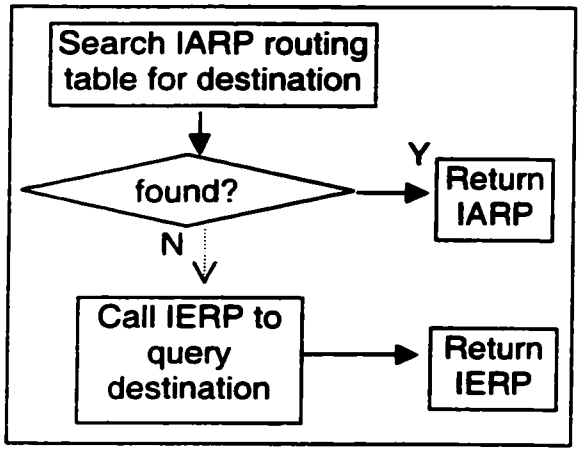


Figure 3-17 Higher layer Open connection

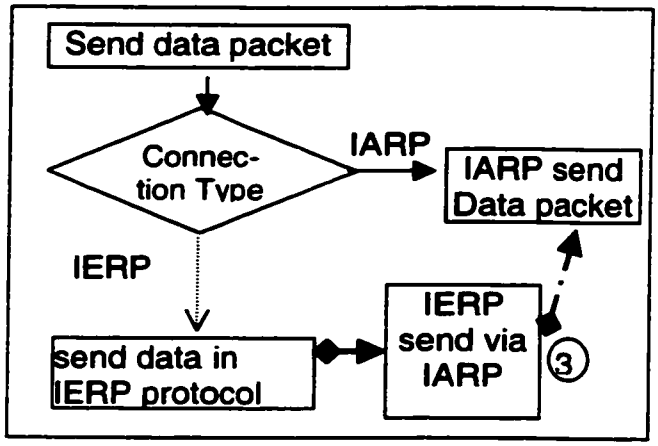
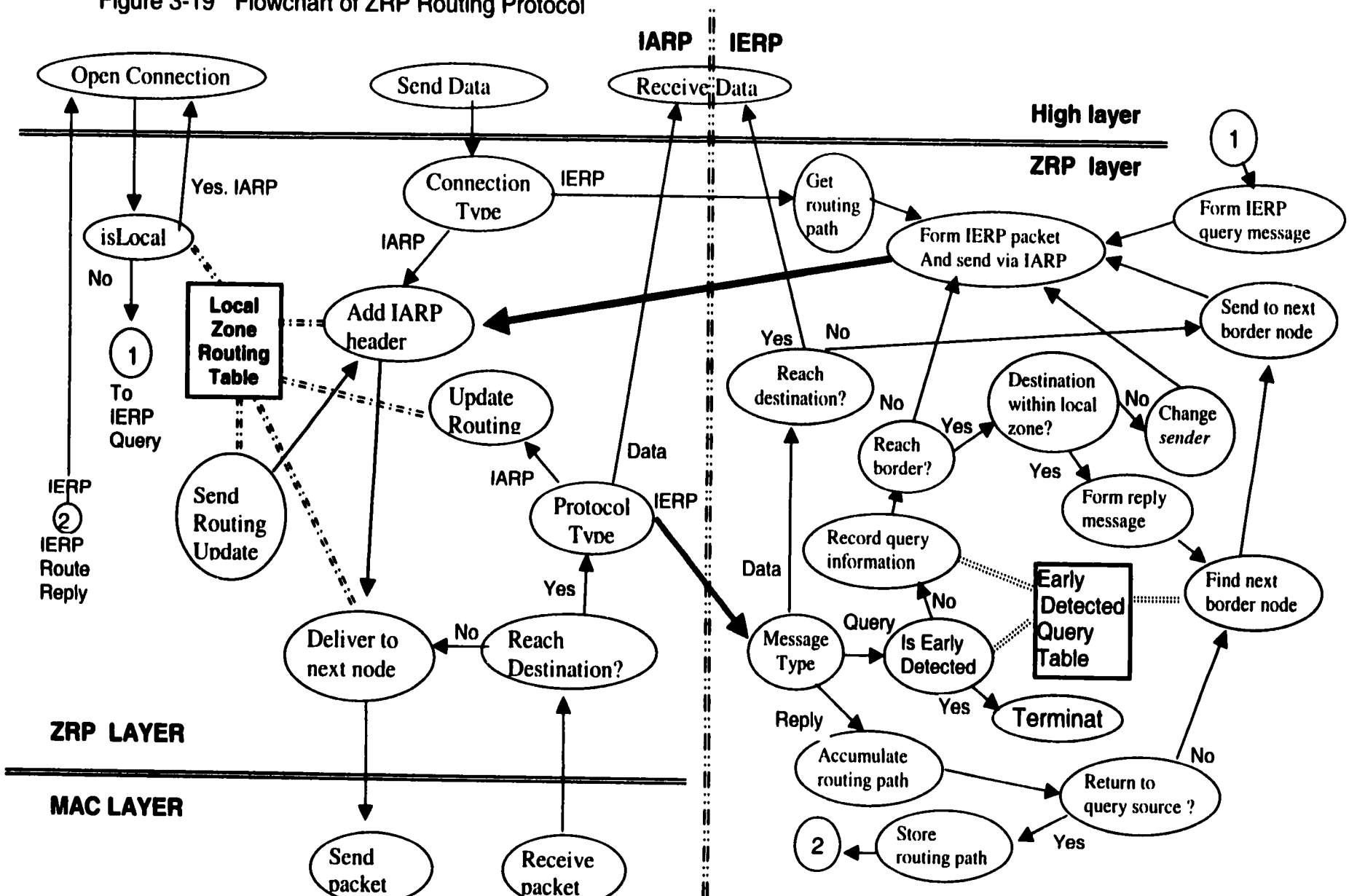


Figure 3-18 Higher layer send data -- send()

To send a data packet, see Figure 3-18, depending on what opened connection type is, the packet is invoked IARP protocol or IERP protocol, which will transform the packet to the destination via IARP protocol using their own already known local routing information or via IERP protocol using the discovered route path information. When ZRP receives a data packet, either by IARP or by IERP layer, it will extract the message header and forward the data containing in the packet to higher layer.

The whole ZRP states that we have implemented are shown in Figure 3-19. For IARP, each node keeps a local zone routing table, which stores up-to-date routing information to all nodes within its local zone. Each node periodically broadcasts its own routing table information to its neighbors; also, it gets routing information from neighbors to update the routing information. The routing information entry consists of the next node address, through which the packet can reach the destination. After adding IARP header, any packet type, including

Figure 3-19 Flowchart of ZRP Routing Protocol



higher layer data, IERP packet and IARP update packet, will be delivered to the next node obtained from the routing table. Upon receiving an IARP packet, the next node will forward the packet through another next node obtained from its own routing table, according to the route information to the destination. This way, the IARP packet is transferred from one node to another until it reaches the destination. The destination will then deliver the packet data to the appropriate upper layer receiver, which is indicated in the *protocol type* field in the IARP header.

IERP protocol is more complex, especially for route discovery procedure. To find a route path from a source to a destination, the source node IERP broadcasts a query message to begin a global search for the destination. The query packet is repeatedly spread out from one zone to another zone. All nodes received the query will record query information into an Early Detected Query Table (ET table), which used to detect same queries and to terminate them early. ET table also records the last border node (previous query sender), showing where the query is received from. Only when the query packet reaches border nodes of each zone, the node checks if the destination is within own local zone. If not, it changes the last border node address (*sender*) in the query message and border-casts the query packet again. Until a border node finds that the destination is within its local zone, it sends a reply message back to the source. The reply packet only delivers to the last queried border node (*sender* of its query) recorded in ET table. After receiving a reply packet, the border node forwards the reply to next border node where the query has come from. As such,

the reply message propagates from one border node to next border node until it returns to the querying source. As the reply packet travels, a route path of a sequence of border nodes from the source to the destination is accumulated from the returning trail. Having the route path to the destination, the source can send IERP data packets to the destination with the route path included. All nodes received an IERP data packet, will forward the packet to the next border node according to the route included in the packet.

Table 3-7 lists main functions having been implemented in our ZRP protocol. The detail C++ codes of each program are presented in Appendix B.

Table 3-7 ZRP functions

FUNCTION	DESCRIPTION
ZRP Interface	
openConnection()	open connection to the destination
send()	send data using opened connection
receive()	receive data using opened connection
sendData()	send ZRP data packet to MAC layer
receiveData()	receive a packet from MAC layer
IARP functions	
IARP::updateTable()	Update/insert a routing entry in the table
IARP::isLocal()	Search routing table, determine whether the destination is within this zone
IARP::getNextNode()	return the next node address to the destination
IARP::checkTTL()	delete timeout entry in routing table
IARP::processUpdate()	received a IARP routing update packet
IARP::sendUpdate()	process and send zone routing table update
IARP::processSend ()	send data to the destination
IARP::processReceive()	Procedure of receive a IARP packet
IARP::deleteEntry()	delete an entry in the position of RT table
IARP::insert()	insert an entry in the position of RT table
IARPHeader::sendToNext()	send packet(include header) to next node
IARPHeader::putHeader()	put header info into IARP packet
IARPHeader::getHeader()	extract header structure from IARP packet
updateMsg::get()	extract a routing entry from update message
updateMsg::put()	put a routing entry into update message
IERP Functions	
IERP:: isEtDetect()	return the sender if item already in etTable
IERP:: isQueried ()	determine if the destination is queried
IERP:: getPath ()	get the route to the determine from queryTable

IERP:: putPath ()	put routing path to the determine in queryTable
IERP:: deleteQuery ()	delete the entry of a destination in queryTable
IERP:: checkTTL ()	delete timeout entry in query and et table
IERP:: process ()	process a received IERP packet
IERP:: processSend ()	send IERP data packet to a destination
IERP:: query ()	Form and Send a query packet
IERPHeader:: query ()	create query header structure
IERPHeader:: reply ()	create reply header structure
IERPHeader:: data ()	form data packet using IERP protocol
IERPHeader:: get ()	extract header information from a packet
IERPHeader:: send ()	send data after getting routing path
IERPHeader:: processQuery ()	process a query message
IERPHeader:: processReply ()	process a reply message
IERPHeader:: processData ()	process a IERP data message

Table3-17 ZRP functions (continue)

V. Realtime Processing Evaluation

We evaluate the real-time processing of our implementation of the ZRP protocol in a computer with an AMD K6 300MHZ CPU, 512K cache, 64M memory. This computer is a low-end computer system in use nowadays, its performance corresponds to a Pentium Pro 300 computer; so most computers should provide better results than presented here. The emphasis here is on the processing time requirements of routing techniques such as ZRP. This is reflection of the complexity of the routing algorithm.

Having implemented the ZRP routing algorithm, to evaluate each subtask, we need to write a separated C++ coded main program with time-tags before and after calling the tested routine. Eventually, by running the real time test program for a routing subtask, the two time-tags are caught. The difference between the two time-tags shows the real processing time for the routing subtask.

Usually, a realtime program runs so fast that it is hard to measure this short time period (several milliseconds or less). Thus, we run each task of the routing algorithm 1000 times repeatedly, then measure its total time consumption. This total running time is divided by the repeat times to obtain the average processing time of the task. In addition, to get rid of random factors such as system calls, disk operations, which may effect our measurements, we measure several times for each evaluation (usually 10 times). The final results presented here are the average values over all test outcomes excluding their minimum and maximum values.

a. Processing Time Comparisons on Different Routing Table Structure

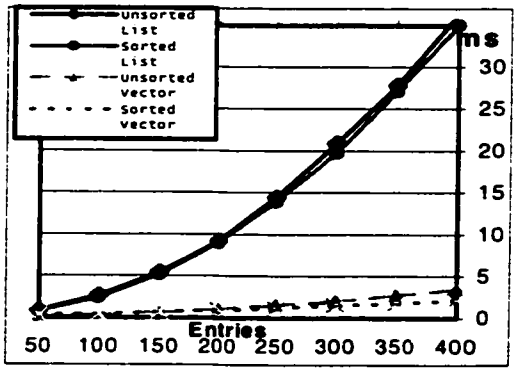
First, let us see how IARP routing table structure has an effect on its real time processing performance. The IARP local routing table contains many routing entries within it. To find one routing entry to a certain destination, the searching routine need to search whole table and to compare each entry with the required destination one by one. When the number of nodes in a zone is large, to find a specific entry from a large size routing table is very time consuming. Moreover, routing information is referenced very often, and is updated frequently, especially for updating, N entries of an update message need N searching routines with N comparisons each routine, totally N^2 comparisons. Thus, the table structure would greatly affect the performance of the routing protocol.

Let us compare four kinds of table structures: unsorted list, sorted list, unsorted vector, and sorted vector. Sorted means the entries in the table are listed in the ascended order according to the destination address. In unsorted table the routing entries are put un-ordered, so it needs to compare all entries if a specific entry is not in the table. List structure is a linear noncontiguous double link list and vector structure is a linear contiguous memory object container. Both list structure and vector structure are implemented as template classes in the new C++ *standard template library* (STL). To compare the above four table storages, we evaluate the processing time for a normal routing table update routine to update a routing table containing a specified number of entries. Supposed in a normal circumstance, a routing table updating routine includes updating 95% old entries and adding 5% new entries. The evaluation results are

present in Table 3-8. Results showed that vector structure has longer processing time (10times) over list structure when the table size is large. This is because vector, like array structure, easily references any positions, whereas list must follow the link one by one. For vector storage, we can further apply Binary Search Algorithm in the sorted vector, so the sorted vector would have improved performance compared with the unsorted vector. For List, the sorted list has better performance than the unsorted, as the sorted list can terminate a search procedure earlier by comparing the value to determine whether the list contains the required entry. From these results, we decided to use the sorted vector storage for IARP routing table.

Table 3-8 Routing Table Structure Comparisons

Process time(ms)	Unsorted List	Sorted List	Unsorted Vector	Sorted Vector
50	0.95	0.88	0.33	0.38
100	2.73	2.52	0.49	0.55
150	5.53	5.33	0.77	0.82
200	9.29	9.12	1.15	1.04
250	14.55	14.06	1.64	1.32
300	21.00	19.88	2.14	1.64
350	27.90	27.19	2.85	1.92
400	36.34	34.99	3.57	2.25



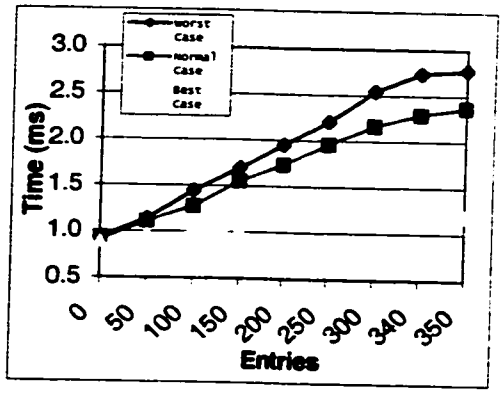
However, as vector is continuous memory storage, its disadvantages are fast inserting and erasing operation only in the end; otherwise, the table needs to be reconstructed. Whereas this is appropriated for frequently updating operation in IARP routing table, it is not suitable for frequently inserting new items and deleting old items like in the IERP Query table and IERP Early Detection table. Usually, the size of IERP Query table or IERP Early Detection table is small, so we use the unsorted list storage structure for these two tables.

b. IARP Sending Update Message

Each node's IARP periodically broadcasts its routing information; the processing time of this send-update procedure depends on the number of entries in its RT table. As the border nodes will become out of the zone range in receivers, only the interior nodes need to be sent to update, so the rate of border nodes in one's table influences greatly its processing time. In the normal situation, we can set the percentage for the number of the interior nodes to 70%. For example, assuming nodes are uniformly distributed over a zone with a radius of 6 hops, the interior nodes area is $\pi(\rho-1)^2$, so the ratio of interior nodes is $\pi(6-1)^2/\pi 6^2=0.69$. While in the worst case most nodes are interior nodes, in the best case, most nodes are border nodes that do not need to be included in the update message. We set the percentage of the interior nodes to 90% in the worst case and to 10% in the best case. The evaluation results of these three cases are shown in Table 3-9.

Table 3-9 Evaluation of IARP Sending Update Message

process time (ms) over # of entries	Worst Case	Normal Case	Best Case
0	0.93	0.91	0.89
50	1.14	1.11	0.97
100	1.44	1.28	0.99
150	1.69	1.55	1.02
200	1.95	1.73	1.08
250	2.21	1.96	1.10
300	2.54	2.17	1.15
340	2.74	2.29	1.19
350	2.78	2.37	1.19



We can see that it takes only 1~3 milliseconds for a node to process sending a routing update message. As normally the rate of interior nodes in its table is high, the normal processing time is approaching the worst case.

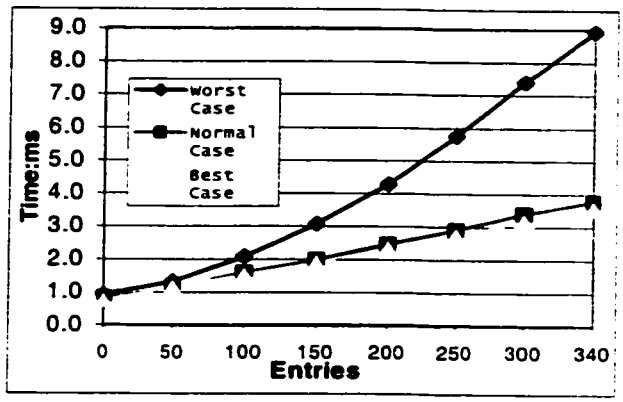
c. IARP Receiving Update Message

In this evaluation, we measure the process time from a node receiving an update message, to having finished updating or adding routing information in its routing table. Each entry in the update message need search whole RT table to locate its old entry position before it can be updated. As mention above, it needs $O(N^2)$ comparisons in the searching routine for updating N entries. Even though we implement *Binary Search* algorithm, updating the routing table is still a time consuming operation. While updating an old entry, the node just needs to locate the old position in the table and replace old values with new values. For an entry of new routing information, it first needs to search the whole RT table, and then knows the entry is new because it cannot be found. Finally, the new entry can be inserted into the appropriate position in RT table; this operation may include locating memory, copying values and resorting vectors. Thus, adding new entries takes more time than updating old entries. In the worst case, all entries in an update message are new (like a node just powered-on), and need to be inserted into RT table. In the best case, all entries already have old values contained in the table, which just need to be updated. In normal operation, only small part of routing information is new, we set this part to be 10% of total entries in our evaluation. In our evaluation, we assume a node, which has N entries in its RT table, receives N entries of update message. The evaluation results are shown in

Table 3-10. We can see that the size of the routing table influences greatly the table updating. Moreover, the worst cast takes much longer than others; the normal case processing time approaches the best case. Fortunately, the worst case only happens once when a node is powered on.

Table 3-10 Evaluation of IARP Receiving Update Message

process time (ms) over table size	Worst Case	Normal Case	Best Case
0	0.93	0.83	0.76
50	1.32	1.15	1.15
100	2.08	1.59	1.54
150	3.08	1.98	1.92
200	4.29	2.47	2.36
250	5.76	2.91	2.75
300	7.41	3.40	3.18
340	8.95	3.79	3.57



From the evaluation results one can see that, if normally a zone has 100 nodes and on average each node has 10 neighbours, the time for the routing information update operation is one process of sending update message (1.28ms form Table 3-9), and ten processes receiving update message from its 10 neighbours (1.59ms form Table 3-10), totally 1.28+10*1.59 ≅ 17(ms); the worst case is 1.44+10*2.08 ≅ 22(ms). Therefore, if a node's updating interval is every two seconds, the routing control processing only consumes one percent (1.1% in the worst case) of a node's total CPU time.

d. IARP Data Transfer

IARP data transfer has three different routines: source node forming packet, intermediate node forwarding packet and destination node receiving

packet. The processing time of these three data transfer methods is shown in Table 3-11. This evaluation is done in a 100-node zone, but as we implemented sorted vector storage and the binary search algorithm for the routing table, and as only need one search to obtain *next node* to forward data packet, the time difference between the sizes of RT table is ignorable. For forming a data packet, the source node needs to add the IARP header before data. This operation needs to locate new memory space for the total length of the data and header; and after putting the IARP header, the original data payload needs to be copied into the allocated space to form a IARP data packet. So the longer the data size, the more processing time it needs. For intermediate nodes to forward a data packet, it needs to receive the packet, and then sends the packet to the *nexthop* to the destination, thus its processing time almost equals to the sum of sending packet and receiving packet processing.

Process time(ms) over payload	Send Data	Forward data	Receive Data
256	0.69	1.29	0.58
512	0.82	1.40	0.60
768	0.85	1.46	0.61
1024	0.96	1.54	0.62
1280	1.06	1.62	0.63
1536	1.10	1.68	0.66
1792	1.13	1.73	0.67
2047	1.21	1.84	0.69

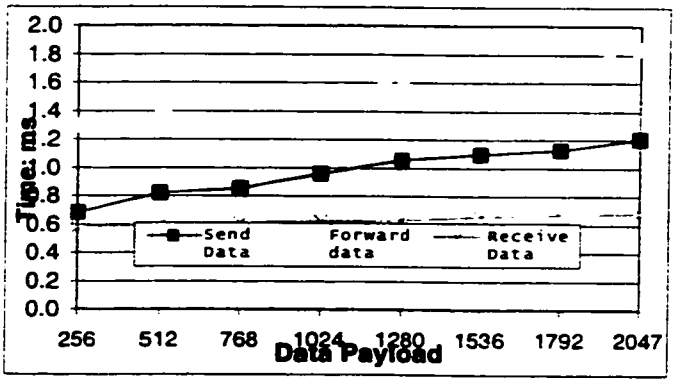


Table 3-11 Evaluation of IARP Data Transfer

We can see that the processing delay for the intrazone IARP data transition is less than 2 ms for each node, and the processing rate for a large data packet (2047*8/1.84 ≅ 9Mbps) is much better than a small data packet

(256*8/1.29 ≅ 1.5 Mbps). If sending a 2047-byte-long data packet in the above mentioned testing zone with a radius of 6 hops, the longest travel distance is from the central node to a border node, the total processing delay is one sending, 4 forwarding, and one receiving processing, only 1.21+4*1.84+0.69 = 9.26 (ms).

e. IERP Query and Reply Process

When a source node does not know the route path to a destination, it will begin a route discovery procedure. This procedure includes: the source node forming the query packet, interior nodes within zones forwarding the query, intermediate border nodes between zones processing and forwarding the query, the destination border node processing the query and forming the reply message, interior nodes delivering the reply as a IARP data packet, intermediate border nodes accumulating the route path and forwarding the reply, and at last, the querying source getting the reply message and extracting the routing path from the reply message. Among these routines, the IERP reply packet transferring between interior nodes within a zone is an IARP data packet delivery procedure.

Table 3-12 Processing Time of IERP Query and Reply Routines

Routines	processing time
source form qurey	0.96 ms
interior forward qurey	1.65 ms
intermediate border node forward qurey	1.65 ms
destiantion border node process qurey/reply	1.65 ms
intermediate border node forward reply	1.43 ms
querying source get reply	0.66 ms

Table 3-12 shows the evaluation of processing time for each IERP query/reply processing routines. Except the source node sending a query and receiving the reply, all others involve receiving and re-transmitting two routines, thus it takes much more processing time. Because local zone routing table searching is fast, and the early detected table and query table is small, the size of these tables does not affect much on the realtime performance of processing the query or reply message. The evaluation also shows a very close performance on the difference table size; the results present in Table 3-12 are the average processing time while the number of entries in the tables is 100, 200, and 300. We can see that as the length of an IERP query or reply message is very short, its processing time for each node is small (1~2 ms); thus our IERP route discovery algorithm is efficiency and bandwidth saving.

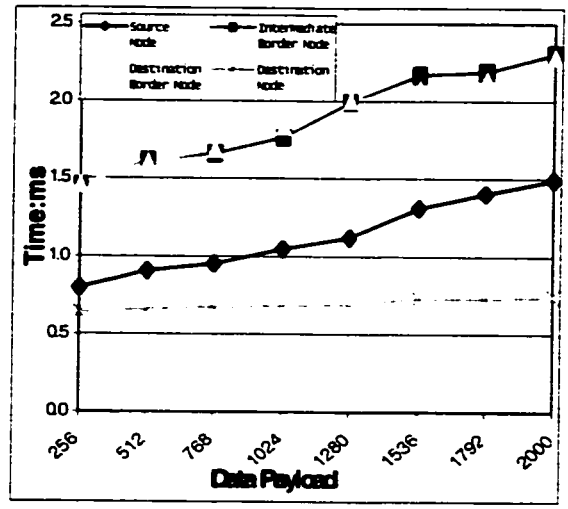
f. IERP Data Delivery

After IERP discovers a route path to the destination, the source can begin to send data packets between zones using IERP protocol. The IERP data delivery procedure includes: the source node forming a IERP data packet, intermediate border nodes forwarding the packet according to the included route path, the destination border node transferring the packet to the destination, and the destination receiving the IERP data packet. We evaluate the real time IERP data delivery processing in different data payload sizes; the results are shown in Table 3-13. Except the destination node receiving data packet, in other three cases, the size of the data packet affects a lot on the processing time. This is

mostly caused by data moving in the memory while forming IARP packet, as the border node will deliver the IERP data packet to *next border node* within a zone as IARP data packets.

Table 3-13 Evaluation of IERP Data Delivery

Process time(ms) via payload	Source Node	Intermediate Border Node	Destination Border Node	Destination Node
256	0.80	1.46	1.46	0.64
512	0.91	1.62	1.61	0.66
768	0.95	1.66	1.69	0.67
1024	1.05	1.77	1.83	0.68
1280	1.12	1.99	2.00	0.69
1536	1.31	2.17	2.09	0.73
1792	1.40	2.19	2.15	0.73
2000	1.49	2.30	2.29	0.74



Chapter Four Conclusion and Future Consideration

The WLAN field is rapidly evolving. A fast-growing market introduces the flexibility of wireless access for mobile users. In this thesis, Chapter 1 first introduced the wireless communication standard -- IEEE 802.11, emphasizing on an Ethernet like CDMA/CA MAC protocol, the ad hoc network access mode, and broadcast/multicast algorithms.

However, few application programs are special designed to take advantage of mobile wireless communication. In Chapter 2, a wireless application program, MYMAR system, was developed. Our MYMAR system is a distributed free public access wireless network with no centralized base stations, no cellular system, and no service providers to pay. MYMAR is intended to be a yellow page like advertisement tool for tourists, travelers, and dwellers. A wireless network, digital maps and user-friendly voice recognition system enable the mobile users to navigate service information and to reach the destination in the shortest possible time. Also short voice messages can be exchanged between MYMAR users. Wireless access to the Global Internet is also provided.

We present the basic architecture and system operation of the MYMAR system with emphasis on the software development. Three MYMAR applications are specially choosed to satisfy the most demands in mobile computing. Voice recoginition based voice command system provides a user-friendly, hand-free operation environment. Some controls are programmed using Java or WINDOW

C++ language; moreover, network socket and communication port API techniques are used in programming. The new concept of MYMAR has been verified both theoretically and practically. A BS and a MS were setup for test, and bussiness informatioin about Montreal downtown and Brossard, Quebec were collected as test model. Both indoor and outdoor testing proved the practicality, usefulness, and versatility of the multi-application multimedia services obtained. Excellent communication ranges were obtained.

Due to the mobility factor, the MS may move out of the WLAN of the contacted BS, in this case the associated packets may be routed via a suitable routing strategy. The packet routing is also necessary for communication between WLAN segments. In Chapter 3, we looked over current available routing protocols for wireless network, namely flooding, proactive routing, on–demand routing, and location assisting routing. A realtime Zone Routing Protocol (ZRP), which combines precumputed routing for in-zone traffic and on-demand routing for out-zone traffic, was implemented. The IARP and IERP two separate sublayer structure made the implementation of ZRP simple. C++ Object oriented programing style was introduced to make the program fast reference and easy modification in the future.

Some optimal configurations were made to improve the ZRP performance. Broadcasting known routing information in the update messages instead of traditional "HELLO" message reduces routing control traffic. The local update message carrying timestamp adapts the fast mobility factor of wireless network and kept routing information most up-to-dated. Broadcasting IERP queries to

neighbours instead of bordercasting the queries to border nodes took advantage the broadcasting nature of wireless communication. Early detection/termination and backward route accumulation greatly improved the efficiency of the interzone route discovery. At last, performances under four different table structure implementations are compared. The realtime processing time for each ZRP subtask was evaluated for reference.

Our realtime messaging and routing approaches make an effective and efficient use of the unlicensed free bands within the nested wireless networks. Future research and development in this area can prove to be very useful and can bring about a new revolution in mobile computing devices. More applications should be developed for mobile users, for instance, Internet phone (voice IP over WLAN). Voice recoginitioin program should be updated to a more accuracy version. And other alternatives should be seeked; for example, attaching few simple buttons or using touch screen instead of our voice commands to control the MYMAR system may prove to be effective and accurate. The implementation of ZRP or other routing protocols should be intergrated into MYMAR system for out of WLAN segment traffic. The result of the evaluated process time under ZRP routing protocol should be taken into the consideration of the performance of realtime voice messaging over wireless LANs. All these considerations remain the important subjects for future research.

References

[1] J. Schiller, *Mobile Communications*, Addison-Wesley, 2000, ISBN 0-201-39836-2, pp 276-286.

[2] Committee on Evolution of Untethered Communications,"Technology Limits, trade-offs, and Challenges," http://www.nap.edu/readingroom/books/evolution/2.html.

[3] Brain P. Crow, Indra Widjaja, Jeong Geun Kim and Prescott T. Sakai, "IEEE 802.11 Wireless Local Area Networks", *IEEE Communications*, September 1997, p116-126

[4] Wireless Media Access Control and Physical Layer WG, "Wireless LAN Medium Access Control (MAC) and Physical layer (PHY) specification," IEEE standard P802.11, Nov. 1997.

[5] K. Feher, *Wireless Digital Communication: Modulation and Spread Spectrum Applications*, Prentice-Hall, 1995.

[6] R. Peterson, R. Ziemer and D. Borth, *Introduction to Spread Spectrum Communications*, Prentice Hall, 1995.

[7] S. Khurana, A. Kahol, and A.P. Jayasumana, "Effect of Hidden Terminals on the Performance of IEEE 802.11 MAC Protocol", *In Proc. of IEEE Annual Conference on Local Computer Networks* (LCN'98), Boston, Massachusetts, October 1998, *p12-20.*

[8] J. Kuri and S.K. Kasera, "Reliable Multicast in Multi-access Wireless LANs", *INFOCOMM 1999*, pp 760-767,

[9] K. Tang and M. Gerla, "Random Access MAC for Efficient Broadcast Support in Ad Hoc Networks", *Proceedings of IEEE WCNC 2000*, Chicago, Sep. 2000.

[10] The Bluetooth Consortium, http://www.bluetooth.com, 1998

[11] IEEE, *http://grouper.ieee.org/groups/802/11*

[12] European Telecommuniction Standards Institute (ETSI), *Broadband Radio Access Networks*, EN 300 652, http://www.etsi.org, 1998

[13] A.K.Elhakeem, H.Shafik Ullah, L.Boucher, G.Chan, "SFH Access to Mobile Yellow page Messaging and Retrieval, *The advent of the Local Wireless Internet*, SCI 2000 Conference, Orlando, Florida, July 2000.

[14] A.K. Elhakeem, L. Boucher, G.Chan, "MYMAR, A new Mobile Yellow Page Messaging and Retrieval, *the advent of the Local Wireless CCEE*, May 2000, Halifax.

[15] B.P. Crow, I Widjaja, J.G. Krim, P.T Sabai, "IEEE 802.11 Wireless local Area Networks", *IEEE Comm. Magazine*, pp 166, Sept 1997.

[16] Z.J. Haas, "Wireless Ad hoc Networks", *IEEE J. on Selected Areas in Com.*, Vol 17, No. 8, August 1999.

[17] GINA American, "GINA 2002/2003 Wireless LAN USER'S MANUAL"

[18] Sun Microsystems, Inc., "Java 2 SDK, standrd edition", http://java.sun.com/j2se/1.3/

[19] Realize Software Cor., "Realize Voice", http://www.realizesoftware.com/rv

[20] B.Wellenhaf, H, Wicthenegger, C., *Global Positioning Systems*, Springer-Verlas, New York 1993.

[21] Canadian Macori Com. "ALLSTAR USER'S MANUAL", V4, Nov. 1998.

[22] Sun Microsystems, Inc., "Java Communications API", http://java.sun.com/products/javacomm/

[23] ArGo Software Design, "FTP Server", http://www.argosoft.com/applications/ftpserver

[24] GlobalSCAPE Inc., "CuteFTP", http://www.cuteftp.com/products/cuteftp

[25] Vicinity Cor., *Map Blast! Web site*, http://www.mapblast.com

[26] Sagebrush Systems Inc., "RecAll-Pro", http://www.sagebrush.com

[27] Ositis Softwre, "Winproxy 3.0", http://www.winproxy.com, 2000

[28] M. Gerla, G. Pei, and S.J. Lee, "Wireless, Mobile Ad-hoc Routing", *IEEE/ACM FOCUS'99*, New Brunswick, NJ, May 1999

[29] C.E. Perkins and P. Bhagwat, "Highly dynamic Destination-sequenced Distance-vector Routing (DSDV) for Mobile Computer", *ACM SIGCOMM'94*, pp 234-244, 1994.

[30] A. Iwata, C. C. Chiang, G. Pei, M. Gerla and T. W. Chen, "Scalable routing strategies for ad hoc wireless networks", *IEEE J. on Selected Areas in Com.*, Vol. 17, Aug. 1999, pp1369-1378

[31] D.Johnson and D.Maltz, "Dynamic Source Routing in Ad hoc Wireless Network", *Mobile Computing*, edited by Taasz Imelinsky, Klumer Academic Publishers, 1996

[32] Y. B. Ko and N.Y. Vaidya, "Location-Aided Routing (LAR) in Mobile Ad Hoc Networks", *In Proceedings of MOBICOM'98*, Dallas, TX, Oct. 1998, pp 66-75.

[33] S. Basagni, I. Chlamtac, V.R. Syrotiul, and B.A. Woodward, "A Distance Routing Effect Algorithm for Mobility (DREAM)", *In Proceedings of MOBICOM'98*, Dallas, TX, Oct. 1998, pp 76-84.

[34] A.K.Elhakeem, L.Boucher, G.Chan, Z.Li, A.Masood, "Forwarding Data Basis and Routing Techniques for Nested Clustered of Wireless LANS", to be submitted to the IEEE JSAC Journal Oct. 2000.

[35] M.R. Pearlman and A.J. Hass, "Determining the Optimal Configuration for the Zone Routing Protocol", *IEEE J. on Selected Areas in Com.*, Vol. 17, Aug. 1999, p1395-1414

[36] D.E. Comer, "Internetworking with TCP/IP Vol 1: Principle, Protocol, and Architecture", 2nd edition, Prentice-Hall, 1991.

[37] D.E. Comer and D. Stevens, "Internetworking with TCP/IP Vol II: Design, Implementation, and Internals", 3rd edition, Prentice-Hall, 1999.

Appendix A MYMAR Programs

GPSRead.java

```
import javax.comm.*;
import java.util.TooManyListenersException;
import java.io.*;

public class GPSRead implements
SerialPortEventListener {
    //Default port for GPS
     static String portName="COM1";
    // port parameters
     static int portParameter[]={9600,
            SerialPort.DATABITS_8,
            SerialPort.STOPBITS_1,
        SerialPort.PARITY_NONE};

     String outputFileName="gpsreading.txt";

     int msgLength=60;     //length of message
     //Header of NMEA GPS message
     static String msgStart="$GPGGA";
    //field parameter in the message read from comm
port
     static int msgField[][]={ {29,39,40},{  //longitude
first
                17,     //begin of message field index
                26,    //end of massage field index
                27          //position of signed
                } };
     static char msgSign[]={'E','N'};  //the sign for
positive
                                  //value:'S','w' for
                        nagetive
     static String msgData[]=new
String[msgField.length];

     SerialPort sPort;
     CommPortIdentifier portId;
     InputStream is;
     String dataString=""; //hold blocked input data

/** Write data from GPS output string to file outputfile.
*  if in contain GPS NMEA format H-
header=msgStart*/
public void writeFile(String in) {
     int start;
     if( (start=in.indexOf(msgStart))>=0) {
        for(int i=0;i<msgField.length;i++){

msgData[i]=in.substring(start+msgField[i][0],
                        start+msgField[i][1]);

if(in.charAt(start+msgField[i][2])!=msgSign[i])
              msgData[i]="-"+msgData[i]; //add the
sign
        } // end of for loop

        try  {
        // Create a new file output stream connected to
        //"gpsreading.txt".  Connect print stream to the
        //output stream
         PrintStream p = new PrintStream( new
FileOutputStream(outputFileName));
        for(int i=0;i<msgField.length;i++) {
            try{
                float f=new Float(msgData[i]).floatValue();
                int k=(int)f/100;
                f=f-k*100+k*60;     //trans to seconds
                p.println (f);
                System.out.println("GPS Reading is ::" + f);
            }catch (NumberFormatException fe) { };
         }
         p.close();
        }    catch (Exception e)  {
          //Don't exit, in case file is opened for reading
        System.err.println ("Error writing to file");
        }
     } //End of if
   } //End of writeFile()

   /**   Main method. Checks to see if the command line argument
contains comm port information ("COM" or "com"). otherwise
display a usage message and   exit  */
   public static void main(String args[]) {
      if (args.length >0) {
        if(args[0].startsWith("COM")
                    || args[0].startsWith("com"))
          portName=portName.substring(0,3)+
                args[0].substring(3,4);
        else {
           System.out.println("usage: java GPSRead [commPort]");
           System.exit(1);
           }
     };
      GPSRead serialRead = new GPSRead();
   } //end of main()

   /**    Create new GPSRead and initilizes it.    */
   public GPSRead(){
     // Obtain a CommPortIdentifier object for the port
     try {
        portId =CommPortIdentifier.getPortIdentifier(portName);
     } catch (NoSuchPortException e) {
        System.err.println("Error. Can't find the port:"+portName);
        System.exit(-1);
     }

      if(portId.isCurrentlyOwned()) {
           System.err.println("This port is owned by other
application");
           System.exit(-1);
      }

/* Open the port represented by the CommPortIdentifier object.
Give the open call a relatively long timeout of 5 seconds */
      try {
           sPort = (SerialPort)portId.open("GPSRead", 5000);
      } catch (PortInUseException e) {
           System.err.println("Error. Can't open the
port:"+portName);
           System.exit(-1);
      }

/*   // Set the parameters of the connection  If they won't set, close
the port before throwing an exception.*/
      try {
        sPort.setSerialPortParams(portParameter[0],
                    portParameter[1],
```

```
                    portParameter[2],
                    portParameter[3]);
        } catch (UnsupportedCommOperationException e)
 {
                System.err.println("Unsupported
 parameter");
                sPort.close();
                System.exit(-1);
              }
 // Open the input and output streams for the connection.
 // If they error, close the port before throwing an
 exception.
      try {
          is = sPort.getInputStream();
      } catch (IOException e) {
         sPort.close();
         System.err.println("Error opening i/o streams on
 serial port");
         System.exit(-1);
      }

     // Add this object as an event listener for the serial
 port.
      try {
         sPort.addEventListener(this);
      } catch (TooManyListenersException e) {
         sPort.close();
         System.err.println("too many listeners added");
      }

 // Set notifyOnDataAvailable to true to allow event
 input.
      sPort.notifyOnDataAvailable(true);

 } //end of GPSRead()

 /**  Handles SerialPortEvents. The type of
 SerialPortEvents that this  program is registered to
 listen for are DATA_AVAILABLE and . During
 DATA_AVAILABLE the port buffer is read.   */
    public void serialEvent(SerialPortEvent e) {
      // Determine type of event.
      switch (e.getEventType()) {
         case SerialPortEvent.DATA_AVAILABLE:

              try {
                 if (is.available() > 0) {
                      byte readBuffer[]= new
 byte[is.available()];
                   is.read(readBuffer);
                      String tmp=new String(readBuffer);
                      dataString+=tmp;
                      int indx;

 if((indx=dataString.indexOf(msgStart))>=0){
     dataString=dataString.substring(indx);
     if(dataString.length()>=msgLength) {
      //wait for get full length of message
           writeFile(dataString);
           dataString=dataString.substring(msgLength);
      }
   }else if((indx=dataString.length()-msgStart.length()-
 2)>0)
          dataString=dataString.substring(indx);
   } //end of if
                }catch (IOException ie) {
                  System.err.println("Error while reading
 data from port.");
                }
              break;
          default: break;
        } //End of switch
    }  //end of serialevent()

   //Close open serial port before finish program
   protected void finalize() {
     try{
       if(sPort!=null) sPort.close();
       if(is!=null) is.close();
     }catch(IOException e) { };
   }
 }     //end of class GPSRead
```

MymarMultiServer.java

```
import java.net.*;
import java.io.*;

public class MymarMultiServer
{
   public static void main(String args[]) throws IOException
   {
     ServerSocket serverSocket = null;
     boolean listening = true;
     PrintStream out = null;
     DataInputStream in = null;
     String IPaddr =  null;

     try
        {
          //InetAddress localaddr = InetAddress.getLocalHost();
          serverSocket = new ServerSocket(4444);
          //IPaddr = localaddr.getHostAddress();
            IPaddr = "1.1.1.102";
        }
     catch (IOException e)
        {
         System.err.println("Could not listen on port: 4444");
         System.exit(1);
        }

     while (listening)
       new MymarMultiServerThread(serverSocket.accept(),
                        IPaddr).start();

     serverSocket.close();
   }
}
```

MymarMultiServerThread.java

```
import java.net.*;
import java.io.*;

public class MymarMultiServerThread extends Thread
{
   private Socket socket = null;
   private String IPaddr = null;

   public MymarMultiServerThread(Socket socket, String IPaddr)
   {
     super("MymarMultiServerThread");
     this.socket = socket;
     this.IPaddr = IPaddr;
   }
```

```
    public void run()
    {
      try
        {
        PrintStream out = new
 PrintStream(socket.getOutputStream());
        DataInputStream in = new
 DataInputStream(socket.getInputStream());

        out.println("WELCOME TO MYMAR");
        while(in.readLine() != null)
        {
        out.println(IPaddr);
        if (in.readLine().equals("ACK"))
          {
          out.println("BYE");
          break;
          }
        }
        socket.close();
        in.close();
        out.close();
        }
     catch(IOException e)
        {
          System.err.println(" IO faliure....");
        }
    }
}
```

MymarClient.java

```
import java.io.*;
import java.net.*;

public class MymarClient {
      static FileOutputStream addr = null;
      static PrintStream fopen = null;

   public static void main(String[] args) throws
IOException
   {
      Socket MymarClientSocket = null;
      PrintStream out = null;
      DataInputStream in = null;

     String IPaddrtmp = "1.1.1."; //Constant field for all
the
                                  //IP addresses
     String IPaddr = null; //String for the current IP
address
     int i = 102;              //last byte field for the IP
address
     boolean IPaddrflag = true;
     Runtime rt = Runtime.getRuntime();

     while(IPaddrflag)   {
       try
       {
       IPaddr = IPaddrtmp + i;
       System.out.println(" Host..." +IPaddr);
        MymarClientSocket = new Socket(IPaddr,
4444);
       IPaddrflag = false;
       }
     catch (Exception e)
       {
          i++;
          IPaddrflag = true;
          }
           if( i == 103)
             {
             try
               {
                  addr = new FileOutputStream("ipaddress.txt");
                  fopen = new PrintStream(addr);
                  fopen.println("");
                  fopen.close();
               }
             catch( Exception e)
               {       System.err.println("Error opening file....");
               }
              System.exit(-1);
             }
              }

     try
        {
         out = new
PrintStream(MymarClientSocket.getOutputStream());
         in = new
DataInputStream(MymarClientSocket.getInputStream());
        }
     catch(IOException e)
        {
          System.err.println("Couldnt get IO for the connection to
the host");
        }

      BufferedReader stdIn = new BufferedReader(new
InputStreamReader(System.in));
      String fromServer;
      String fromUser;
     out.println("Hi");// Client to initialize handshake

      while ((fromServer = in.readLine()) != null)
        {
        System.out.println("Server: " + fromServer);

        if (fromServer.equals(IPaddr))
        {
         out.println("ACK");
         try
            {
                addr = new FileOutputStream("ipaddress.txt");
                fopen = new PrintStream(addr);
                fopen.println("host " + IPaddr);
            }
         catch(IOException e)
            {
              System.err.println("Error writing the file....");
            }
       }

       if (fromServer.equals("BYE"))
          break;
       }
     out.close();
     in.close();
     stdIn.close();
     MymarClientSocket.close();
   }
}
```

Display.java

```
import java.awt.*;
import java.io.*;
import java.util.*;
import java.awt.event.*;
import java.lang.*;

public class f1_gpstst extends Frame implements
Runnable,
WindowListener {

MediaTracker imageTracker ;
Font f = new Font("TimesRoman",Font.BOLD,10);
int xpos, ypos,testposx, testposy , thewidth , theheight;
float javdiffx = 0,javdiffy = 0,cordiffx = 0,cordiffy = 0,
ftestposx;
float propx, propy;
String numa,numb,numc,numd;
float corx1, corx2, cory1, cory2;
String abc1,abc2,abc3,abc4;
float testx , testy ;
Image offscreenImg,flagimg ;
Graphics offscreenG;
int firstTime = 1;
boolean Ablink = true;

static f1_gpstst gpstester;

public static void main(String[] args)
{
      gpstester = new f1_gpstst();
      gpstester.setVisible(true);
}

public f1_gpstst()
{
      addWindowListener(this);
     setLayout(null);
     setSize(640,480);
     setTitle("MYMAR");
      Thread t = new Thread(this);

     t.start();
}

public void run()
{
   while (true)
   {
         repaint();
         try
         {
           Thread.sleep(1600);
         }
         catch (InterruptedException e)
          {
           e.printStackTrace();
          }
   }
}

public void update(Graphics g)
{
     paint(g);
}

public void paint(Graphics g) {
    // Open a new frame for a new map
     try
     {   //text file contains 1 each time the map is changed
         FileInputStream in_newframe = new
FileInputStream("newframe.txt");
         DataInputStream in_newframestream = new
DataInputStream(in_newframe);
         String newframeflag = in_newframestream.readLine();
         in_newframe.close();
         in_newframestream.close();

         if(newframeflag.equals("1"))
         {
             FileOutputStream out_newframe = new
FileOutputStream("newframe.txt");
             PrintStream out_newframestream = new
PrintStream(out_newframe);
             out_newframestream.println("0");
             out_newframestream.close();
             out_newframe.close();
             System.out.println("exit...");
             System.exit(-1);
         }
     }
     catch(Exception e)
     {
     System.err.println("Problems handling file newframe.txt");
     }

          imageTracker = new MediaTracker(this) ;
          offscreenImg = createImage(this.size().width,
this.size().height);
          offscreenG = offscreenImg.getGraphics();
          flagimg = getToolkit().getImage("D:\\MYMAR-
USER\\USER_JAVA\\map.gif");
          imageTracker.addImage(flagimg, 1);

          try
          {
            imageTracker.waitForAll();
          }
          catch(InterruptedException ex)
          {
            ex.printStackTrace();
          }

          thewidth = flagimg.getWidth(this);
          theheight = flagimg.getHeight(this);

    try
          {  //MAP-COORDIANATES FILE
        FileInputStream fstream = new
FileInputStream("D:\\mymar_user\\user_java\\map.txt");
          DataInputStream in = new DataInputStream(fstream);
          numa = in.readLine();
          numb = in.readLine();
          numc = in.readLine();
          numd = in.readLine();

              corx1 = new Float(numa).floatValue();
                    // top left corner
              cory1 = new Float(numb).floatValue();
              corx2 = new Float(numc).floatValue();
                    //Bottom right corner
              cory2 = new Float(numd).floatValue();

            cordiffx = Math.abs(corx1-corx2);
               // size of map in terms of gps coordinates
               cordiffy = Math.abs(cory1-cory2);

                propx = thewidth/cordiffx;
```

```
                    //Propotionality constants
                  propy = theheight/cordiffy;

                      in.close();
                fstream.close();
            }
            catch (Exception e)
            {
              e.printStackTrace();
            }

          try
            {
        // GPS READING file
              FileInputStream fstream = new
            FileInputStream("D:\\mymar_user\\user_jav
            a\\gpsreading.txt");
          DataInputStream in = new
 DataInputStream(fstream);

              abc1 = in.readLine();
              abc2 = in.readLine();

                      in.close();
    }
    catch (Exception e)
    {
          e.printStackTrace();
    }

          int i=0;
          float destination_log[] = new float[10];
          float destination_lat[] = new float[10];
          String str;
          try
          {
        // DESTINATION File
            FileInputStream fstream = new
FileInputStream("D:\\mymar_user\\user_java\\destinati
on.txt");
        DataInputStream in = new
DataInputStream(fstream);

              while(((str=in.readLine()) != null))
                {
                  if(str != null && str != "" && str !=
" ")
                  {
                      destination_log[i] = new
Float(str).floatValue();
                      if((str=in.readLine()) != null)
                          destination_lat[i] = new
Float(str).floatValue();
                  }
                  i++;
                }
            in.close();
    }
        catch (Exception e1)
    {
        e1.printStackTrace();
    }

    testx =  new Float(abc1).floatValue();
              //coordinates for GPS readings
    testy =  new Float(abc2).floatValue();
   testposx = Math.round(Math.abs(testx - corx2) *
propx); //GPS position in terms of pixels
   testposy = Math.round(Math.abs(testy - cory2) *
propy);

    g.setFont(f);
    offscreenG.drawImage(flagimg,0,0, this);

    offscreenG.setColor(Color.red);

    for(int j=0;j<i;j++)  // Destination projection
    {
        offscreenG.setColor(Color.red);
        offscreenG.fill3DRect(640-(Math.round(Math.abs
(destination_log[j]-corx2)*propx)), 480-(Math.round(
Math.abs(destination_lat[j]-cory2)*propy)), 15, 15, true);

        offscreenG.setColor(Color.yellow);
        offscreenG.drawString("D",640-(Math.round
(Math.abs(destination_log[j]-corx2)*propx))+3,480-(Math.
round(Math.abs(destination_lat[j]-cory2)*propy))+10);

    }

        offscreenG.setColor(Color.blue);
       // GPS position projection
    offscreenG.fillOval(640-testposx,480-testposy,10,10);

    g.drawImage(offscreenImg,0,0,this);
    offscreenG.dispose();

    }

    public void destroy()
    {
        offscreenG.dispose();
    }

    public void windowClosing(WindowEvent e)
    {
    dispose();
    System.exit(0);
    }

    public void windowOpened(WindowEvent e){};
    public void windowDeactivated(WindowEvent e){};
    public void windowDeiconified(WindowEvent e) {};
    public void windowIconified(WindowEvent e) {};
    public void windowClosed(WindowEvent e) {};
    public void windowActivated(WindowEvent e) {};
}
```

StartListening.cpp

```
// CFindwinApp initialization
BOOL CFindwinApp::InitInstance()
{
    // Standard initialization
    /*If you are not using these features and wish to reduce the
size of your final executable, you should remove from the
following  the specific initialization routines you do not need.*/

    HWND h=FindWindow(NULL,"Realize Voice");

    PostMessage(h,WM_SETFOCUS,(WORD)h,0);
    PostMessage(h,WM_SYSKEYDOWN,0x00000053,0x201f00
01); //Alt-s
    PostMessage(h,WM_KEYDOWN,0x00000053,0x101f0001);
//s

    return FALSE;
}
```

StopListening.cpp

```
// CFindwinApp initialization
BOOL CFindwinApp::InitInstance()
{
    // Standard initialization
    /* If you are not using these features and wish to
reduce the size of your final executable, you should
remove from the following the specific initialization
routines you do not need.*/

    HWND h=FindWindow(NULL,"Realize Voice");

    PostMessage(h,WM_SETFOCUS,(WORD)h,0);
    PostMessage(h,WM_SYSKEYDOWN,0x00000053,0x201f0001); //Alt-s
        PostMessage(h,WM_KEYDOWN,0x00000054,0x10140001); //t

    return FALSE;
}
```

readaddr.java

```
import java.awt.*;
import java.io.*;
import java.awt.event.*;
import java.util.*;

public class readaddr extends Frame implements
Runnable
{
  Graphics g;
  Font f1 = new Font("TimesRoman",
Font.ITALIC,20);
  Font f2 = new Font("TimesRoman", Font.BOLD,28);
  Thread thread;
  int flag = 0;

  public static void main(String[] args)
  {
   readaddr temp = new readaddr();
   temp.setSize(600, 150);
   temp.setVisible(true);
  }
  public readaddr()
  {
   setLayout(null);
   thread = new Thread(this);
   thread.start();
  }

  public void run()
  {
   while (true)
     {
      repaint();
      try{
        Thread.sleep(100);
      }
      catch (InterruptedException e){}
     }
  }

  public void update(Graphics g)
  {
   paint(g);
  }

  public void paint(Graphics g)
  {
   String str = null;
   try{
     FileInputStream fin = new FileInputStream("url.txt");
     DataInputStream din = new DataInputStream(fin);
     str = din.readLine();

     if (str.equals("http://"))
       {
         if (flag == 1)
             System.exit(-1);
       }else{
        flag = 1;
       int i = str.length();
       str = str.substring(0,i-1);
       }

     din.close();
     fin.close();
   }
   catch (Exception e){}

   g.setFont(f1);
   g.setColor(Color.black);
   g.drawString("MYMAR GLOBAL INTERNET",200,50);
   g.setFont(f2);
   g.setColor(Color.blue);
   g.drawString(str,125,100);
   g.dispose();
  }
}
```

Appendix B ZRP Routing Program

mydefine.h

```
/* file mydefine.h */
/* define short name for unsigned interger */

#ifndef MYDEFINE
      #define MYDEFINE

      #include <trace.h>

      //type define unsigned for short
      typedef unsigned long ul;
      typedef unsigned int ui;
      typedef unsigned short us;
      typedef unsigned char uc;

      #define SP "  " //4 space
#endif
```

UTILITY.H

```
#ifndef UTILITY_H
#define UTILITY_H

#include <mydefine.h>
// Utilities about timer function in time.cpp
//get current time in unit 10ms
unsigned short timeNow();
//compare two time, check time to live value
// return less than 0 is time out
short timeTTL(const unsigned short ex,const unsigned
short now);

//send int/short data in big margain format
short putByte( uc* &p,ui in,int size);
ui getByte( uc* &p,int size);

void waitfor(bool& b);
void sleep(ui i);

#endif   //ndefine UTILITY
```

entry.cpp

```
/*  file entry.cpp  */
/*  simulate socket  */
#include "zrp.h"

short putByte( uc* &p,ui in,int size)
{     p+=size;
      uc* ptmp=p;
      ui tmp=0x000000ff;
      for(;size--;in>>=8)
      *--ptmp=(uc)(tmp&in);
      return size;
}

ui getByte( uc* &p,int size)
{     ui in=*p++;
      for(;--size;)
            in=(in<<8)+(*p++);
      return in;
}
```

time.cpp

```
/*  file time.cpp  */
/*  utilties deal with timing  */
#include <iostream.h>
#include <sys/timeb.h>

#ifdef _WIN32
      typedef  struct _timeb tb;
      #define ft(x) _ftime(x)

      //simulate delay in WIN32 system
      void usleep(unsigned i)
      {     for(unsigned j=0;j<i;j++) for(int k=0;k++>0;);}
#else
   #include <unistd.h>
      typedef struct timeb tb;
      #define ft(x) ftime(x)
#endif

//wait for other process to finish operate in table(unlocked)
void waitfor(bool& b) { while(b) usleep(100);}

//#include <time.h>
extern const unsigned short timeUnit;        //time unit 10ms

//return current time, within 600s=10minute format
unsigned short timeNow()
{
  tb timebuffer;
  unsigned short tmp=1000/timeUnit;

  ft( &timebuffer );
  return (timebuffer.time /*%
(60000/tmp)*/)*tmp+timebuffer.millitm/timeUnit;
}

//check living time, when t1>=t2 return TTL
short timeTTL(unsigned short expire,unsigned short now) {
      //cout<<endl<<expire<<"e-n"<<now;
      const unsigned short module=4096;
      expire=expire%module;
      now=now%module;
      short tmp=expire-now;
      if(tmp>=0)
            if(tmp>module/2) return tmp-module;
            else return tmp;
      else {
            tmp*=-1;
            if(tmp>module/2) return module-tmp;
            else return (-1)*tmp;
      }
}
```

zrp.h

```
/* File zrp.h */
/* defclare zrp protocol and parameter */
#ifndef ZRP_H
#define ZRP_H

#include <mydefine.h>

extern ui thisNode;

const us MaxPayload=2047;
enum  msgType {Error,IARPmsg,IERPmsg,Data} ;
const ui BROADCAST=0xffffffff;//broadcasting
address
const us MAXCONNECTIONS =16;
            //th emax number of connection can create

//interface fuction between higher layer and ZRP
typedef msgType connectDescriptor;
connectDescriptor openConnection(ui destinatin);
            //open connection to the destination
short send(connectDescriptor c,uc* p,ui d,us length);
            //send data using connection c
short receive(connectDescriptor c,uc* buffer,us length);
//receive data using connection c

//interface function between ZRP and MAC layer
short sendData(uc*& pData,ui nNode,us payload,us
priority);
short  receiveData(); //receive apcket from MAC layer

#endif  //ndefine ZRP
```

zrp.cpp

```
/*   file zrp.cpp  */
/*   zrp protocol  */

#include "zrp.h"
#include "iarp.h"
#include "utility.h"
#include "ierp.h"

extern iarp IARP;
extern ierp IERP;

//send out one entry of route table
//return the bytes sent, pointer move forward
short routeEntry::put(uc* &p,short timer)
{           //trace2("send node",node);
            putByte(p,node,sizeof(node));
            us tmp=(dist+1)<<12;
            tmp+=timer;             //send timestamp
            putByte(p,tmp,sizeof(tmp));
            return 6;  //size of each entry
}

//receive one entry from IARP update data
short routeEntry::get(uc*& p,short now)
{           node=getByte(p,sizeof(node));
            us tmp=(us) getByte(p,sizeof(tmp));
            dist=tmp>>12;
            timestamp=now+tmp&0x0fff;
            return 6;
}
```

```
//convert data to iarp header format
void IARPHeader::getHeader(uc* p)
{     sAddr=getByte(p,sizeof(sAddr));
      dAddr=getByte(p,sizeof(dAddr));
      nNode=getByte(p,sizeof(nNode));

      us tmp=getByte(p,sizeof(tmp));
      priority=(tmp&0x0000007);
      tmp=tmp>>3;
      msgtype=(tmp&0x0000003);
      payload=tmp>>2;
      crc=getByte(p,sizeof(crc));
}

//forming packet, put header info into data packet
uc* IARPHeader::putHeader(uc*& p)
{     putByte(p,sAddr,sizeof(sAddr));        //source address
      putByte(p,dAddr,sizeof(dAddr));
      //destination address
      putByte(p,nNode,sizeof(nNode));                    //next node
      us tmp=payload;//payload length 0--2k
      tmp=(tmp<<2)+msgtype;
      tmp=(tmp<<3)+priority;
      putByte(p,tmp,sizeof(tmp));  //next interger
      putByte(p,crc,sizeof(crc));  //crc

      return p;
}

//forward data packet to next node
short IARPHeader::sendToNext(uc* pData)
{     //get next node address
      if(this->dAddr==BROADCAST)
            //is broadcast to all node
          this->nNode=BROADCAST;
      else if
      ((this->nNode=IARP.getNextNode(this->dAddr))<=0)
      {
            cerr<<"Cannot find destination in routing table"
                  <<endl;
            return -1;
            //can't find next node of path to destination
      }

      //change next node address in packet header
      uc* ptmp=pData+8;
      putByte(ptmp,this->nNode,sizeof(nNode));

      trace3(thisNode," iarp sending/forwarding data to Next
node:",this->nNode);
      return sendData(pData,this->nNode,this->payload+this-
>length,this->priority);
}
```

interface.cpp

```
/*  file interface.cpp  */
/*  simulate socket interface send(), receive()  */

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include "zrp.h"
#include "iarp.h"
#include "ierp.h"

char* sendFile="send.dat";
char* receiveFile="send.dat"; //"receive.dat"
```

```
extern iarp IARP;
extern ierp IERP;

//send data ZRP packet to lower layer
short sendData(uc*& pData,ui nNode,us length,us
priority)
{   trace3("sending file and data
size:",sendFile.length);

    ::ofstream out(sendFile,ios::binary|ios::out);
        if(!out) {
        cerr<<"Open file "<<sendFile<<"
error "<<endl;
        exit(-1);
    }

    out.write(pData,length);
    out.close();

    //delete[] pData;

    return 0;
}

//receive zrp packet from lower layer (MAC)
short receiveData()
{
    ::ifstream in(receiveFile,ios::binary|ios::in);
    if(!in) {
        cerr<<"Open file "<<receiveFile<<"
error."<<endl;
        exit(-1);
    }

    in.seekg(0,ios::beg); //begin of file position
    ::streampos pos=in.tellg();
    in.seekg(0,ios::end); //end of file position
    int length=(int)(in.tellg()-pos);
    in.seekg(pos);

    trace3("receiving file and data
size:",receiveFile.length);

    //assigned memory for data
    uc *pData=new uc[length];
    in.read(pData,length);

    IARP.processReceive(pData);

    delete pData;
    return 0;
}

//open connection to the destination
//return connection descriptor
connectDescriptor openConnection(ui dest)
{
    if(IARP.isLocal(dest)) //destination wihtin local
zone
        return IARPmsg;
    else if(IERP.query(dest)>=0)
        return IERPmsg;
    else
        return Error;
}

//send data using zrp protocol
short send(connectDescriptor c,uc* p,ui dest,us length)
{
    if(c==IARPmsg) //local data packet
        return IARP.processSend(p,dest,length,Data);
    else if(c==IERPmsg)          //outer zone data packet
        return IERP.processSend(p,dest,length,Data);
    else
        cerr<<"Unknown connection type"<<endl;
    return -1;
}

//receive data from ZRP
short receive(connectDescriptor c,uc* buffer,us length)
{   //for(us i=0;i<length;i++) cout<<*buffer++;
    trace2("High-layer receive data length ",length);
    return 0;
}
```

IARP.H

```
#ifndef IARP_H
#define IARP_H

#include <mydefine.h>

#define SORT //sorted RT table
//#define unsort        //unsorted RT table
#define VECT  //vector storage
//#define LISTS // list storage

#ifdef LISTS
    #include <list>
#endif
#ifdef VECT
    #include <vector>
#endif
using namespace std;

extern const unsigned short timeUnit;          //time unit 10ms
const us UpdateInterval= 50;   //update routing eaery 500ms
const us TTL=UpdateInterval/timeUnit*10;
    //TTL of entry, max value 12bits=4095
const us Radius=2;

//structure of each node in routing table
struct routeEntry {
    ui node;//node address
    ui nextNode;       //next Node path to address
    us dist:4;                    //distance of hops(0_15)
    us timestamp:12; //next update time

    short put(uc*& p,short ttl);
    //send out one entry of route table
    short get(uc*& p,short ttl);
     //get entry from IARP packets data

    friend bool inline operator< (const routeEntry& a,const
routeEntry& b) {return a.node<b.node;};
    friend bool operator> (const routeEntry&,const routeEntry&);
    friend bool operator== (const routeEntry&,const
routeEntry&);
    friend bool operator!= (const routeEntry&,const routeEntry&);

    void print(){
cout<<node<<SP<<nextNode<<SP<<dist<<SP<<timestamp<<end
l;};
};

//iarp header
struct IARPHeader {
        ui sAddr;              //source address
```

```
    ui dAddr;                   //destination address
    ui nNode;                   //next node
    us payload:11;   //payload length 0~2k
    us msgtype:2;    //message type
    us priority:3;   //priority of packet to send
    us crc;

    uc length;                  //the length of header
    IARPHeader(){length=16;};
    short sendToNext(uc* pData);
    //send packet(include header) to next node
    uc* putHeader(uc*& p);
    //put header info into packet
    void getHeader(uc*);
    //get header structure from packet
    //short process(uc*&);
    //zrp process packet.
        //send data only to higher layer
};

#ifdef VECT
    typedef vector<routeEntry> RTtable ;
#elif defined LISTS
    typedef list<routeEntry> RTtable ;
#endif

class iarp {
public:
    static RTtable routeTable;
    static bool routeTableLocked;

    void updateTable(routeEntry item);
    //update the routing entry in the table. if
    //not there yet -- insert, otherwise update if
    bool isLocal(ui dest);
    //Is the destination within this zone?
    ui getNextNode(ui dest);
    //return the next node address to the destination
    void checkTTL();
    //delete timeout entry in routing table
    void printTable();
    //utility to print out routing table to STDOUT

    iarp(); //constructor
    ~iarp(); //distructor

    void processUpdate(uc* data);
    //received a iarp packet
    void sendUpdate();
    //process zone routing table update

    //send data to destination
    short processSend(uc* p,ui destination, us length,
msgType type=Data);
    short processReceive(uc* p);

private:
    //search if the item consisting a destination
        //is within the table
    RTtable::iterator search(routeEntry& item);
    //recursive delete an entry in table
    RTtable::iterator deleteEntry(RTtable::iterator
iItem);
    void insert(RTtable::iterator,routeEntry&);
        //insert an entry in position
};

class updateMsg {
public:
 ui source;  //source address
 ui timestamp:12;   // living time
 ui entries:12;   //number of entries
 uc *entry;

 us inline sourceSize(){return 7;}; //size of source info
 us inline entrySize() {return 6;};
        //size of each entry in message

 void get(uc* data);
 uc* put(uc* p,us n) ;
};

#endif //ndefine IARP
```

iarp.cpp

```
/*  file iarp.cpp  */
/*  iarp protocol  */

#include "zrp.h"
#include "iarp.h"
#include "ierp.h"
#include "utility.h"
#include <algorithm>

extern iarp IARP;
extern ierp IERP;

    //initialize static member rout table
    RTtable iarp::routeTable;
    bool iarp::routeTableLocked=false;

//send data using IARP
short iarp::processSend(uc* p,ui destination, us length, msgType
type)
{   trace2("iarp process send data to",destination);

    if(length>MaxPayload) {  //check payload size
        cerr<<"Too big data packet"<<endl;
        return -2;
    }

    IARPHeader h;
    h.sAddr=thisNode,h.dAddr=destination;
    h.msgtype=type;
    h.payload=length;

    //set priority
    h.priority=0x7-type;

    //locate memory for packet
    uc* pData=new uc[h.length+length];
    if(pData==NULL) {
      cerr<<"Cannot locate memory for data"<<endl;
        return -1;
    }

    //put data
    uc* ptmp=pData;
    h.putHeader(ptmp);          //add header
    for(int i=0;i<length;i++)
        *ptmp++=*p++;

    short tmp=h.sendToNext(pData);
        //send data to next node

    delete pData;
    return tmp;
}
```

```
//iarp receive a packet
short iarp::processReceive(uc* p)
{    trace2(thisNode," IARP receive a packet");

     IARPHeader h;
     h.getHeader(p);//get header. pointer not move

     //check if i am the receiver
     if(h.nNode!=thisNode&&h.nNode!=BROADCAST
){
          cerr<<"Error. not receiver of the
packet"<<endl;
          return -1;
     }

   //process the data
     if(h.dAddr==thisNode||h.dAddr==BROADCAST){
     //is the destination of packet
       switch(h.msgtype) {
        case Error:
               //control process
              break;
       case IARPmsg:
              IARP.processUpdate(p+h.length);
              //IARP update process
              break;
       case IERPmsg:
              IERP.process(p+h.length);
              //IERP message type
              break;
       case Data:  //send data to higher layer
               receive(IARPmsg,p+h.length,h.payload);
              break;
        default:
              break;
         } //end of switch
     return 0;
     } else      //not destination of the packet
      return h.sendToNext(p);
       // forwarding packet to destination via next node
}

//routine to send routing table update message
void iarp::sendUpdate()
{    updateMsg h;
     // memory to put data
     us length=
     (routeTable.size()*h.entrySize()+h.sourceSize());
     if(length>MaxPayload) length=MaxPayload;
     //select the min packet size
     uc* pData=new uc[length];
     if(pData==NULL)
cerr<<__FILE__<<__LINE__<<
          "Don't have enough memory"<<endl;

     us radius_1=Radius-1;
     //radius use in distance compare
     us MaxNumber=MaxPayload/h.entrySize()-1;
     //max number of entry per packets (12 bit)

     RTtable::iterator it=routeTable.begin();
     do {
          us number=0;              //number of entries
          length=h.sourceSize();
          //initial actually size of data packet
          uc* ptmp=pData+length;  //advance to enntry
          waitfor(routeTableLocked);
          routeTableLocked=true;
          unsigned int tmpTime=timeNow();
          //get current time.
          for(;it!=routeTable.end()&&number<MaxNumber;)
           {
               short ttl=timeTTL((*it).timestamp,tmpTime);
               if(ttl<=0) //out of update time
                         it=deleteEntry(it);
                          //iterate to next after delete
               else if ((*it).dist>radius_1)       //?border node
                         it++;          //next entry
               else {                   //send
                         (*it++).put(ptmp,ttl);
                         number++;
                         length+=h.entrySize();
               }         //endif
          }    //endfor

       trace3(thisNode,"iarp form upate() entries:",number);

          routeTableLocked=false;
          //put source info
          ptmp=pData;
          h.put(ptmp,number);

          //send update msg
          this->processSend(
               pData,BROADCAST,length,IARPmsg);
       } while(it!=routeTable.end()); //end while

       delete pData;
}

void iarp::processUpdate(uc* pData)
{    //get header(source) info
 updateMsg h;
 h.get(pData);

 trace4(thisNode,"receive IARP table update
from/entries:",h.source,h.entries);

 //update entry contain source info
 ui now=timeNow();
 routeEntry tmpEntry={h.source,h.source,1,now+h.timestamp};
 updateTable(tmpEntry);

 //update routing table
 uc* ptmp=h.entry;
 while(h.entries-->0) {
  tmpEntry.get(ptmp,now); //convert data to struct entry

    //discard info about this node itself
    if(tmpEntry.node==thisNode) continue;

  updateTable(tmpEntry);//update table
  } //end while
}

iarp::iarp() { };//set update timer
iarp::~iarp() { };

void updateMsg::get(uc* pData)
{
 source=getByte(pData,sizeof(source));
 ui tmp=getByte(pData,3);  //cout<<"receive "<<tmp<<endl;
 timestamp=tmp & 0x0fff;
 entries=tmp>>12;
 entry=pData;
}
```

```
uc* updateMsg::put(uc* p,us n)
{   uc* ptmp=p;
    putByte(ptmp,thisNode,sizeof(thisNode));
    ui tmp=n;
    tmp=(tmp<<12)+TTL;  //cout<<"send
"<<tmp<<endl;
    putByte(ptmp,tmp,3);
    return p;
}


void iarp::printTable()  //recursive delete
{
        waitfor(routeTableLocked);
        routeTableLocked=true;
        RTtable::iterator
it=routeTable.begin(),ie=routeTable.end();
        for(;it!=ie;it++)
            (*it).print();
        routeTableLocked=false;
}


RTtable::iterator iarp::deleteEntry(RTtable::iterator
iItem)  //recursive delete
{   routeEntry deleted=*iItem;
    RTtable::iterator tem=routeTable.erase(iItem);
    //discard deleting node

    //delete whose next-node is this deleting node
    if(deleted.dist==1) {
    //only when this deleting node is neighbor
        RTtable::iterator it=routeTable.begin();
        for(;it!=routeTable.end();) {
            if((*it).nextNode==deleted.node)
                    it=deleteEntry(it);
            else it++;
        }
    }
    return tem;
}

void iarp::updateTable(routeEntry item)
{   //trace2("updating node",item.node);
    waitfor(routeTableLocked);
    routeTableLocked=true;

 RTtable::iterator
ib=routeTable.begin(),ie=routeTable.end();
 ib=search(item);
 if(ib==ie||(*ib).node!=item.node)   //not there yet ---
insert
   insert(ib,item);
 else if(timeTTL(item.timestamp,(*ib).timestamp)>10)
{
     //new item has more time to live
     if((*ib).dist==1&&item.dist>1) {
        //drop who cannot work as next node any more
        deleteEntry(ib);
        ib=search(item); //search position again
       insert(ib,item);
     }
     else
        *ib=item;       //update
   }
 else if(item.dist<(*ib).dist)      //in table,if  better
distance
   *ib=item;       //update
 routeTableLocked=false;
}

ui iarp::getNextNode(ui dest)
{   waitfor(routeTableLocked);
    routeTableLocked=true;

 RTtable::iterator ib=routeTable.begin(),ie=routeTable.end();
 routeEntry item={dest,1,1,1};
    //construct an entry include dest for compare
 ib=search(item);
 ui tmp;
 if(ib==ie||(*ib).node!=dest)  tmp=0;    //not there yet
 else       //return next node address
  tmp=(*ib).nextNode;

 routeTableLocked=false;
 return tmp;
}

//search if destination is with in routing table -- local zone
bool iarp::isLocal(ui dest)
{   waitfor(routeTableLocked);
    routeTableLocked=true;

 routeEntry item={dest,1,1,1};
    //construct an entry include dest for compare
 RTtable::iterator ib=routeTable.begin(),ie=routeTable.end();

 ib=search(item);
   //return the position or where supposed to be if not found

 routeTableLocked=false;
 return ((ib!=ie)&&(*ib).node==dest);
  //ib==ie; not there yet, return false;
            // (*ib).node==dest is within this zone
}

//search if destination is with in routing table -- local zone
//return the position or where supposed to be if not found
RTtable::iterator iarp::search(routeEntry& item)
{
 RTtable::iterator ib=routeTable.begin(),ie=routeTable.end();
#ifdef unsort            //For unsorted
 for(;(ib!=ie)&&((*ib).node!=item.node);ib++);
#elif defined SORT//For sorted
#ifdef VECT
 ib=lower_bound(ib,ie,item);//vector
#elif defined LISTS
 for(;(ib!=ie)&&((*ib).node<item.node);ib++);
#endif
#endif
 return ib;   //ib: pointer to the entry or
              //where it supposed to be
}

//insert an entry in position ib
void iarp::insert(RTtable::iterator ib,routeEntry& item)
{
#ifdef unsort            //for unsorted
  routeTable.push_back(item);
#else
    #ifdef SORT      //For sorted
    routeTable.insert(ib,item);
    #endif
#endif
}

//delete timeout entry in routing table
void iarp::checkTTL()
{   trace("check timeout of routing table");

    us now=timeNow();
```

```
    waitfor(routeTableLocked);
    routeTableLocked=true;

  RTtable::iterator
ib=routeTable.begin(),ie=routeTable.end();
  for(;(ib!=ie);ib++)
      if(timeTTL((*ib).timestamp,now)<=0)
          deleteEntry(ib);

  routeTableLocked=false;
}
```

IERP.H

```
#ifndef IERP_H
#define IERP_H

#include <mydefine.h>
#include "iarp.h"

#include <list>
using namespace std;

const us QueryTTL=60000/timeUnit;//60s of query
time out
const us RouteTTL=60000/timeUnit;
    //60s of routing keep in query table
const us EtTTL=60000/timeUnit;
    //60s of et keep in cache time out
const uc MaxZoneCount=127;

enum IERPmsgType {IERPControl, Query, Reply,
IERPData};

struct IERPHeader {
    ui sAddr;                    //source address
    ui dAddr;                    //destination address
    ui sender;          //previous sender node address
    uc zoneCount;
    //the number of zones. For IERP query message
    // it's TTL of query in zones.  For others, it's
    // the unmber of entries in the followed routing
path.
    us totalLength:14;//total length of IERP packet
    us msgtype:2;     //message type of IERP message
    us headerLength:12;          //the size of header
only
    us hopCount:4;
        //the number of hops, to determine border
node.
    us headerCRC;    //chech sum of header

    ui inline pathOffset() { return 20; }
        //routing path offset in the header

    void query(ui d);  //creat query header stucture
    void reply(uc* p);//creat reply header stucture
    // form data packet using IERP protocol
    short data(uc* p, ui destination, us length);
    void put(uc* p);   //put header into data packet
    void get(uc* p);   //get header from data packet
    void send(uc* p);// send data after get routing table

    void processQuery(uc* p);//process query message
    void processReply(uc* p);//process a reply message
    void processData (uc* p);//process a data  message
};
```

```
struct queryEntry {
    ui dAddr;        //destination address
    uc* pRoutPath;   //pointer point to routign path
    us entries;      //the number of entries in routing path
    us timeOut;      //this query is timeout at the time

    friend bool operator< (const queryEntry&,const queryEntry&);
    friend bool operator> (const queryEntry&,const queryEntry&);
    friend bool operator== (const queryEntry&,const
queryEntry&);
    friend bool operator!= (const queryEntry&,const
queryEntry&);

};

struct etEntry {        //structure for store query info for early
detection
    ui sAddr;        //source address
    ui dAddr;        //destination address
    ui sender;       //previous sender node address
    us timeOut;      //timeout for this entry to stay in cache

    friend bool operator< (const etEntry&,const etEntry&);
    friend bool operator> (const etEntry&,const etEntry&);
    friend bool operator== (const etEntry&,const etEntry&);
    friend bool operator!= (const etEntry&,const etEntry&);

};

class ierp {
public:
    static list<queryEntry> queryTable;
    static list<etEntry>   etTable;
    static bool queryTableLocked;
    static bool etTableLocked;

    ui isEtDetect(const ui s, const ui d);
        //return the sender if item already in etTable
    //determine if entry containing (destination) in Table
    bool isQueried(const ui d);
    //get routing path to the determine if entry in queryTable
    uc* getPath(const ui d, volatile uc& n);
    //put routing path to the determine in queryTable
    uc* putPath(const ui d, const uc* p,const uc n);
    //delete the entry containing (destination) in queryTable
    void deleteQuery(const ui d);
    //delete timeout entry in query and et table
    void checkTTL();

    short process(uc* p);//process a IERP packet
    //send IERP packet data to destination
    short processSend(uc* p,ui destination, us length, msgType
type=Data);
    short query(ui destination);//send IERP query

};
#endif  //ifndefine IERP_H
```

ierp.cpp

```
/*  file ierp.cpp  */
/*  IERP protocol  */

#include "zrp.h"
#include "ierp.h"
#include "utility.h"
```

```
extern ui thisNode;
extern iarp IARP;
extern ierp IERP;

//initialize static member et table and query table
list<queryEntry> ierp::queryTable;
list<etEntry>   ierp::etTable;
bool ierp::queryTableLocked=false;
bool ierp::etTableLocked=false;

//process a IERP packet
short ierp::process(uc* p)
{   trace("ierp process received packet");

    IERPHeader h;
    h.get(p);
    switch(h.msgtype){
    case IERPControl:
        break;
    case Query:
        h.processQuery(p);
        break;
    case Reply:
        h.processReply(p);
        break;
    case IERPData:
        h.processData(p);
        break;
    default: break;
    };
    return 0;
}

//FORM IERP query
short ierp::query(ui destination)
{   trace("create a IERP query message");

    if(isQueried(destination)) return 0;
     //already queried or has routing in table

    //creat query header
    IERPHeader h;
    h.query(destination);  //creat query message
    //creat query packet
    uc* pQ=new uc[h.totalLength];
    h.put(pQ);
    us tm=timeNow();

    //record query info into query.ET table
    queryEntry tmpQ=
        {destination,NULL,0,QueryTTL+tm};
    waitfor(queryTableLocked);
    queryTableLocked=true;
    queryTable.push_back(tmpQ);
    queryTableLocked=false;

    etEntry tmpE={thisNode.destination,0,EtTTL+tm};
    waitfor(etTableLocked);
    etTableLocked=true;
    etTable.push_back(tmpE);
    etTableLocked=false;

    //send query to ZRP
    IARP.processSend(pQ,BROADCAST,h.totalLengt
h,IERPmsg);
    delete pQ;
    //wait for reply

    return 1;
}

//process a query message
void IERPHeader::processQuery(uc* p)
{   trace("it's a query message");

    if(IERP.isEtDetect(this->sAddr,this->dAddr)>0){
        //Early Detected
        trace("early detected");
        return; }       //don't need further process
            //record query info for ET

    etEntry item={sAddr,dAddr,sender,EtTTL+timeNow()};
    waitfor(IERP.etTableLocked);
    IERP.etTableLocked=true;
    IERP.etTable.push_back(item);
    IERP.etTableLocked=false;

    if(this->hopCount-->1) {
        //if hopCount!=0, not reach border node
        this->put(p);  //modify header in meaasge
        IARP.processSend(p,BROADCAST,this-
>totalLength,IERPmsg);//forwarding query
    } else {  //hopCount==0, query reach border node
            if(IARP.isLocal(this->dAddr))
                    //destination with this zone
                this->reply(p);
                    //send reply message to source
            else if(this->zoneCount--) {
                //zoneCount>0, not reach zones limited
                    this->sender=thisNode;
                    this->hopCount=Radius;
                    this->put(p);
                    ////modify header in meaasge
                    //send new IERP packet
                    IARP.processSend(p,BROADCAST,this-
>totalLength,IERPmsg);
                }
                //else zoneCount reach limited,
                //don't forwarding query anymore
    } //end if hopCount
}


//determine if entry containing (source,destination) in etTable
//if is Early Detacted, return sender address, ow return 0;
ui ierp::isEtDetect(const ui s,const ui d)
{   waitfor(etTableLocked);
    etTableLocked=true;
    list<etEntry>::iterator ib=etTable.begin(),
        ie=etTable.end();
    for(;ib!=ie&&(*ib).sAddr!=s&&(*ib).dAddr!=d;ib++);
    ui tmp;
    if(ib==ie)
     tmp=0;   //ib==ie;not there yet -----isnot detect
    else
        tmp=(*ib).sender;
            //ib!=ie; in table -----is detected
    etTableLocked=false;
    return tmp;
}


//determine if the entry containing (destination) is in queryTable
bool ierp::isQueried(const ui d)
{   waitfor(queryTableLocked);
    queryTableLocked=true;

    list<queryEntry>::iterator ib=queryTable.begin(),
        ie=queryTable.end();
    for(;ib!=ie&&(*ib).dAddr!=d;ib++);
```

```
    queryTableLocked=false;
    return (ib!=ie);    //ib!=ie: in table -----is queried
            //ib==ie:not there yet -----isnot queried
}


//delete the entry containing (destination) is in
queryTable
void ierp::deleteQuery(const ui d)
{   waitfor(queryTableLocked);
    queryTableLocked=true;

    list<queryEntry>::iterator
ib=queryTable.begin(),ie=queryTable.end();
    for(;ib!=ie&&(*ib).dAddr!=d;ib++);
    if(ib!=ie) {//ib!=ie: in table -----is queried
        if((*ib).pRoutPath!=NULL)
            delete (*ib).pRoutPath;
        queryTable.erase(ib);
    }               //ib==ie:not there yet -----isnot
queried
    queryTableLocked=false;
}

//get routing path to the determine if the entry is in
queryTable
uc* ierp::getPath(const ui d, volatile uc&  n)
{   waitfor(queryTableLocked);
    queryTableLocked=true;

    list<queryEntry>::iterator
ib=queryTable.begin(),ie=queryTable.end();
    for(;ib!=ie&&(*ib).dAddr!=d;ib++);
    uc* p;
    if(ib!=ie) {        //ib!=ie: in table -----is queried
        p=(*ib).pRoutPath.n=(*ib).entries;
    }else {  //ib==ie:not there yet -----isnot queried
        p=NULL,n=0;
    }
    queryTableLocked=false;
    return p;

}

//put routing path to the determine in queryTable
uc* ierp::putPath(const ui d, const uc* p,const uc n)
{   waitfor(queryTableLocked);
    queryTableLocked=true;

    list<queryEntry>::iterator
ib=queryTable.begin(),ie=queryTable.end();
    for(;ib!=ie&&((*ib).dAddr!=d);ib++);
    uc* pPath=NULL;
    if(ib!=ie) {        //ib!=ie: in table -----is queried

    if(((*ib).pRoutPath==NULL)||((*ib).entries>=n)){
        //empty or not shorter path

            trace("Update IERP routing path in table");
            if((*ib).pRoutPath!=NULL)
                    delete (*ib).pRoutPath;//free old
path
            (*ib).entries=n,(*ib).timeOut=RouteTTL;
            us leng=sizeof(thisNode)*n;
            pPath=new uc[leng];
            uc* ptmp=pPath;
            for(int i=0;i<leng;i++)
                    *ptmp++=*p++;
                    //put routing path in memory
            (*ib).pRoutPath=pPath;
        };
    }; //else //ib==ie:not there yet -----isnot queried
    queryTableLocked=false;
    return pPath;
}


//delete timeout entry in query/et table
void ierp::checkTTL()
{   trace("check timeout on query table& et table");

    us now=timeNow();

    waitfor(queryTableLocked);
    queryTableLocked=true;

 list<queryEntry>::iterator
ib=queryTable.begin(),ie=queryTable.end();
 for(;(ib!=ie);ib++)
     if(timeTTL((*ib).timeOut,now)<=0)
         queryTable.erase(ib);

 queryTableLocked=false;

 waitfor(etTableLocked);
   etTableLocked=true;

 list<etEntry>::iterator ib1=etTable.begin(),ie1=etTable.end();
 for(;(ib1!=ie1);ib1++)
     if(timeTTL((*ib1).timeOut,now)<=0)
         etTable.erase(ib1);

 etTableLocked=false;
}


void IERPHeader::put(uc* p)
{   putByte(p,sAddr,sizeof(sAddr));
    putByte(p,dAddr,sizeof(dAddr));
    putByte(p,sender,sizeof(sender));
    putByte(p,zoneCount,1);
    us tmp=totalLength;
    tmp=(tmp<<2)+msgtype;
    putByte(p,tmp,sizeof(us));
    tmp=headerLength;
    tmp=(tmp<<4)+hopCount;
    putByte(p,tmp,sizeof(us));
}

void IERPHeader::get(uc* p)
{   sAddr=getByte(p,sizeof(sAddr));
    dAddr=getByte(p,sizeof(dAddr));
    sender=getByte(p,sizeof(sender));
    zoneCount=getByte(p,1);
    us tmp=getByte(p,sizeof(us));
    msgtype=tmp&0x0003;
    totalLength=tmp>>2;
    tmp=getByte(p,sizeof(us));
    hopCount=tmp&0x000f;
    headerLength=tmp>>4;
}

// create query messages
void IERPHeader::query(ui d)
{   msgtype=Query;  //message type of IERP message
    sAddr=sender=thisNode;                  //source address
    dAddr=d;
    //destination address
```

```
    zoneCount=MaxZoneCount; // For IERP query
message.
                    // it's TTL of query in zones.
    hopCount=Radius;//header length of IERP packet
    totalLength=headerLength=20;
    //header length of IERP packet
}

// create reply messages
void IERPHeader::reply(uc* p)
{   trace2("create a reply message to ",sAddr);

    msgtype=Reply;  //message type of IERP message
    zoneCount=2;
    // reset zoneCount(path length) for reply message.

    totalLength=(headerLength+=2*sizeof(sAddr));
     //add two node address in packet size
    //create new data packet
    uc* pData=new uc[totalLength];
    this->put(pData); //put new header into packet

    uc* ptmp=pData+pathOffset();
    //advance pointer to routing path position
    putByte(ptmp,thisNode,sizeof(thisNode));
     //accumulate routing path
    putByte(ptmp,dAddr, sizeof(dAddr));

    IARP.processSend(pData,this->sender,this-
>totalLength,IERPmsg);//send new IERP reply packet

    delete pData;
}

// send data packet using IERP protocol
short ierp::processSend(uc* p, ui destination, us
length,msgType type)
{   IERPHeader h;
    return h.data(p,destination,length);
}

// form data packet using IERP protocol
short IERPHeader::data(uc* p, ui destination, us length)
{   trace("create ierp data packet");

    msgtype=IERPData;//message type of IERP
message
    sAddr=thisNode;             //source address
    dAddr=destination;          //destination address

    //get query info
    uc* path=IERP.getPath(dAddr,zoneCount);
    if(path==NULL) {
        cerr<<"Error, cannot find routing to destination
in query table"<<endl;
        return -1;
    }


headerLength=pathOffset()+zoneCount*sizeof(thisNod
e);
    totalLength=headerLength+length;
    //add two node address in packet size
    if(totalLength>MaxPayload) {
        cerr<<"Too big payload in IERP data
packet"<<endl;
        return -2;
    }
    //create new data packet
    uc* pData=new uc[totalLength];
    uc* ptmp=pData+pathOffset();
     //advance pointer to routing path
    for(int i=headerLength-pathOffset();i>0;i--)
        *ptmp++=*path++;        //put the routing path
    ptmp=pData+pathOffset()+sizeof(thisNode);
    sender=getByte(ptmp,sizeof(sender)); //get next_hop
    zoneCount=2;
        // reset zoneCount(path length) for data message.
    ptmp=pData;
    this->put(ptmp);  //put ierp header
    //put data
    ptmp=pData+headerLength;
    for(ui j=0;j>length;j++)
        *ptmp++=*p++;

    //send data packet
    short tmp=IARP.processSend(pData,this->sender,this-
>totalLength,IERPmsg);
            //send new IERP reply packet

    delete pData;
    return tmp;
}


//process reply message
void IERPHeader::processReply(uc* p)
{   trace("process ierp reply message");

    //accumulate path
    totalLength=(headerLength+=sizeof(ui));
     //accumulate node address in packet size
    uc* pData=new uc[totalLength];//create new data packet
    uc* ptmp=pData+pathOffset(); //point to routing table;
    uc* ptmp1=p+pathOffset(); //point to routing table;
    putByte(ptmp,thisNode,sizeof(thisNode));
    //insert this node address
    //copy original path table
    for(int i=headerLength-pathOffset();i>0;i--)
        *ptmp++=*ptmp1++;
    //delete p;
    this->zoneCount++;//increase routing path count
    this->put(pData); //put new header into packet

    if(this->sAddr==thisNode) {
    // reply message return to source
                    //found routing path to destination
        IERP.putPath(this->dAddr,pData+pathOffset(),this-
>zoneCount);
    } else { // not reach destination yet
        if(this->sender=IERP.isEtDetect(sAddr,dAddr)) {
            //search and found for sender in etTable
            this->put(pData);
            //forward reply packet to sender
            IARP.processSend(pData,this->sender,this-
>totalLength,IERPmsg);
        } else
            cerr<<"Can't find sender in etTable to forward
reply"<<endl;
    }

    delete pData;
}

//process Data message
void IERPHeader::processData(uc* p)
{   trace("process ierp data packet");

            if(this->dAddr==thisNode) {
             // data message reach destination
```

```
            //ZRPtoHigherLayer(p+headerLength);

        receive(IERPmsg,p+headerLength,totalLength-
headerLength);
        } else { // not reach destination yet
            uc*
ptmp=p+pathOffset()+zoneCount*sizeof(thisNode);
            //point to next routing path
            this->sender=getByte(ptmp,sizeof(sender));
            //search and found for sender in etTable
            this->zoneCount++;
            //increase routing path count
            this->put(p);

            //forward reply packet to sender
            if(IARP.processSend(p,this->sender,this-
>totalLength,IERPmsg)<0) {
            //send error message to source
            cerr<<"Error while forwarding packet to
"<<this->sender<<endl;
            }
    }
}
```