Effective bootstrapping of Peer-to Peer networks over Mobile Ad-hoc networks

Wei Ding
Louisiana State University and Agricultural and Mechanical College, wding1@lsu.edu

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_dissertations

Part of the Computer Sciences Commons

Recommended Citation
https://digitalcommons.lsu.edu/gradschool_dissertations/3684
EFFECTIVE BOOTSTRAPPING OF PEER-TO-PEER NETWORKS OVER MOBILE AD HOC NETWORKS

A DISSERTATION

Submitted to the Graduate Faculty of the Louisiana State University and Agricultural and Mechanical College in partial fulfillment of the requirements for the degree of Doctor of Philosophy

in

The Department of Computer Science

by

Wei Ding
Undergraduate Diploma, Northeastern University (China), 1988
M.S. University of Science and Technology of China, 1996
August 2006
This dissertation is dedicated to my lovely wife, Poy, and my wonderful daughter, Bingbing. It is a product of their patience.
ACKNOWLEDGEMENTS

First I would like express my gratitude to my parents. They support me spiritually, emotionally, and physically. Their love is unconditional and endless. My father, a distinguished electronic engineer and winner of multiple national professional awards in China, has been an inspiring example of hardworking for all my life. For him, working is almost as essential as air and water. My mother, also achieving high in her career, not only inspired my brother and me, but also taught us various principles and wisdom in life. Any my achievement is considerably attributed to my parents.

It is my honor to have following distinguished professors in my Ph.D. Advisory Committee: Dr. Iyengar from the Department of Computer Science as the major professor, Dr. McMillin from the Department of Economics as the minor professor, Dr. Rinks from the Department of Information Systems and Decision Sciences as dean’s representative, Dr. Karki and Dr. Park form the Department of Computer Science as committee members. They provided helpful suggestions for the dissertation. Dr. Iyengar also helped me in necessary non-academic issues. My sincere appreciation goes to them.

In last three months, my wife and my daughter supported me, tolerated me, and took care of me in everyday life. Along with my parents, they are always my great source of motivation; and they will be my source of motivation in future achievement.
TABLE OF CONTENTS

ACKNOWLEDGEMENTS ........................................................................................................iv

ABSTRACT ........................................................................................................................ix

CHAPTER 1 INTRODUCTION .........................................................................................1
  1.1 Overview ..................................................................................................................1
  1.2 Conceptual Framework ..........................................................................................1
    1.2.1 Structured P2P Networks ..............................................................................2
    1.2.2 Distributed Hash Table ..............................................................................2
    1.2.3 Bootstrapping a Structured P2P Network ..................................................2
    1.2.4 Bootstrapping a Chord over MANETs .......................................................3
    1.2.5 Node Address Automatic Configuration ..................................................3
  1.3 Problem Statement ...............................................................................................4
    1.3.1 Problems with Automatic Address Configuration ....................................4
    1.3.2 Setting up Structured P2P Topology over a MANET .................................5
  1.4 Solutions Provided ...............................................................................................5
    1.4.1 FAPSR for Automatic Address Configuration ..........................................5
    1.4.2 RAN for Building Chord Ring ......................................................................6
  1.5 Contribution of Dissertation ...............................................................................6
  1.6 Organization of Dissertation ...............................................................................7

CHAPTER 2 BACKGROUND SURVEY .........................................................................8
  2.1 Mobile Ad Hoc Networks ....................................................................................8
    2.1.1 Definition and Evolution ............................................................................8
    2.1.2 MANETs Protocol Stack ..........................................................................10
    2.1.3 Routing in MANETs .................................................................................11
      2.1.3.1 Proactive Protocols ...........................................................................12
      2.1.3.2 Reactive Protocols ...........................................................................12
    2.1.4 Bootstrapping in MANETs .........................................................................13
  2.2 Peer-to-Peer overlay networks ............................................................................13
    2.2.1 P2P Paradigm and Architecture ................................................................13
    2.2.2 P2P Systems ..............................................................................................14
    2.2.3 Unstructured P2P Networks .......................................................................15
    2.2.4 Structured P2P Network ...........................................................................16
  2.3 P2P Overlay Networks above MANETs .............................................................19
    2.3.1 Motivation and Reason ..............................................................................19
    2.3.2 Typical Application of P2P Systems/Networks over MANETs .................20
    2.3.3 Guideline for Implementation ....................................................................20
ABSTRACT

Mobile Ad-hoc Networks (MANETs) and Peer-to-Peer (P2P) networks are vigorous, revolutionary communication technologies in the 21st century. They lead the trend of decentralization. Decentralization will ultimately win clients over client/server model, because it gives ordinary network users more control, and stimulates their active participation. It is a determinant factor in shaping the future of networking.

MANETs and P2P networks are very similar in nature. Both are dynamic, distributed. Both use multi-hop broadcast or multicast as major pattern of traffic. Both set up connection by self-organizing and maintain connection by self-healing. Embodying the slogan “networking without networks,” both abandoned traditional client/server model and disclaimed pre-existing infrastructure. However, their status quo levels of real world application are widely divergent. P2P networks are now accountable for about 50 ~ 70% internet traffic, while MANETs are still primarily in the laboratory.

The interesting and confusing phenomenon has sparked considerable research effort to transplant successful approaches from P2P networks into MANETs. While most research in the synergy of P2P networks and MANETs focuses on routing, the network bootstrapping problem remains indispensable for any such transplantation to be realized. The most pivotal problems in bootstrapping are: (1) automatic configuration of nodes addresses and IDs, (2) topology discovery and transformation in different layers and name spaces.

In this dissertation research, we have found novel solutions for these problems. The contributions of this dissertation are: (1) a non-IP, flat address automatic configuration scheme, which integrates lower layer addresses and P2P IDs in application layer and makes simple cryptographical assignment possible. A related paper entitled “Pastry over Ad-Hoc Networks with Automatic Flat Address Configuration” was submitted to Elsevier Journal of Ad Hoc Networks in May. (2) an effective ring topology construction algorithm which builds perfect ring in P2P ID space using only simplest multi-hop unicast or multicast. Upon this ring, popular structured P2P networks like Chord, Pastry could be built with great ease. A related paper entitled “Chord Bootstrapping on MANETs - All Roads lead to Rome” will be ready for submission after defense of the dissertation.
CHAPTER 1 INTRODUCTION

1.1 Overview

It is interesting to notice how research effort and real world application interact in completely different ways for peer-to-peer (P2P) overlay networks\(^1\) and mobile ad hoc networks (MANETs). In former case, successful business applications (if we disregard those notorious legal issues and focus on technical aspects) aroused public interest and initiated active interaction with research community. No doubt, this is healthy and vigorous growth pattern.

In latter case, ardent theoretical research leaded and dominated the whole area for more than a decade; however, no generally acknowledged application except Bluetooth has been developed yet. We do not count IEEE 802.11 — the popular Wireless LAN standard with a cute nickname Wi-Fi, developed by IEEE Working Group 11 — because it actually is not a MANET. In fact Wi-Fi is not directly comparable to MANETs, it is just an physical and MAC layer extension of wired network. To be able to work at normal status, Wi-Fi always needs access points, which are against basic defining rules of MANET.

The malformed resource distribution has set off a warning signal in MANETs community. In addition the inherent homogeneity between P2P networks and MANETs has reminded scholars the propobility of transplantation for a long time. More and more researchers have initiated comparative study and remarkable achievements in the synergy of P2P networks and MANETs — or use another more popular term, P2P over MANETs — has been reflected in publications.

1.2 Conceptual Framework

To fully understand the objective of this dissertation, that is, bootstrapping structured P2P networks over MANETs, following concepts need to be clarified.

\(^1\)Peer-to-peer overlay networks are also called peer-to-peer overlay systems since they may cover whole protocol stack and provide a carrier-like platform for various applications. For convenience, in rest of this dissertation, peer-to-peer overlay networks and peer-to-peer overlay systems are called P2P networks and P2P systems respectively.
1.2.1 Structured P2P Networks

Structured P2P networks use fixed topologies like ring or grid for routing. Nowadays structured P2P overlay networks have become a convenient template paradigm for numerous diverse distributed services. The fundamental abstraction, on which all kinds of applications are based, is key-based routing. Keys are mapped to nodes. Routing a given key is to find the host node responsible for the key. The overlay topology is defined by neighborhood relations specified by local routing tables on all nodes. Both key allocation and key-based routing are via Distributed Hash Table (DHT).

1.2.2 Distributed Hash Table

A DHT distributes data over a structured P2P network with aid of a fixed topology. Basic storage unit and data structure are (key, value) pairs. Node ID is homogeneous to a key. Hence a (key, value) pair could be mapped to a node by hashing the key. Each node is responsible for some section of the key space. The power of a DHT lie in its efficiency to quickly find any given (key, value) pair. The efficiency comes from carefully designed topology structure, node data structure and routing strategies seamlessly integrated into the topology. Basic DHT operations are

- Store(key; value): node ID ← (key; value)
- Locate(key): node ID ← Locate(key)  // Also called Lookup
- Retrieve(key): value ← Retrieve(key)

1.2.3 Bootstrapping a Structured P2P Network

Bootstrapping is the automatic self-organizing procedure to initialize the network and all nodes inside the network, such that the structure P2P network can smoothly start its normal operations. It involves three tasks: (1) node address automatic configuration (assignment); (2) setting up the specific network topology; (3) building node data structures (DHT).

Since node address configuration is performed below the Network layer in ISO/OSI model — i.e. International Standard Organization's Open Systems Interconnection model — the task is not assigned to P2P network.

Comparing to other areas like routing, MAC, and security, bootstrapping (in some literature, synonyms like automatic configuration, self-organization are used with same semantics) has received less attention since very beginning in both MANETs and P2P over MANETs. A common circumvention technique is using idealistic
assumptions, which often do not hold in reality and make application-oriented implementation impossible.

1.2.4 Bootstrapping a Chord over MANETs

This is just a special case of the task described in last section. Chord is probably the most popular structure P2P network. Now the model covers all seven layers in the entire ISO/OSI network model/protocol stack. So we need carry out all three tasks in Section 1.2.3. In most case task (3) building node data structures (DHT) is contained in task (2). Now we have two fundamental problems to solve: (1) node address automatic configuration (assignment); (2) setting up the specific network topology. Chord uses ring topology, so (2) can be refined as: setting up a Chord ring over a MANET.

1.2.5 Node Address Automatic Configuration

As the name implies, node address automatic configuration is the protocol-controlled procedure in which all nodes are assigned unique addresses. Primary element of computer networks is node, which stands for an individual electronic device, a computer in most cases. In the biggest network — Internet, there are hundreds of millions of nodes. To deal with a node among many other nodes, a mechanism is required to reliably distinguish the node from all other nodes. This mechanism is address configuration. Its basic job is to assign a unique address to this node. Only after a unique address is given to all nodes can nodes be accessed and exchange information with each other. Address assignment is foundation of almost all other pivotal network operations, such as routing, sending, receiving, synchronization, unicast, multicast, broadcast, etc.

In wired networks like Internet, via configuring protocols like TCP/IP and data structure like routing table, node address can be set manually and statically by human operator. Typical applications for a MANET are emergence response, battlefield strategic communication, and temporary casual meeting. A MANET is usually set up impromptu without expectation. Therefore the manual approach for address assignment is not appropriate for MANETs because of their dynamic and volatile nature. Automatic, program controlled configuration should be employed for address assignment in MANETs.

As shown in Figure1.1, P2P networks usually sit at upper layers of protocol stack, while MANETs are located at lower layer (generally below transportation layer). A P2P network has its own node address at application (overlay) layer, which is more often called node ID. A MANETs has a MAC address at MAC/Physical layers, and
has routing address at Network layer as well. The address Automatic Configuration problem applies to both MANETs and P2P networks. However, node ID at P2P overlay layer is usually cryptographically hashed according to application. So the problem is more often left for MANETs. The essential question is how to assign unique address to all nodes in a MANET with optimal time complexity, storage complexity, and message complexity.

<table>
<thead>
<tr>
<th>ISO OSI Model</th>
<th>MANETs</th>
<th>P2P Networks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td></td>
<td>Application</td>
</tr>
<tr>
<td>Presentation</td>
<td></td>
<td>Presentation</td>
</tr>
<tr>
<td>Session</td>
<td></td>
<td>Session</td>
</tr>
<tr>
<td>Transport</td>
<td>Transport</td>
<td>Transport</td>
</tr>
<tr>
<td>Network (Routing)</td>
<td>Network (Routing)</td>
<td>Network (Routing)</td>
</tr>
<tr>
<td>Data-link (MAC)</td>
<td>Data-link (MAC)</td>
<td>Data-link (MAC)</td>
</tr>
<tr>
<td>Physical</td>
<td>Physical</td>
<td>Physical</td>
</tr>
</tbody>
</table>

Figure 1.1 MANETs and P2P networks in protocol stacks

1.3 Problem Statement

1.3.1 Problems with Automatic Address Configuration

Historically, MANETs originated from Packet Radio project. After a long time of relative quiescence, MANETs became hot in last decade when TCP/IP has become the dominant standard in networking society. Naturally, most research proposals in MANETs inherited context from IP, sometime unconsciously. However, IP standard, especially IP address scheme, is not compatible with MANETs environment.

Due to hardware constraints such as no bus or ring available for long distance fast transmission, hierarchical MANETs IP address loses its routing functionality in MANETs which made it so successful in wired networks. In MANETs, it becomes a pure identifier, which could be replaced by any other node identifier. All specifications, regulations, and special provisions come along with IP address become cumbersome impediment for performance and efficiency.
1.3.2 Setting up Structured P2P Topology over a MANET

Specific topological structure is essential for structured P2P networks, most controls and data transmission are based upon this structure. After assigning addresses at underlay layer and P2P IDs at overlay layer, the structure should be built before a structured P2P network can work normally. For Chord, it is a ring; Pastry, it is a three plus a ring; for CAN, it is a grid, or to be precise, n dimensional torus.

In P2P community, significant effort has been made to construct initial topology for bootstrapping. Successful approaches like T-Man, T-Chord, and Ring Network has been proposed.

On the other hand, in area of P2P networks over MANETs, no success has been made in spontaneous topology construction in a distributed self-organized manner. Most related recent publication is a paper entitled “Bootstrapping Chord in Ad Hoc Networks - Not Going Anywhere for a While” by Cramer and Fuhrmann [CF2006], which gives a rather gloomy prediction. After careful examining, we found that neither ground nor reasoning is tenable in their arguments. Our analysis is discussed in Section 6.4.2.

1.4 Solutions Provided

1.4.1 FAPSR for Automatic Address Configuration

One of our proposed protocol, Flat Address P2P Source Routing protocol FAPSR suggests a novel non-IP approach to liberate addressing in MANETs from above mention IP constraints. FAPSR discarded IP address, the OSI Network layer address, use Pastry node ID as address for both P2P overlay layer and the underlying Network layer.

The FAPSR protocol has completely liberated peer-to-peer systems in ad-hoc networks from the shackle of old stereotype of IP and IP addresses. It opens up a space for efficient and light-weighted protocols. In this paper, we adopt flat address format for both DSR at network and transport layers and Pastry at application layer. This way we have avoided all complicated IP related problems in systems automatic configuration, routing, and maintenance. Other MANETs routing protocols could be easily adapted to this paradigm as well. Many MANETs routing protocols are compatible with it.
1.4.2 RAN for Building Chord Ring

We propose a complete configuration free self stabilizing protocol Ring Ad-hoc Network (RAN), which we believe is the first successful attempt in the filed of bootstrapping structure P2P networks over MANETs.

RAN has integrated merits from T-Chord, and Ring Network and adapt very well to MANETs. RAN uses only neighbors and local information to build a ring topology in node ID space. Upon this ring, entire Chord protocol could run immediately with optimal configuration at full speed, without any stabilization. RAN includes automatic non-IP address configuration into bootstrapping. Dynamic address configuration is usually deliberately ignored in previous approaches by assuming that an ideal IP address configuration has been a priori established from the very beginning.

1.5 Contribution of Dissertation

In this dissertation, we made following contribution to networking community.

- Systematically analyzed non-IP flat addressing scheme in MANETs
  - Compared pros and cons of non-IP approach to IP approach
  - Demonstrated the advantage of non-IP approach in terms of address collision
  - Pointed out its potential and applications
  - Suggested remedy measures to make up the disadvantages brought up by non-IP approach

- Designed and implemented FAPSR, a complete P2P source routing method based on Pastry ID

- Criticized ungrounded prediction of Cramer and Fuhrmann in [CF2006]

- Thoroughly analyzed solution for the initial topology construction problem in structured P2P networks (systems) over MANETs

- Designed and implemented the first (we believe to best of our knowledge) successful automatic ring construction protocol RAN for structured P2P networks over MANETs, which can be directly applied to Chord without any revision.

- Proved the correctness and effectiveness of RAN by exhaustive simulations

- Pointed out future research direction in bootstrapping structured P2P networks over MANETs
1.6 Organization of Dissertation

Rest of dissertation is organized as follows: Chapter 2 gives background knowledge of our research, that is, MANETs, routing and addressing in MANETs, P2P networks, bootstrapping problem, and structured P2P networks over MANETs; Chapter 3 summarizes related works in automatic addressing in MANETs and topology construction for P2P networks; Chapter 4 analyzes why non-IP addressing scheme is better than IP addressing schemes in MANETs; Chapter 5 introduces our non-IP P2P source routing protocol FAPSR; Chapter 6 discusses problems and solutions on topology construction for structured P2P systems over MANETs; Chapter 7 introduces our ring construction protocol RAN and its simulation; Chapter 8 concludes this dissertation.
CHAPTER 2 BACKGROUND SURVEY

2.1 Mobile Ad Hoc Networks

2.1.1 Definition and Evolution

In English, the word “ad hoc” is defined as “improvised and often impromptu” [Heritage2000], which in networking context implies dynamic, temporary, autonomous, and no fixed infrastructure. Mobile ad hoc networks are distributed communication and computing systems that are consisted of multiple wireless mobile nodes (to use P2P efficiently, it should be in a large scale, which usually has hundreds or more nodes). These nodes switch on or off, move, and make decisions dynamically and independently, and form arbitrary and transitory (so called “ad-hoc”) networks by node discovery and self-organization. Nodes in the formed ad hoc network cooperate to provide variety of network functionalities. The network is maintained by self-healing among these nodes inside the network. No pre-existing communication infrastructure is needed. A physical central controller is absolutely a taboo in MANETs, though a logical central controller is possible in virtual structures like hierarchies, for example, a cluster head. [CCL2003], [RR2002]

Other frequently used terms like pervasive computing, ubiquitous computing, personal computing, are closely related to the concept of ad-hoc networks. Figure 2.1 illustrates the history of MANETs.

At the beginning, the primary application, tactical network, was pure military, which tried to improve battlefield communications. The earliest MANET project was DARPA Packet Radio Network (PRNet) project in 1972 [FL2001], which stemmed from packet switching technology. The nature of dynamic battlefield ruled out the possibility of fixed pre-placed communication infrastructure. Since radio frequency higher than 100 MHz hardly propagate beyond line of sight multi-hop, store-and-forward routing techniques must be employed, which in turn removed the radio coverage limitation and remarkably expanded network geographic area.

PRNET evolved into the Survivable Adaptive Radio Networks (SURAN) program in 1983 to address network scalability, security, processing capability and energy management. In 1980s, a family of advanced network management protocols was developed, and hierarchical network topology based on dynamic clustering was developed to support network scalability. [CCL2003], [Clip2]
Ten years later, the advance of VLSI resulted in microcomputer revolution, plus the breakthrough in radio communication, all these injected tremendous enthusiasm and interest into ad hoc networks. In early 1990s, the US Department of Defense funded the Global Mobile Information Systems (GloMo) and Near-Term Digital Radio (NTDR). [RR2002] The IEEE 802.11 Wireless Local Area Network (WLAN) working group adopted the term “ad hoc networks.” In 1991 the European Telecommunications Standards Institute (ETSI) started standardization of ad hoc networks, that is, the High Performance Radio Local Area Network/1 (HIPERLAN/1), which was released in 1996. In 1997 the first IEEE 802.11 standard was released [Gast2002]. At the same time ETSI formed the Broadband Radio Access Networks (BRAN) Working Group and the MANET Charter to work on HIPERLAN/2 and MANETs routing protocols respectively. In 1998, a special interest group was founded to investigate Bluetooth, a “cable replacement technology,” the first complete implementation in ad hoc networks and the first Bluetooth standard was released in 1999. Later Bluetooth was classified as Personal Area Network (PAN) by IEEE 802.15 PAN Working Group, which worked closely along with Bluetooth SIG. In 1999, IEEE 802.11a and IEEE 802.11b were released. In 2000 The ETSI BRAN Working Group released the first edition of the HIPERLAN/2 standard [BRAN2006]. In 2002 the IEEE 802.15.1 standard and Bluetooth standard 1.1 were released. In 2003 IEEE 802.11g was released. In same
year the IETF MANET charter submitted four Internet Request For Comments (RFC) drafts to the Internet Engineering Steering Group (IESG). [CCL2003], [Clip2]

2.1.2 MANETs Protocol Stack

Chlamtac et al [CCL2003] propose a concise architecture as shown in Figure 2.3. The lowest layer is Enabling Technologies Layer, which correspond to Physical and Data Link layers in ISO OSI seven layers model. The key component in the Data Link Layer is Media Access Control (MAC) component. Networking Layer integrates OSI Network and Transport layers. Most important topic at Network Layer in MANETs is routing problem, which is also most important question in overall research about MANETs. Other issues include addressing, location service for routing, data/node discovery. Key problem in Transport Layer is the implementation of TCP in MANETs environment. Middleware Layer and Application Layer provide interface for network wide applications like service looking up, information sharing and dissemination.

![Figure 2.2 MANET protocol stack](image)

Besides the disjoint hierarchy discussed above, there are many cross-layer issues like energy management to save power, security management, coordination and synchronization, QoS, etc.
MANET protocol stack is very similar to most accepted Internet (famous TCP/IP suite) protocol stack, which is shown in Figure 2.3. Since present P2P networks are built on Internet, the similarity gives us a good reason to believe in the feasibility and effectiveness of overlay networks on MANETs and the proposed transplantation.

### 2.1.3 Routing in MANETs

MANETs routing protocols [RT1999], [RS1998], [CCL2003] are categorized as proactive table-driven protocols and reactive on-demand protocols. Proactive protocols maintain up-to-date routes for every node pair by flooding route updates when a link in path is broken. They stem from Internet distance-vector and link-state protocols. Since common data structure for routing is table, these protocols are termed as table-driven protocols. Well-known proactive protocols are Destination-Sequenced Distance-Vector (DSDV) protocol [PB1994], Clusterhead Gateway Switch Routing (CGSR) protocol [CWLG1997], and Wireless Routing Protocol (WRP) [MG1996].

<table>
<thead>
<tr>
<th>Proactive Protocols</th>
<th>Reactive Protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSDV</td>
<td>LMR</td>
</tr>
<tr>
<td>CGSR</td>
<td>ABR</td>
</tr>
<tr>
<td>WRP</td>
<td>AODV</td>
</tr>
<tr>
<td></td>
<td>DSR</td>
</tr>
<tr>
<td></td>
<td>LMR</td>
</tr>
<tr>
<td></td>
<td>ABR</td>
</tr>
<tr>
<td></td>
<td>TORA</td>
</tr>
<tr>
<td></td>
<td>SSR</td>
</tr>
</tbody>
</table>

On the contrary, reactive on demand protocols set up routes only when there is a demand for it. Route discovery is usually initiated on-demand by source node, the route request is forwarded by the source to the neighbors, and so on, until either the destination or an intermediate node with a fresh route to the destination, are located. An established route is maintained dynamically until the end of the task, i.e. the path tree may change along with the mobility of nodes in the tree. To reduce the overhead, the route between two nodes is discovered only when it is needed. Representative reactive routing protocols include: Dynamic Source Routing (DSR) [JM1996], Ad hoc On Demand Distance Vector (AODV) [PR1999], Lightweight Mobile Routing...
protocol (LMR), Temporally Ordered Routing Algorithm (TORA) [PC1997], Associativity Based Routing (ABR), Signal Stability Routing (SSR). Since reactive protocols have much more overlapping with P2P paradigm, we introduce them in greater detail.

2.1.3.1 Proactive Protocols

DSDV [PB1994] is a distance-vector protocol tailored for MANET. Every node maintains a shortest path routing table with one entry for each destination node. The sequence number is used to avoid routing loops. A node increments its sequence number when a neighbor moves out or in its radius. Nodes select the route with the greatest number to reflect latest update. CGSR [CWLG1997] extends DSDV with clustering to increase the protocol scalability. WRP [MG1996] uses four tables to maintain distance, link cost, routes and message retransmission information. The shortest path to each destination on both the distance metric and the second-to-last hop metric guarantees it is loop-free.

2.1.3.2 Reactive Protocols

DSR [JM1996] is a loop-free, source based, on demand protocol for packets forwarding. Each node maintains has a route cache to buffer source routes learned by this node. DSR is triggered if the route cache does not have a valid route to the destination. Entries in the route cache are only updated in the new route discovery procedure. AODV [PR1999] is adapted from the DSDV protocol. DSR has a larger control overhead and needs more memory than AODV. A DSR packet must carry full routing path information, while an AODV packet only contains the destination address. On the other hand, DSR can utilize both asymmetric and symmetric links during routing, while AODV only works with symmetric links. This constraint is difficult to satisfy in mobile wireless environments. In DSR nodes maintain multiple routes to a destination to guard against link failure. AODV and DSR work well in small to medium size networks with moderate mobility. We will use DSR in our protocols and simulators because of its simplicity and efficiency.

TORA [PC1997] is built on the concept of link reversal of the Directed Acyclic Graph (ACG). TORA is highly adaptive and very quick in link failure recovery. It is also highly reliable since it provides multiple routes for any pair of source and destination nodes. It is especially suitable for large, dense, highly dynamic MANETs. However, its application suffers from its reliance on synchronized clocks. It simply does not work without an accurate external time source like GPS system.
2.1.4 Bootstrapping in MANETs

Although often being ignored intentionally or subconsciously, bootstrapping is an essential step in MANETs design and implementation. No real world MANET designer would think it skippable. The bootstrapping must contain an address configuration, may include topology construction, but latter is not essential.

2.2 Peer-to-Peer overlay networks

2.2.1 P2P Paradigm and Architecture

P2P overlay networks provide fast, accurate, and scalable resource discovery, resource sharing, and storage services without a central controller. The concept of peer-to-peer networks first appeared in the mid 1990s. As file sharing platforms, especially to distribute MP3 music files over Internet, peer-to-peer networks turned into a hot topic in the late 1990s. A traditional P2P network is built upon IP. It uses IP as the communication platform. An IP capable host can reach anyone and anything attached to the Internet or other IP networks like IEEE 802 family by an IP address. However, IP layer could tell a host how and where to find a given content or another participant host. This is done by P2P overlay networks (systems). The basic task of P2P overlay networks is to connect to other peers and find out interesting content.

These P2P networks are completely distributed and self-organized networks. By P2P jargons, a host is called peer, because all hosts usually have same status, share same responsibility, and the relationship among them is characterized by equality. Unused bandwidth, storage, CPU cycles at the edge of the network are utilized. Peers enjoy great freedom and privacy. Consumers are also producers, so aggregate resources grow exponentially with utilization. There is no single point of failure in a P2P network.

The emergence of P2P overlay networks was a counteraction against long time tradition of client/server model in computing and communication society. In the client/server model, powerful, reliable servers provide data and services; clients request data and services from servers. The client/server model has proved to be so successful by its famous offspring like World Wide Web, database systems, and FTP. However, it has following inherent defects:

- need central controller
- presents a single point of failure
- unused resources at the system edge
- poor scalability
dictation in which terminals feel like slaves

2.2.2 P2P Systems

P2P systems address above defects of client/server model very well. At large P2P computing aims at sharing and exchanging resources and services between terminals or systems. These resources and services include information (file or data structure), CPU cycles, storage (memory, cache, and disk), I/O devices, etc. P2P paradigm takes advantage of existing computing capacity, storage, and networking connectivity, so end users can unite and leverage their collective power to carry out huge task or achieve mutual benefits.

In a P2P network, all nodes are clients, servers, and routers at same time, provide and consume data and services. No centralized data source endangers the system as the single point of failure. Nodes collaborate directly with each other. Any node can initiate a connection. All nodes are totally free; they may enter and leave the network arbitrarily and frequently. It will be “the ultimate form of democracy on the Internet” as well as “the ultimate threat to copy-right protection on the Internet.” [Kaashoek2003]

P2P networks have following advantages: [Muthusamy2003]

- Efficient use of resources
  - Unused bandwidth, storage, CPU cycles at the edge of the network become available to any user

- Scalability
  - Consumers of resources also donate resources. If remarkable consumers turn into producer, aggregate resources will grow with utilization.
  - Self-scaling

- Reliability
  - No single point of failure
  - Geographic distribution
  - Replicas
  - Built-in fault tolerance
  - Fault tolerance

- Easy administration
  - Nodes self organizing
  - No need to deploy servers
  - Load balancing

Besides file sharing, P2P paradigm could be applied in collaborative Internet (e.g. ICQ, shared whiteboard), distributed computing and grid computing (e.g. UC Berkley Seti@home Project), multiplayer network games (e.g. Doom) and many other fields. However, P2P networks, especially those for file sharing, remain to be
the oldest and most sophisticated P2P application. In a typical file sharing network, a user makes files (music, video, etc.) on her computer available to others. Then another user connects to the network, searches for the files, finds the first user’s computer, and downloads files directly from first user’s computer.

P2P networks are fall into two categories: unstructured P2P networks and structured P2P networks. [Muthusamy2003] An unstructured P2P network does not have a fixed topology for routing. By the existence of central index servers, Unstructured P2P networks are divided into three subgroups: centralized with a central index server, like Napster; semi-centralized with local index servers, like KaZaA; decentralized without any index server, like Gnutella. [Clip2], [Ivkovic2001]

Structured P2P networks use fixed topologies like ring or grid via Distributed Hash Table (DHT) for routing. They impose specific local relationships between the peers, which finally generate global structures. These topology structures can be used for efficient data placement, search, and retrieval. They have guaranteed scalability — the hops in routing is not linear with number of nodes. Most of them could reach the logarithm. They are self-organized, fault-tolerant, and they support load balancing. Structured P2P networks are usually implemented via Distributed Hash Tables (DHTs). Typical systems include Chord [DBKKMSB2001], [SMKKB2001], Pastry [RD2001], CAN [RFHKS2001], and BitTorrent [Cohen2003I], [Cohen2003B].

### 2.2.3 Unstructured P2P Networks

Napster was devoted to sharing music files on Internet. Providers upload their list of files and IP addresses to Napster server. Downloaders send queries to Napster server for files of their interest in the format of keyword search. Keywords could be artist, song, album, even bit rate. Napster server replies with IP address of users with
matching files. Downloaders connect directly to the provider’s computer to download file. Using a central directory/index server and a central query database, Napster guarantees correct results. At the same time, the central server forms a single point of failure and bottleneck for scalability. Napster is susceptible to denial of service attack and mischief from malicious users.

Gnutella enables sharing any type of files, not just MP3. It employs decentralized search. In Gnutella, a user A asks her neighbors for files of interest, those neighbors ask their neighbors, and so on. Finally, either users with matching files reply to A’s query, or the packet is destroyed after a preset Time To Live (TTL). Each message has a parameter which sets the max number of hops the packet can “live”. Here, search is distributed by the means of queries flooding. Comparing to Napster, Gnutella is decentralized, robust to denial of service attacks, has no single point of failure. Nevertheless, it can not guarantee correct results for every query. Furthermore, Gnutella is still not scalable. [Clip2, Ivkovic2001]

KaZaA is a hybrid of centralized and decentralized structures, where super-peers act as local central nodes and local search hubs. Each super-peer is similar to a Napster server in a smaller scale. Super-peers are automatically chosen by the system based on their capacities (storage, bandwidth, etc.) and availability (connection time). Users upload their list of files to a super-peer, which periodically exchange file lists with neighbor super-peers. When the query reaches a super-peer for files of interest, the file is transferred back to the requesting node following the reverse of the original path. [Muthusamy2003]

**2.2.4 Structured P2P Network**

Structured P2P Networks are also called second generation P2P overlay networks. [Muthusamy2003] It is self-organizing and fault-tolerant, balance Load. Scalability
is guaranteed on numbers of hops to answer a query. The major difference with unstructured P2P systems is based on a Distributed Hash Tables (DHT) interface.

A DHT stores (key, value) pairs. Each peer stores a subset of (key, value) pairs. Core functions of DHT API include insert, lookup, and delete. Insert function stores a (key, value) pair at the node responsible for the key. Lookup function returns value associated with a key from a peer. Basic operation is to find node responsible for a key. First we map key to node, then insert, lookup, or delete the key to this node. DHT maps Keys evenly to all nodes in the network. Each node maintains information about only a few other nodes. Messages can be routed to a node efficiently. Arrival or departure of one node only affects a few nodes.

Many services can be built on top of a DHT interface, like file sharing, archival storage, databases, naming, service discovery, chat, rendezvous-based communication, publish and subscribe. There are several implementations of DHT generic interface, for instance, Chord from MIT, Pastry from Microsoft Research in UK and Rice University, Tapestry from UC Berkeley, Content Addressable Network (CAN) also from UC Berkeley, SkipNet from Microsoft Research and Univ. of Washington, Kademia from New York University, Viceroy from Israel government and UC Berkeley, P-Grid from EPFL in Switzerland, Freenet developed by Ian Clarke. These systems are also called P2P routing substrates.

Our first contribution is non-IP flat addressing scheme and its application in Pastry, i.e. our FAPSR protocol. This part of this dissertation is based on Pastry, primarily in Chapter 4 and 5. Our second contribution is ring topology automatic construction used to bootstrap Chord from scratch. The corresponding protocol is called RAN (Ring Ad-hoc Network). This part is based upon Chord, primarily Chapter 6 and 7 in this dissertation. Naturally, we will emphasize Chord and Pastry in Structured P2P networks (systems), not only in this section, also in related contexts of following chapters and sections.

Routing in Chord is implemented on Chord ring, on which nodes are organized according to their node IDs. Keys are assigned to their successor node in the circle. The consistent hash function ensures even distribution of nodes and keys on the circle. In a system with \( N \) nodes and \( K \) keys, with high probability, each node receives at most \( K/N \) keys, each node tracks about \( O(\log N) \) other nodes, lookups are resolved with \( O(\log N) \) hops. However, the efficiency comes with a loss in accuracy. In Chord, there is no guaranteed delivery, no consistency among replicas. Hops have poor network locality, nodes close on ring can be amazingly far in the physical network.

Pastry has a similar interface to Chord, however, it has good network locality to minimize hop traveling distance. To achieve locality new node needs to know a
nearby node. Each routing hop matches the target identifier by one more digit. There are many choices in each hop, called possible locality.

Figure 2.6 CAN network with 5 nodes in 2-d space

CAN uses a $d$-dimensional Cartesian coordinate space on a $d$-torus. Each node occupies a distinct zone in the space. Each key hashes to a point in the space.

BitTorrent has a highly connected ring topology with a center, like a bike wheel. BitTorrent uses economic methods in file sharing. It is faster and more reliable than most P2P approaches. BitTorrent forces concurrent downloaders for a same file to share the cost of upload. By using BitTorrent, they have to upload pieces of the file to each other. [Cohen2003I, Cohen2003B]

Figure 2.7 BitTorrent nodes upload pieces of common file to each other
2.3 P2P Overlay Networks above MANETs

2.3.1 Motivation and Reason

Till now, the synergy of MANETs and P2P Networks is almost unidirectional. In most case, only achievements from P2P systems are adapted into MANET environments, not the reverse. This is due to the tremendous difference between their levels of real world application. P2P file sharing and other application software has accomplished brilliant commercial achievements, although many of them also were accompanied by notorious legal issues. On the other hand, MANETs is like a highly regarded prodigy with bad luck. In spite of well circulated prospect and prediction, no much commercial success in industries.

![Figure 2.10 Relationship of P2P and ad-hoc from the first viewpoint](image)

Generally there are two points of view to understand the transplantation from P2P networks to MANETs. One regards P2P approaches purely as a scheme at application layer, while regarding MANETs as an infrastructure at lower layers, see Figure 1 [TR2002]. In this semantics, P2P is used as a platform for various applications, especially its traditional roles in file sharing, resource and content discovery.

In another viewpoint, P2P systems and MANETs are treated equally as two networking technologies; each has its own full length protocol stack. This approach aims at improving performance of MANETs and solving problems in MANETs, especially in file search, service/content discovery, and routing. Basic metrics are
average routing time, scalability, overheads in critical resources like energy, CPU cycles, and storage.

2.3.2 Typical Application of P2P Systems/Networks over MANETs

- Content/resource lookup
- Content/resource discovery
- Data dissemination/replica
- Distributed storage system
- Wireless beeping
- Small scale instant messaging
- Indexing
- Emergency data sharing
- Subscriber/distributor system

2.3.3 Guideline for Implementation

Corresponding to two viewpoint models given above, there are two rules to implement P2P systems over MANETs: the layered approach and the integrated approach. The layered approach puts P2P system on upper layers in ISO OSI model, usually the highest application layer; at the same time, it uses MANETs as lower layers, usually below the transport or network layer. The relation of P2P layer and MANETs layers is vertical and their interface is very clear. This approach will be easy to understand and would facilitate design and implementation. However, the efficiency of systems implemented in the layer approach is not high. On the contrary, the integrated approach has higher efficiency, but blurry and difficult to design and understand. It mixes up mechanism of two frameworks and emphasizes the efficiency and other performance metrics.
CHAPTER 3 RELATED WORKS

3.1 Related Works in P2P Systems over MANETs

Generally, there are two schools of research in integration of P2P and MANETs. First school builds their work upon existing P2P systems and MANET protocols. They have more legacies from MANETs than from IP. Their systems are likely small and efficient. The second school does not copy or adapt existing protocols or approaches; instead they ignore layer models in P2P and MANETs and try to set up something different. However, all research is historical; many of them actually have more heritages in IP than in MANETs. Many schemes proposed by this group have index or indexing in their name. Their systems are usually complicated, heavy-weight; and many implementations only have good performance with full support from platforms and operating systems. Many literatures in this group mixed up basic concepts, like comparing P2P document search with MANET routing. [LW2002]

In first school, Dynamic P2P Source Routing (DPSR) [HPD2003] and Ekta [PDH2004] by Y. C. Hu et al are among few that feasibly deviate from IP framework. They integrate Pastry [RD2001] and DSR [JM1996] at network level. Almost everything from Pastry is inherited; but in the low level implementation, they substitutes IP address in Pastry routing tables and leaf sets with DSR source routes. However, IP address is still their foundation for almost everything, from self organization, DSR implementation, to setting up of Pastry overlay. And they assume original IP addresses are a prior configured. It is a rather impractical assumption.

In the second school, 7DS [PS2001] used cooperative caching concept to implement P2P file sharing in MANETs. However, using their techniques, the success rate of a search is not predictable and highly depends on the search locality in the system. Passive Distributed Indexing in ORION designed by C. Lindermann et al [KLW2003], [LW2002] is not closely related to MANETs except mobility model. S.Y. Lee et al proposed Backtracking Chord and Redundant Chord [LJLQC2004], however, they did not describe how Chord was built on MANETs and what network layer protocol was used.

Konark service discovery and delivery protocol [DVH2003], [HDVL2003], designed by N. Desai et al, simply assumes an IP level connectivity among devices in the ad-hoc networks. It also assumes network support for routing multicast traffic when a node leaves or several networks merge. These assumptions are extremely unrealistic, almost equivalent to the hypothesis that without any effort a perfect P2P over MANETs system already exists, from physical layer to transport layer. The Proem platform, proposed by Gerd Kortuem [Kortuem2001], has similar problem in
practicability. It bypasses IP, but needs IP addresses for all nodes, TCP/UDP, and SOAP for XML. On the other hand, Multi-level Peer Index developed by Mei Li et al [LLS2004] is very constructive in this group.

Besides these two schools, there are also some research on evaluation and theoretical discussion. L. B. Oliveira et al have done extensive evaluations on ns2 platform. However, they did not give detailed description of how P2P was built over MANETs. [OSL2003] They even did not mention which P2P system was used. These evaluations seem detached with relative research because of vagueness nature in their implementation. [OSMLWN2005] used Gnutella and Chord as representative of unstructured and structured P2P systems, but still did not give the interface of these P2P layer with underlying layers, for example, what is used in Chord finger table to replace IP address.

3.2 Related Works in Automatic Address Configuration

Almost all network operations targeting individual nodes need some kind of identifiers to distinguish one node from the others. Only exception is flooding operation, and semi-flooding operations like broadcast and multicast, no matter what kind of network it is. It makes no difference if the network is wired or wireless, with or without infrastructure, with or without central controller, uses this or that transmission media, this or that physical layer technology, etc.

For major part of wired TCP/IP networks with fixed connection, especially those key addresses of a network, the configuration of routable address (global IP address which usually involve routers) is done long before the nodes are used, that is, before bootstrapping. The addresses usually are set statically by manufacturer or end user who already researched the network, its task, its traffic pattern, and most important, has global information, so duplicate addresses are always blocked from very beginning. However, there are quite some dynamic automatic address configuration schemes in TCP/IP family, like ARP, DHCP etc. Among them only link-local address scheme is fully decentralized and does not need a server in the whole procedure. But this default setting has so strong influence that when dealing with dynamic networks like MANETs, many people still make similar assumption that unique IP addresses have already been configured by force of habit.

3.2.1 Automatic Configuration of Link-Local IP Addresses

For all IP based networks, including IP-based MANETs, when no external configuration information is available, an automatic configuration scheme has been proposed by RFC 3927 [CAG2005], which generates an IPv4 link-local address such
that a local network could always works no matter what condition exists in external world. The address has a prefix 169.254/16 and ranges from 169.254.1.0 to 169.254.254.255. This scheme enables using IP tools in local communication without global address configuration either by static approach or using DHCP.

The basic idea of RFC 3927 is: select a random address in the range, then test if it is already use by other nodes on the local link. If test confirms that the address is not used, the requesting node claims this address by broadcasting ownership announcements in the local link. Otherwise the requesting node chooses a new address for test. Messaging format conforms to ARP.

RFC 3927 defines the link-local node (address) as a node (address) on the same single link. A set of hosts is regarded to be on the same link, if the link-layer packet payload is not changed when a node from that set sends a packet to another node in that set.

3.2.2 Automatic Addresses Configuration in MANETs

If two or more nodes are assigned a same address, an address collision occurs. Address collision is the major problem in automatic address configuration. By possibility of address collision, automatic address configuration schemes in MANETs are classified into two categories: collision-free addressing and collision-prone addressing. The former assigns an address to each node, then check if there are address collisions. The latter has some mechanism in assigning algorithm to prevent address collision.

Automatic addressing schemes could also be classified as decentralized, leader-based (or use another term clustered) in terms of distributed algorithm used. Another way to organize automatic MANETs addressing schemes is simply split them into IP addressing and non-IP addressing.

3.2.2.1 Collision-free Addressing

The collision-free addressing guarantees that every node in the network is assigned a unique address or ID. The oldest and simplest method is to use collision-resistant random number generator to get a new address. This requires at least a cryptographic pseudo-random number generator or collision-resistant hash like RSA or SHA-1, better use an entropy harvester [VM2003]. Our protocols FAPSR, RAN are non-IP collision-free schemes using collision-resistant random number generator.
Other collision-free addressing schemes usually use hierarchical structured cluster to makes all nodes have disjoint address pools. For example, binary buddy system, used by Mohsin and Prakash [MP2002], and Dynamic Configuration and Distribution Protocol, proposed by Archan Misra et al [MDMD2001], allocate half address pool of an existing node to a new comer. One advantage of this disjoint allocation is that it works well with network partitions. When a network partitions into two separate networks, their sets of node address pools keep disjoint. Thus the addresses allocated are different as well. This way if the partitions merge later, no further work is needed to handle address pool. The binary buddy system uses network to detect network partitions and merges.

### 3.2.2.2 Collision-prone Addressing

The collision-prone addressing follows a “try and validate” strategy and uses computationally expensive **Duplicate Address Detection** (DAD) for new address allocation, sometimes for old address as well, for example, massive DAD when two MANETs merge. The newcomer is assigned a tentative address, and then request for validation messages are usually broadcasted through entire network to test the uniqueness of this address. If a collision is found, usually the newcomer has to try another address. This procedure is repeated until no collision is reported. Then the newcomer takes the tested collision-free address as its address.

From above description, it is obvious that **Duplicate Address Resolution** (DAR) is mostly dump and retry. However, if a positive DAD is found when tow identical addresses have been used for a long time and have already disseminated into many nodes’ routing table and other data structures, like in a merger, above simple DAR can not be adapted. Instead, much more sophisticated algorithm has to be deployed.

Most existing approaches in MANETs auto-configuration are collision-prone. Many collision-prone schemes have been proposed, however, no one is good enough to be standardized and be able to replace other schemes.

### 3.2.2.3 Typical Collision-prone Addressing Schemes

**ZeroConfigure**

Perkins et al [PMWBS2001] presented an early DAD solution in 2001. In their Internet Draft “IP Address Autoconfiguration for Ad Hoc Networks,” addresses are randomly selected from the address ARP range 169.254/16. Each node performs DAD by flooding an Address Request message in the network to find if duplicate usage of its tentative address exists. If a duplicate address is found, an Address Reply
message is sent back and a different address has to be generated. The absence of an Address Reply message after a timeout indicates the availability of the requested address. This approach does not consider network partitions and merges.

**MANETconf**

In 2001 Nesargi and Prakash proposed MANETconf scheme to solve partitions and merges problems in [PMWBS2001]. In this scheme, each node maintains a list of all IP addresses used in the network. A newcomer is simply assigned a free address not in the list unless two or more new nodes arrive at the same time. DAD in MANETconf is primarily used for handling this special case. A new node obtains an IP address through an existing node A. A performs an address query for the new address. Positive acknowledgment (ACK) messages mean no collision.

To detect partitions and mergers, each node is given a partition ID. A periodic Hello message is used to circulate the partition ID. If a node fails to obtain ACKs from all other nodes, it means that a partition has occurred. Those nodes from which an ACK are not received would be deleted from the address list. Any change of group member is accompanied by corresponding partition identifier reconfirmation. When partitions merge, nodes from different partitions exchange their current addresses set to find duplicates. A network ID is generated by the node with the lowest IP address and broadcasted throughout the new network periodically. However, MANETconf requires entire network starts from one single bootstrapping node, which results in numerous unconnected networks at beginning, and considerable workload for merger later.

**Weak DAD**

Vaidya proposed a weak DAD scheme [Vaidya2002] to facilitate network merger, which. The scheme allows duplicate addresses. The objective of the scheme is to prevent a packet from being routed to a wrong destination when duplicate addresses occur. Key data structure is a unique per-node key, which is included in routing table entries and in routing control packets. Every node is identified by a unique tuple of key and IP address. If two nodes are assigned a same IP address, they are still distinguished by their keys. MAC address is recommended as the candidate for keys. This scheme favors proactive routing protocols. A “lazy” detection is employed to detect duplicate addresses. Here lazy means the detection only occurs after data is sent and routing information is exchanged. The scheme can not detect partitions and merges.

**Address Authority**

Sun and Belding-Royer [SB2003] use an elected Address Authority (AA) to maintain the state information of the network, including node addresses, node
lifetime, and unique network identifier. A node obtains a candidate address through a network-wide Address Request service and registers the address with an AA if no collision is detected. The AA periodically broadcasts Network ID Advertisement messages to detect partitions and merges. When nodes do not receive AA advertisement for a certain timeout, they will elect a new AA. When an AA hears another AA’s advertisement, a network merger is initiated and duplicate addresses are dealt with by both AAs.

**Prophet**

The prophet addressing approach [ZNM2003] utilizes a stateful function $f(n)$ to generate a series of random numbers. $f(n)$ is selected such that the possibility of duplicates is kept low. The first node in the network sets its address and chooses a random number as the seed for its $f(n)$ to compute a sequence of addresses for the network. Another node obtains IP address and state value from the first node. This state value is used as the seed for its $f(n)$. The same process continues as nodes join the network.

### 3.2.2.4 Non-IP Addressing

Non-IP addressing schemes are best suitable for stand-alone MANETs, which do not connect to Internet or other wired networks with cumbersome infrastructure. Stand-alone MANETs support typical, or to be precise, signatured, MANETs applications like emergency response, battlefield C3I systems, impromptu laptop communication in conference. Basic argument in favor of non-IP addressing is in stand-alone MANETs IP address loses its routing functionality in other infrastructured networks like Internet, LAN, MAN, and WAN; hence it become a pure identifier, so could be replaced by any other node identifier, which might be more efficient and more effective.

Few schemes have been proposed using non-IP addressing. Boleng [Boleng2002] adopts variable length non-IP addresses, which aims at saving storage and reduce data transmission overhead by minimizing node address in message header. The rest part of the scheme is similar to MANETconf. For every joining node, a nearby agent floods an address request message to entire network. Any node has the knowledge that the address is already in use must reply with a negative response. The negative response may contain a suggested address. To simplify the new address selection, all nodes have a record of known highest address. The scheme could handle node join and leave effectively, but does not cover solution for partition. For network merger, the detection and resolution both are not carefully examined. Besides, the network initialization is identical as node joining, which may cause too many very small networks at beginning and a lot of merging overhead subsequently. Another problem
is the scheme has not been validated by either theoretical reasoning or simulation data.

Elson and Estrin suggested an Address Free Architecture [EE2000], which only applies to sensor networks.
CHAPTER 4 USING NON-IP ADDRESS IN MANETS

4.1 Introduction

Many researchers are exploring dynamic IP address assignment in mobile ad-hoc networks, while much less effort is committed to non-IP addressing. The overwhelming success of TCP/IP in last two decades may be a good explanation. The pressure to connect to the Internet and keep all IP-oriented applications available might further explain the difference from a more pragmatic viewpoint. Nevertheless, these practical concerns are not necessarily tenable. To reach the Internet or other external networks, few gateway nodes could establish good enough interface. To access IP-oriented applications, a virtual IP overlay layer could be built upon non-IP transport layer; and a ARP like protocol could implement the address translation.

Let's first define stand-alone MANETs as MANETs which do not connect to Internet or other wired networks with cumbersome infrastructure. Stand-alone MANETs support typical, or to be precise, signatured, MANETs applications like emergency response, battlefield C3I systems, impromptu laptop communication in conference.

Beyond above mentioned concerns, it is not essential to use IP address to configure a stand-alone MANET, especially for adoption of peer-to-peer systems. Furthermore, among previous research projects on MANETs, except those targeting dynamic address assignments, most assume that IP addresses are a priori configured, preferably by hardware manufacturers or an administrator, who has global information and global control of all participating devices. This assumption is impractical and conflicts with many typical MANETs application scenarios, such as conference, emergency response, and tactic communication systems.

The cold reality is: IP address has lost its advantage in MANETs, for its hierarchical structure stops to play a role in MANETs routing, which in turn is probably due to the absence of buses and rings in MANETs hardware. Compare to wired networks like Internet and LAN, this hierarchy is foundation for entire TCP/IP suite, from classful address to subnet mask, from ARP, RIP to multicast. This hierarchy has been inherited from IPv4 to IPv6. However, when coming to MANETs, IP address becomes pure identifier, no more powerful implicit role in routing. This makes IP address identical to any other identifier. There is no essential difference between IP address and non-IP flat address in MANETs scenario.
We argue that non-IP addressing schemes are better suitable for stand-alone MANETs. It is more efficient and more effective for MANETs bootstrapping, especially for bootstrapping P2P systems over MANETs. The fastest automatic address configuration is achieved using a collision resistant random number generator, including collision resistant hash function, to avoid or largely reduce the possibility of duplication. Flat non-IP address is more suitable, because it has no restriction on ID number and has more randomness with a given length. In addition, self-organizing algorithms for flat address are easier to develop than for IP addresses.

As described in [Henson2003], cryptographic hashes only work well when the randomness of input is sufficiently high. By definition, IP addresses are hierarchically structured, syntactically segmented, and semantically rich, no matter classless or classful. They are not stochastic, have less entropy than flat addresses with same length. With IP addresses as input, a cryptographic hash function has less entropy than flat address in domain, so the range will be less random, less uniform in distribution.

4.2 Advantages of Flat Address over IP Address

In MANETs, we are more interested in setting up a local, temporary communication system. Simple flat addressing is preferable than complicated, subtle, globally standardized IP addressing. If this local system needs connections to Internet or other external IP networks, a few gateway nodes are sufficient. In peer-to-peer overlay networks upon ad-hoc networks, an IP address is excrescent, since the address is purely used as identifier without any effect on routing. Any unique integer is sufficient and more efficient than IP addresses. Furthermore, flat addresses are more flexible, easier to use in design and implementation. Further more, the set of flat addresses is a superset of the set of node IP addresses.

4.2.1 IPv4 Address

A flat address with no structure is better than a hierarchical IP address. First, the length of flat address is not fixed; while an IP address has to be fixed at either 32 bits for IPv4 or 128 bits for IPv6. Second, with same length as IP address — suppose flat addresses use 32 bits as IPv4, a flat address has randomness at all 32 bits. On the other hand, an IPv4 address uses considerable bits in hierarchical information and does not have good randomness. IP addresses reflect the hierarchy of IP networks, which is subject to strict requirements of complicated TCP/IP protocols family.

In classful IP addresses, network ID portion of an IP address can not be changed in a small neighborhood, which is usually the case in MANETs. In a MANET, all
wireless nodes in the 3D physical neighborhood must belong to the same network. These nodes have to share same network ID to save the communication overhead. Class A IP addresses has only first octet fixed. Its last three octets are source of randomness, so it has 24 random bits. However, it is very hard, if not impossible, for a normal network user, especially a user with MANET applications, to apply for a Class A address. Only about 63 Class A address had been handed out till 1998, according to Matthew Naugle [Naugle1998]. Nowadays you can only get a subnet of Class A address, with a subnet mask one byte or two bytes long, hence the randomness is reduced to 8 ~ 16 bits. Class B is most popular, but its popularity has made it a rare resource. Its maximum randomness is just 16 bits. For a Class C address, the randomness is just 1/254. 254 is from 256 – 2, since all 1 is for broadcasting, all 0 indicates a network number.

In case of classless IP addresses, used by the Classless Inter-Domain Routing protocol, which is regarded as a stopgap measure in the interim from IPv4 to IPv6, all classless IP addresses have a prefix. These prefixes have same disadvantage as network ID in classful IP addresses. The prefix can not be included when we count the random bits.

### 4.2.2 IPv6 Address

We did not compare IPv6 addresses in the simulation, because they are unnecessarily luxurious and do not help much in terms of entropy. According to current standard IPv6 Provider-Based Unicast Address Format, i.e. RFC3587 [HDN2003], in a 128-bit IPv6 unicast address, only up to 64 bits have real randomness in a small network environment like MANETs. Other 64 bits has been used to provide global hierarchy: 3-bit IPv6 prefix, 45-bit global routing prefix (including registry ID and provider ID), and 16-bit subnet ID. Our simulation shows that over 32 bits, cryptographic random number is already sufficient to guarantee the uniqueness of node ID. So IPv6 addresses do not need go through computational costly cryptographic hash after cryptographic random number generation to be collision free. We believe in a pure mobile ad hoc network context and for a temporary local application, it is reasonable to use only one class of IP address instead of a mixture of multiple classes.

### 4.2.3 Local IP Address

In pure MANETs setting, link-local IP addresses are usually classful addresses. They are better in entropy-bit rate than global IP addresses. However, similar to global IP addresses, they are still inferior to non-IP flat addresses. Link-local IPv4 addresses, as suggested by RFC 3927 [CAG2005], have 169.254/16 prefix, which leaves only 16 random bits, so their collision estimate is same as Class B IPv4 addresses.
Link-local or site-local IPv6 addresses usually get their randomness from the last 48 bits, which are suggested to be IEEE MAC addresses. Same as global IPv6 addresses, local IPv6 addresses are not cost-effective and do not help much in gaining entropy.

4.3 Simulation

The objective of our simulation to show that flat address is superior to IP address in the autoconfiguration of Pastry over MANETs context. The basic metrics is address collision. We compare collision numbers in 128-bit Pastry node ID generated from our flat non-IP address approach and to those generated from IP-based approaches. As specified in Pastry, IP-based approaches use security hash of IP addresses as Pastry node ID. Our approach uses 128-bit cryptographic random number generator (CRNG) to generate node ID, which is used for all purposes, including Pastry, source routing and else. However, IP addresses here are not assumed as a prior configured. To be able to compare their collision resistance, IP addresses are subject to cryptographic random number generator as well. To be magnanimous, we just let IP address lose their randomness or entropy from the network ID with the consideration of MANETs, and do not apply other restrictions on randomness like subnet mask, multicast special address, and link-local address restriction 169.254/16 (which further limit Class B IP address space) etc.

The simulator is coded in Microsoft Visual C++ .NET 2003. It uses CryptGenRandom function in Microsoft Platform SDK 2001 Edition as random number generator, and uses SHA1CryptoServiceProvider class in .NET Framework Class Library to perform SHA-1 hash. CryptGenRandom uses numerous sources — such as mouse or keyboard timing, process ID, thread ID, system clock, system counters, memory status, etc. — to produce the seed. The result is SHA-1 hashed, and the output is used to seed an RC4 stream. The stream is then used as the random stream and to update the stored seed.

As we argued in Section 3.2, IPv6 addresses are not compared in the simulation, because they are less bit effective and do not help much in gaining entropy. To be fair, we add another stream from 32-bit node ID from a CRNG, then through a cryptographic hash. So it has same source randomness as IPv4. Totally we compare five streams of collusion number from:
1. 128-bit flat addresses (Pastry node IDs) generated by CRNG in FAPSR
2. 32 bit flat addresses generated by CRNG then cryptographic hashed into 128-bit Pastry node IDs
3. Class A IPv4 addresses by CRNG then cryptographic hashed into 128-bit Pastry node IDs in DPRS
4. Class B IPv4 addresses by CRNG then cryptographic hashed into 128-bit Pastry node IDs in DPRS
5. Class C IPv4 addresses by CRNG then cryptographic hashed into 128-bit Pastry node IDs in DPRS

We group our data according to the size of MANETs, that is node number in MANETs.
- Small MANET: 200 ~ 1,000 nodes
- Medium MANET: 2,000 ~ 10,000 nodes
- Large MANET: 30,000 ~ 150,000 nodes

For small, medium, and large MANETs, all collision numbers are average of 20, 10, and 5 runs of same simulators respectively. Because the difference between schemes is huge, three scales of same data are shown in ascending resolution.

Table 4.1 Collision in Small MANET

<table>
<thead>
<tr>
<th>Nodes in Network</th>
<th>200</th>
<th>400</th>
<th>600</th>
<th>800</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>128-bit Flat Address</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32-bit Flat Address</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IP Class A</td>
<td>0</td>
<td>0</td>
<td>0.2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IP Class B</td>
<td>0.2</td>
<td>0.4</td>
<td>2.2</td>
<td>5.2</td>
<td>6.2</td>
</tr>
<tr>
<td>IP Class C</td>
<td>81.8</td>
<td>318</td>
<td>708</td>
<td>1237.6</td>
<td>1949.8</td>
</tr>
</tbody>
</table>

Figure 4.1 Collisions in small MANET
Figure 4.2 Collisions in small MANET without IP Class C

Figure 4.3 Collision in small MANET without IP Classes B and C

Table 4.2 Collision in Medium MANET

<table>
<thead>
<tr>
<th>Nodes in Network</th>
<th>2000</th>
<th>4000</th>
<th>6000</th>
<th>8000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>128-bit Flat Address</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32-bit Flat Address</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IP Class A</td>
<td>0.2</td>
<td>0.2</td>
<td>0.8</td>
<td>1.2</td>
<td>2.6</td>
</tr>
<tr>
<td>IP Class B</td>
<td>32.6</td>
<td>127.4</td>
<td>278.6</td>
<td>488.4</td>
<td>741.4</td>
</tr>
<tr>
<td>IP Class C</td>
<td>7834</td>
<td>31307.4</td>
<td>70385.6</td>
<td>124917.8</td>
<td>195495.4</td>
</tr>
</tbody>
</table>
From the simulation results, it is easy to see that using DPSR like approach, 32-bit flat address is almost collision free (collision < 3 when network node number ≥ 100, 000). On the other hand, best of IP addressing schemes, the Class A scheme has more than 100 collision when network scale grows beyond 60,000. Actually, Class A IP address is very rarely used by normal network users. When it is used, it always along with significant subnet masks. So the true representative should be Class B IP scheme, which overflow the 100 line at just 4,000 nodes network scale. The real scheme used in FAPSR, 128-bit flat address scheme, never had any collision in our simulation, even at network scale as large as 150,000.

![Figure 4.4 Collisions in medium MANET](image)

![Figure 4.5 Collisions in medium MANET without IP Class C](image)
Figure 4.6 Collisions in medium MANET without IP Classes B and C

Table 4.3 Collision in Large MANET

<table>
<thead>
<tr>
<th>Addressing Scheme</th>
<th>30000</th>
<th>60000</th>
<th>90000</th>
<th>120000</th>
<th>150000</th>
</tr>
</thead>
<tbody>
<tr>
<td>128-bit Flat Address</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32-bit Flat Address</td>
<td>0.25</td>
<td>0.25</td>
<td>1.75</td>
<td>2.25</td>
<td>0.75</td>
</tr>
<tr>
<td>IP Class A</td>
<td>27.75</td>
<td>107.5</td>
<td>234</td>
<td>420.5</td>
<td>2687</td>
</tr>
<tr>
<td>IP Class B</td>
<td>6840.75</td>
<td>27499.75</td>
<td>61861</td>
<td>110019.25</td>
<td>171404.5</td>
</tr>
<tr>
<td>IP Class C</td>
<td>1758482.75</td>
<td>7032215.5</td>
<td>15818333</td>
<td>27858639</td>
<td>42546778</td>
</tr>
</tbody>
</table>

Figure 4.7 Collisions in Large MANET
Tables 4.1, 4.2, 4.3 and Figures 4.1 ~ 4.9 show detailed results from the simulation. The simulation has made it crystal clear that 128-bit flat address generated by CRNG is collision free in MANETs, especially for common MANETs with small scale of several hundred of nodes. Using this 128-bit flat addressing scheme, DAD would become unnecessary.
We believe flat address is a better choice than IP address, not only because it is collision free, but also because of its great simplicity and ease in network operation. More comparison study between flat address approach and IP approach should be conducted to elaborate the advantage of flat address scheme.
CHAPTER 5 FLAT ADDRESS PEER-TO-PEER SOURCE ROUTING

5.1. Overview

Most P2P applications are designed for and restricted to wired Internet context, in which IP is used to provide underlying connectivity. Recently, efforts were made to transplant successful Internet based P2P paradigms to wireless mobile ad-hoc networks. However, most attempts are deeply rooted in IP.

This chapter describes a novel Flat Address P2P Source Routing protocol FAPSR, which builds Pastry peer-to-peer system (network) over stand-alone mobile ad-hoc networks. FAPSR is based upon the non-IP addressing scheme introduced in details in Chapter 4. It is one of few P2P over MANETs systems which cover entire procedure of a functional system and do not assume a prior address configured MANETs. It employs a full-fledged bootstrapping algorithm to initialize the network, with realistic dynamic non-IP node ID configuration. In our jargons, a node ID is a counterpart of node address in IP-based systems. It uses cryptographical random number generator for node ID generation. Actually in our simulator, an entropy harvester bundled in .NET framework is used.

The non-IP addressing scheme has following advantage

- fast configuration
- light-weight
- efficient routing
- scalable

The FAPSR protocol has completely liberated peer-to-peer systems in ad-hoc networks from the shackle of old stereotype of IP and IP addresses. It opens up a space for efficient and light-weighted protocols. In this paper, we adopt flat address format for both DSR at network and transport layers and Pastry at application layer. This way we have avoided all complicated IP related problems in systems automatic configuration, routing, and maintenance. Other MANETs routing protocols could be easily adapted to this paradigm as well. Many MANETs routing protocols are compatible with it.

Discussion in rest of the dissertation is set in pure mobile ad-hoc networks context, which means the connection to external Internet or other IP-based wired networking infrastructure is provided by a few gateway nodes and addressing, naming, and
routing do not need any global consideration involving outside wired infrastructure. It is equivalent to previous term stand-alone MANETs. In another word, pure MANETs are closed, isolated, and independent of the world of wired networks. This assumption is rational because most applications of MANETs are either local or temporary, that is, limited in space or time. Their objectives have nothing to do with the external networks.

FAPSR is directly related to DPSR designed by Y. C. Hu et al [HPD2003], which in turn directly related to DSR and Pastry. In following sections, we first discuss Pastry, DSR, and DPSR, and then present FAPSR in details.

5.2 Foundation — Pastry and DSR

5.2.1 Pastry

As one of most popular structured P2P systems, Pastry has shown very good locality. [RD2001] Chord and Pastry are probably most widely used P2P platform till now. They have become classic in P2P community. Many follow up studies and research projects are built upon Chord and Pastry. Many other projects try to extend or improve their functionalities. Pastry is designed as a substrate for building various P2P applications. Applications already developed include PAST, a global, persistent storage, and SCRIBE, a scalable publish/subscribe system.

Data Structure

In a Pastry, each node has a unique, randomly assigned node ID in 128-bit uniformly distributed ID space. In Pastry, a message is mapped to a 128-bit key and routed to a node with numerically closest node ID to the key. For a Pastry network with N nodes, any message can be routed to any node in less than $\lceil \log_2 N \rceil$ hops. $b$ is a configuration parameter, usually assigned to 4. At each node, there is a $\lceil \log_2 N \rceil \times 2^b$ routing table. Each entry of the routing table maps to a destination node’s node ID and IP address. Each of $2^b - 1$ entries at row $n$ points to a node whose node ID shares the first $n$ digits of the owner node’s ID, but its $(n+1)$th digit is different with owner node’s ID. Each node also has a $2^b$ size leaf set and a $2^b$ size neighborhood set, whose entries are also node ID and IP address of a node in the network (system). The leaf set contains nodes with numerically closest node IDs, in which half are larger than owner’s node ID and half are smaller. Neighborhood set stores nodes that are physically close to the current node.
Routing

In each step of routing, the current node first searches its leaf set, if the message key falls in the range of leaf set, the message is forwarded to a node in leaf set whose node ID is closest to the message key. If no such a node is found, the current node searches its routing table and forwards the message to a node whose node ID shares a prefix at least one digit longer with the message key than the current node ID. If no such node is found in the routing table, the message is forwarded to a node in the neighborhood set whose node ID shares a prefix with the key as long as the current node, but is numerically closer to the key than the current node ID.

![Routing Diagram](image)

Figure 5.1 Routing a message with key d46a1c from node 65a1fc in Pastry

An joining node with a newly assigned Pastry node ID X initializes its state, that is, routing table, leaf set, and neighborhood set, by contacting a nearby node A. It requests A to route a special message with key X to an existing node Z whose node ID is numerically closest to X. X copies the leaf set from Z, neighborhood set from A, and the i-th row of the routing table from the i-th node encountered along the route from A to Z. Finally, X announces its presence and transfer its state to the initial members of its leaf set, routing table, and neighborhood set.

5.2.2 DSR

DSR (Dynamic Source Routing) [JM1996] is an efficient reactive protocol. In route discovery of DSR, the source node floods a Route Request packet in a controlled manner. Each Route Request packet contains a route record, which records nodes passed in the route. A sequence number is used to prevent route duplication. When the Route Request packet reaches the destination or a node with a live route to the destination, a Route Reply packet is sent back. Each node maintains a cache of routes that have been learned or overheard to limit routing. The route maintenance
monitors the route and informs the sender of routing errors such as corrupted or lost packets. If a route fails, the detecting node sends a Route Error packet to the source. Notified nodes remove all routes that use the broken hop in its cache after receiving it.

5.3 DPSR

5.3.1 Protocol

In DPSR [21], IP addressing is employed. However, in its predecessor DSR, it is optional to choose IP address or flat address. To put Pastry over DSR in MANETs, DPSR adds an intermediate layer over the Networking layer, which maps Pastry node IDs into IP addresses. The data structures of DPSR routing table and leaf set are similar to those in Pastry. The only difference is: each entry in leaf set and routing table stores a route to the destination instead of the IP address of destination, which means IP addresses in DPSR have no routing functionality.

Everything above this intermediate layer is same as in Pastry. DPSR inherits same pros and cons from Pastry except that DPSR has removed neighborhood set from data structure. By assuming a prior configured IP addresses, DPSR circumvented configuration problem. In initialization, DPSR assigns unique Pastry node ID to a node by cryptographical hashing of its IP addresses.

Routing in DPSR is same as in Pastry. Messages are routed according to Pastry’s prefix based scheme with assurance of locality. There is a little difference in DPSR node joining and node failure procession comparing to that of Pastry due to their difference in data structure. Node failure is again handled similarly as in Pastry. DPSR does not simply delete all route containing a failed node. Before deletion, it initiates a route discovery for failed node. It only performs the deletion when the route discovery fails again.

5.3.2 Considerations about DPSR

Specifically in DPSR, node IP address, which is used to generate 128 bit node ID, is not necessary. In Pastry, IP address is used: (1) primarily for low level routing. This is the fundamental essential function to every node. (2) In some applications (Pastry node ID is application dependent), to generate node ID. The second usage is not an indispensable function. Pastry assumes that the IP address of a node is already configured, either statically by user or OEM, or dynamically by a DHCP server. However, this assumption can not be extended to MANETs, in which the
initialization of a node is always dynamic, stochastic, and distributed, in another word, MANETs are self organized. The second usage of IP address in MANETs, could not be justified. If IP address is used to generate node ID, then how to secure a unique IP address dynamically becomes a more complicated problem. It is easy to see that automatic configuration of collision free IP addresses are more difficult than automatic configuration of conflict free flat address. If IP address is not used for generation of node ID, it becomes totally unnecessary.

5.4 FAPSR Protocol

FAPSR can be described as the integration of a self-organizing automatic configuration algorithm, which dynamically assigns a unique flat address to a new coming node and resolve the address conflict if any and a dynamic Pastry style peer-to-peer source routing protocol based on flat addresses, which is similar to DPSR but more efficient because of its independence of IP addresses. The most significant difference between our P2P source routing protocol and DPSR is adoption of flat address instead of IP address. In DPSR, a node has two addresses — IP address and Pastry node ID. FAPSR uses only one node ID through out entire life cycle of the node. There is another difference between FAPSR and DPSR: DPSR discarded Neighbor Table in Pastry, which we regard as an important component of Pastry. So we keep Neighbor Table in FAPSR.

Table 5.1 Comparison of FAPSR to Other Approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPSR (2003)</td>
<td>• Upon generic MANETs</td>
<td>• Need IP addresses</td>
</tr>
<tr>
<td></td>
<td>• Reactive searching</td>
<td></td>
</tr>
<tr>
<td>Ekta (2004)</td>
<td>• Upon generic MANETs</td>
<td>• Need IP addresses</td>
</tr>
<tr>
<td></td>
<td>• Reactive searching</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• All routing at network layer</td>
<td></td>
</tr>
<tr>
<td>FAPSR (2005)</td>
<td>• No IP address needed</td>
<td>• Routes need more storage</td>
</tr>
<tr>
<td></td>
<td>• Upon generic MANETs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Reactive searching</td>
<td></td>
</tr>
</tbody>
</table>

5.4.1 Node ID Assignment

In FAPSR, the node ID assignment scheme enforces network wide uniqueness of node IDs. We can assign node IDs in a separate procedure after network initialization, but it is more convenient and coherent to integrate the assignment into the node joining procedure.
To follow Pastry convention, we keep using 128-bit Pastry node ID as general node ID, although we are not obliged to do so and have freedom to set another size. The simulation shows that cryptographic random number generator (entropy harvester is a better choice for high-end MANET nodes) is almost sufficient to generate collision free 128-bit node ID in a MANET as large as 150,000 nodes in size.

Flat address assignment takes place in the node joining algorithm. In the node joining algorithm, a node simply takes a random flat address from 128-bit cryptographic random number generator. To make it 100 percent sure, the weak duplicated address detection and correction algorithm presented by Vaidya [Vaidya2002] could be integrated; however, our simulation results do not suggest it very much.

5.4.2 Data Structure

In its data structure, FAPSR uses three tables inherited from Pastry — routing table, leaf set, and neighborhood set — and route cache from DSR protocol. The data structure is similar to DPSR, except for the neighborhood set. The major difference with Pastry, DSR, and DPSR, is the content of entries in these data structure, which is shown in the following table.

<table>
<thead>
<tr>
<th></th>
<th>Routing Table</th>
<th>Leaf Set</th>
<th>Neighborhood Set</th>
<th>Route Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pastry</td>
<td>Pastry node ID</td>
<td>Pastry node ID</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IP address</td>
<td>IP address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSR</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Source Route of node IDs (any kind)</td>
</tr>
<tr>
<td>DSR</td>
<td>Pastry node ID</td>
<td>Pastry node ID</td>
<td>N/A</td>
<td>Source Route of node ID</td>
</tr>
<tr>
<td></td>
<td>Source Route of IP address</td>
<td>IP address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DPSR</td>
<td>Pastry node ID</td>
<td>Pastry node ID</td>
<td>N/A</td>
<td>Source Route of IP address</td>
</tr>
<tr>
<td></td>
<td>Source Route of IP address</td>
<td>IP address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ekta</td>
<td>Pastry node ID</td>
<td>Pastry node ID</td>
<td>N/A</td>
<td>Source Route of IP address</td>
</tr>
<tr>
<td></td>
<td>Source Route of IP address</td>
<td>IP address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FAPSR</td>
<td>128-bit node ID</td>
<td>128-bit node ID</td>
<td>128-bit node ID</td>
<td>Source Route of node ID</td>
</tr>
<tr>
<td></td>
<td>Source Route of node ID</td>
<td>Source Route of node ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Source Route of node ID</td>
<td>Source Route of node ID</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5.4.3 Routing

Routing in FAPSR is very similar to that in DPSR and Pastry: The routing input is a application dependent message key, which is usually generated by hashing application related parameters. Suppose a message with key $D$ arrives at a node with node Id $A$, and $R$, $L$, $N$ are $A$’s routing table, leaf set, and neighborhood set respectively. First look for $D$ in leaf set $L$, if found, take the route. If failed in searching $L$, set $s$ to the length of prefix shared by $A$ and $D$, then read $R[s, D_s]$, where $D_s$ is the $s$th digit of $D$. If $R[s, D_s]$ is not null, take the route from it. If $R[s, D_s]$ is null, search the union set $R \cup L \cup M$, find any entry $T$ which shares a prefix with $D$ longer than or as long as $s$, then take the route from the entry. In their original Pastry paper, Rowstron at el has proved that $T$ must exist. Then we just follow the route to reach the destination, which has the resource matching the key. The difference between FAPSR and DPSR lies in the node ID format. DPSR uses a series of node IP addresses to form a route, while FAPSR uses 128-bit node IDs in the route.

5.4.4 Node Joining and Departure

Node joining of FAPSR is similar to its counterpart in Pastry and DPSR. The difference is (1) node ID or address configuration; (2) the contents of state tables, which we already discussed in Section 4.2. Both Pastry and DPSR skip the real self organization — here it is synonym of automatic configuration. They assume node IP addresses are given and well configured; there is no collision or duplication of IP address in the MANET. Neither manufacturer nor user can assign collision free IP address in advance without consideration of application and environment. FAPSR does not use IP address, so we do not need worry about global consistency of IP address. Only concern of FAPSR is ID collision, which has been effectively handled with cryptographic random number generator.

When a new node $X$ joins, it first generates a 128-bit node ID, which has proved to be collision free by simulation. We suppose the node ID of $X$ is $X$ as well. Then it employs an approach like expanding ring broadcast if it is equipped with an adjustable radius antenna, in which it searches from near to far for a proximately closest neighbor $A$ which has already joined a live FAPSR network. Then it sends $A$ its node ID and asks $A$ to route a Pastry Join message with $X$ as its key. Routing of the Join message will result in a route from $A$ to $Z$. $Z$ will be the numerically closest to $A$. As in standard Pastry node joining procedure, $A$’s neighborhood set is assigned to $X$’s neighborhood set. $Z$’s leaf set is assigned to $X$’s leaf set. Suppose in this case routing table has $k$ rows. From row 0 to row $k-1$, each row of $X$’s routing table is assigned by corresponding row of corresponding node’s routing table. That is, the $i$th row of $X$’s routing table is assigned by the $i$th row of $i$th node in the $A$ to $Z$ route. If
the length of A to Z route (suppose it is \( l \)) is less than \( k \), then row \( l \) to \( k-1 \) would be empty. After construction of state table, \( X \) sends an Update message with its state tables to all nodes in its routing table, neighborhood set, and leaf set. All these nodes update their state tables correspondingly.

Node departure here includes node failure. Handle departure in FAPSR is very similar to the counterpart process in DPSR. Should any node become unavailable, the current node launches an on-demand source routing to it. If the source routing succeeds, only the entries of state tables were updated. All other things keep same. If the routing fails, same recovery procedure as in Pastry would be started.

Like Pastry and DPSR, FAPSR network initialization is realized via node joining. No topology generation is employed.

### 5.4.5 Network Merger and Partition

Like Pastry and DPSR, FAPSR has no elegant solution for network merger and partition, basically due to the Pastry infrastructure. It is very difficult to detect a merger or partition in Pastry framework. Pastry only mentions the possible partition after node failure and it suggests to use expanding ring multicast search to reintegrate partitions. There is even no easy way to tell if another node is in the same network (or partition) in Pastry and DPSR. To solve the merger problem, one choice is to add network ID and network size to the data structure of each node, which is also a collision free random number.

### 5.4.6 DAD and DAR

Since FAPSR uses 128 bit flat address generated by CRNG, which is illustrated in Chapter 4 with considerable details, DAD (duplicate address detection) and DAR (duplicate address resolution) are almost useless. A weak DAD should guarantee the initialization in FAPSR is collision free, especially for common MANETs with small scale of several hundreds to one thousand nodes.

### 5.5 Future Topics

The major difference of our FAPSR with existing approaches like DPSR is its integration of automatic address configuration to routing protocol. FAPSR fully utilizes the advantage of non-IP flat address over IP address in the pure MANETs.
This is a new area of research; many future topics could be developed. Here we mention a few as examples.

- Add a function to map node ID generated in FAPSR to IP address, such that IP-oriented application could remain available for nodes in MANETs running FAPSR.
- Design optimal gateway locating algorithms to determine minimum number of nodes in MANETs that serve as gateway to Internet while also keep internal related data flow at a reasonable low level.
- Limit the size of DSR route cache. Because using 128-bit node ID will increase route storage size 4 times than 32-bit node ID, and route cache can share with Pastry state tables (routing tables, leaf set, neighborhood set). There should be a considerable overlap between route cache and state tables. We can optimize the storage by avoiding redundancy of overlapped routes. It is also suggested by DPSR. This way, by trade off, we can keep balance of storage usage, such that it is not too higher than IP address scheme.
- Solve the deficiency in network merger and partition. Pastry, like most DHT algorithms, is designed from wired Internet environment, which is unlikely to have partition and mergence, so it has no functionality to handle partition and mergence. DPSR and FAPSR inherit this deficiency. This is also a fundamental problem in P2P (or DHT) over MANETs, which has significant importance. Heer at el [HGRW2006] gave a solution for mergence, which simply disassembles one network and let all nodes join into the other.
- Gateway configuration which connects the flat addressed pure MANETs to Internet.
- For normal size MANETs, especially those with a certain upper bound of node number, we can design a local surrogate ID mechanism used in local data structures (routing table, neighborhood table, and leaf set) to minimize the storage and message complexity caused by using 128-bit Pastry node ID in place of 32-bit IPv4 address. It is absolutely feasible because chance of duplicate addresses in local context is very small.
CHAPTER 6 RING CONSTRUCTION

6.1 Introduction

In this chapter, we first introduce background knowledge on Chord. Then basic concepts in three previous research projects — T-Man, T-Chord, and Ring Network — are discussed in details. They are most successful approaches for Chord ring construction in wired networks. Section 4 describes P2P systems in MANETs, their special characteristics, and current status on bootstrapping Chord over MANETs. The last section outlines RAN, our Chord ring constructor. Details about RAN and its simulation are given in Chapter 7.

6.1.1 Fundamentality of Overlay Topology

In computer networks, topology is frequently used to define relationships such as “which node is directly connected to which node,” or “which node is neighbor of which node.” For any structured P2P overlay network, overlay topology is crucial to its survival and success. That is why they are named structured P2P networks.

When bootstrapping a structured P2P network, there are two options. One is starting with a single node, no matter if this node is really isolated from other nodes, or it is actually in a crowdly neighborhood, the network always is jumpstarted as a single node networks. To build the required overlay topology, a very long time has to be spent before a decent topology is ready to use, since only way to expand the network is one by one node joining. Obviously this kind of bootstrapping is unacceptable, irresponsible, almost like an act of sabotage. No one would adopt this approach unless no alternative could be resorted to. Another approach is construct a significant scale overlay topology for each connected component after or while dynamic address assignment. With this option, a P2P system could advance into normal working status immediately after bootstrapping.

Topology lays foundation for routing, looking up, retrieval, data placement, data dissemination, and data aggregation in P2P networks, especially P2P over MANETs. Structured P2P systems impose specific local topological relationships between peers, which in turn form a global structure. Topology is one of dominant factors that effect efficiency, robustness, and feasibility of algorithms. Application of network topology is far more beyond the above list. For example, Jelasity and Babaoglu have shown
that problems such of network clustering and sorting can be transformed into topology problems and could be solved by specific topology construction. [JB2005]

6.1.2 Current Status

Recently, remarkable advance has been achieved in topology generation. All basic network topologies such as line, ring, mesh, star, tree already have state-of-art generator. Now researchers are able to design general purpose topology generator, which could construct any topology if a mathematical expression of preferred selection criteria could be inputted. [JB2005]

However, many problems also exist in this area. Many existing protocols for topology construction and maintenance are based upon unrealistic assumptions or requirements. Worst assumptions are those that assume a specific initial topology configuration of peers. Many structured P2P systems require the manual creation of a “seed” network in their bootstrapping protocols. Some protocols set the requirement that the network remains in an ideal topology all the time as necessary condition for normal operations. Another problem is the ignorance of network merger and partition. Some systems require each node keep and monitor global state of the network.

Another problem is many schemes for topology construction still employ centralized strategy. Some require central coordination; some follow a network-wide top-down view in algorithm/protocol design. Some have deficiency in fault-tolerance and recovery. For structured P2P systems over MANETs, some approaches can not keep up with the rate of change.

6.2 Further Discussion on Chord

6.2.1 Chord Consistent Hashing

Chord employs consistent hashing to assign ID to nodes and keys. The consistent hashing in turn uses SHA-1 cryptographical hash as its base hash function. The compositive effect of these two hash functions provides very fast distributed hash computation. The consistent hashing has three attractive idiosyncrasies.

First, like other DHT, consistent hashing helps routing in Chord remain scalable to network size, that is, node number in the network. Unlike many proactive routing algorithm, Chord does not need its nodes keep tracking of every other node. A Chord node just need track $O(\log N)$ other nodes in its finger table. This is also the
foundation of the Chord distributed hash table. Each node resolves the hash function by communicating with other nodes. Furthermore, a lookup search for a key in Chord DHT only requires $O(\log N)$ messages to be exchanged.

Second, it has superb load balancing feature and can map keys evenly to nodes with uniform random distribution. This character is very important to Chord’s success. It provides solid foundation for Chord’s scalability, that is, the scalability to base. Many calculations in Chord involve modular operation. The scalability to base makes Chord calculations independent of base. No matter how big a base you chose, this feature will keep Chord at similar performance level.

Third, consistent hashing is very stable. With help of consistent hashing, Chord could smoothly absorb disturbance from joining and ungraceful leaving (leaving without handling problems arising from the leave). In Chord ID space, joining or leaving of the $N$th node only affect $O(1/N)$ existing keys in network which need move to other nodes to maintain the network-wide load balance. This is almost theoretical minimum and optimum.

6.2.2 Routing in Chord

Routing in Chord is implemented by Chord Identifier Circle, as shown in Figure 6.1 and 6.2, on which nodes are organized according to node IDs. Keys are assigned to their successor node in the circle. The Hash function ensures even distribution of nodes and keys on the circle. In a $O(\log N)$ size Chord Finger Table associated with a $N$ size node set, $i$th finger points to the first node that succeeds $n$ by at least $2^{i-1}$.

Figure 6.1 Chord identifier circle
To look up a key $n$, first locate the furthest node that precedes the key in the finger table. Chord queries could find the target’s home address in $O\,(\log N)$ hops. In a system with $N$ nodes and $K$ keys, with high probability, each node receives at most $K/N$ keys, each node maintains information about $O\,(\log N)$ other nodes, and lookups are resolved with $O\,(\log N)$ hops. However, the efficiency comes with a loss in accuracy. In Chord, there is no guaranteed delivery, no consistency among replicas. Hops have poor network locality, nodes close on ring can be amazingly far in the physical network.

![Figure 6.2 Looking up a key in Chord](image)

### 6.2.3 Chord Algorithm and API

In Chord, a node ID is a unique $m$-bit identifier, hashed from IP address or other unique ID; a key is an $m$-bit identifier, hashed from a sequence of bytes; a value is sequence of bytes. Chord API includes following functions:

// ask node $n$ to find the successor of id

```c
n.find_successor(id)
```

```c
if (id ∈ (n, successor])
  return successor;
else
  p = closest_preceding_node(id);
  return p.find_successor(id);
```

// search the local table for the highest predecessor of id

```c
n.closest_preceding_node(id)
```
for i = m downto 1
    if (finger[i] ∈ (n, id))
        return finger[i];
return n;

// create a new Chord ring.
node.create()
    predecessor = nil;
    successor = n;

// join a Chord ring containing node n0.
node.join(p)
    predecessor = nil;
    successor = p.find_successor(n);

// called periodically. verifies n’s immediate
// successor, and tells the successor about n.
node.stabilize()
    x = successor.predecessor;
    if (x ∈ (n, successor))
        successor = x;
        successor.notify(n);

// n thinks p might be predecessor.
node.notify(p)
    if (predecessor is nil or p ∈ (predecessor, n))
        predecessor = n0;

// called periodically. refreshes finger table entries.
// next stores the index of the next finger to fix.
node.fix_fingers()
    next = next + 1;
    if (next > m)
        next = 1;
        finger[next] = find_successor(n + 2^{next-1});

// called periodically. checks whether predecessor has failed.
node.check_predecessor()
    if (predecessor has failed)
        predecessor = nil;
6.3 Previous Works in Wired Networks

6.3.1 T-Man — a Gossip-based Approach

Based upon popular gossip communication model [LMM2000] in distributed computing, T-Man [JB2005] is designed as a general purpose protocol for building and maintaining network topology. T-Man targets large scale and highly dynamic networks. It is simple, scalable, robust, and flexible, meeting the requirements of such networks. It may be used as a standalone program or as a bootstrapping component or a recovery component in other protocols. It is mainly used in P2P community, but has an application range far beyond this area. With the aid of its original concept and tool — the ranking function, T-Man controls the self-organization process of topologies in a straightforward, intuitive, and adaptive manner. T-Man follows a stepwise refining procedure with a short asymptotic time. T-Man is completely distributed. Each node relies solely upon local communication to increase the quality of the current set of neighbors. Its fast convergence and high robustness in dynamic environments have attracted considerable follow-up research.

T-Man is so adaptive and flexible that it allows for changing the topology on-the-fly at run time, without changing. All previous approaches have to revise protocol and program for each possible topology to achieve the same objective. As a general abstraction, topology can be used to directly solve problems or to enhance and support other solutions. Therefore on demand changing topology on-the-fly will have significant impact in both theory and practice. It may drastically increase the efficiency of distributed applications as well as the efficiency for deploying such applications. With the support for quick topology change, we can even derive best suiting topology for a certain scenario by progressive automatic evolution in topologies.

6.3.1.1 Gossip Protocol

Gossip Protocol [LMM2000], [JHB2001], [BEGH2004], [MMA2000] provides a scheme for performing reliable network broadcasts, probabilistically. In Gossip protocol nodes send a message to some of the neighbors (usually only one) instead of all. The recipients are usually selected randomly, but some variants use deterministic algorithms for selection. Due to the redundancy in the links, most nodes received the packet in limited steps (hops), so Gossip has similar effect like flooding. Gossip can be used to deliver multicast messages with less overhead and enhanced efficiency than normal flooding style broadcasting. Gossip minimizes amount of transportation, and hence reduce communication overhead.
6.3.1.2 Ranking Function

Key concept of T-Man is ranking function, which specifies the preference for a node to choose its neighbors in the target topology. A node uses the ranking function to order any set of other nodes according to the preference. This simple and elegant abstraction results in a highly effective algorithm which produces various topologies with astonishing preciseness and efficiency. For T-Man the ranking function is the source of effectiveness, versatility, and flexibility.

Suppose a network contains nodes, all connected to each other. Each node has an address sufficient for sending messages to it. Each node maintains addresses of other nodes through a partial view, which is a set of node descriptors. Besides a node address, a node descriptor contains a profile, which contains topology related properties, such as ID, geographical location, etc. Links of the topology are determined by addresses contained in partial views descriptors.

Following the selected ranking function, T-Man use local gossip messages and gradually evolves the current topology towards the target. According to inventors’ simulation report, the convergence is fast and scalable to network size. Convergence time grows as the logarithm of the network size. The high speed guarantees T-Man to be able to build divergent topologies on-the-fly. This feature makes T-Man a perfect fit for dynamic systems where the set of nodes and their properties change rapidly.

Here is a formal description. Suppose $N$ is the node set of a network. Each node $x$ maintains addresses of other nodes through a partial view, also called view for short, denoted as $\text{view}_x$. $c$ is the maximal size of partial views in the network. Ranking function $R$ has following parameters as its input.

- $x$, base node
- $S = \{y_1, y_2, \ldots, y_m\}$, a set of nodes

The output of $R$ is an $m$-tuple, which is a re-ordered $S$. The task is to construct views of all nodes such that the view of node $x$, $\text{view}_x$, contains exactly the first $c$ elements of $R(x; \{\text{all nodes except } x\})$, which is output of $R$ over the entire node set. That is,

$$R(x, \text{view}_x) = R(x, N - \{x\})$$

One convenient way to get a ranking function is through a distance function, which is derived from a metric space over the node set. The ranking function measures the Euclidean distance or any other distance from the base node. Following are few examples of defining distance function.
**Line**

Here the profile of a node is a real number. The distance function for a line is

\[ d(a; b) = |a - b| \]

Its variant can be extended to a ring. For example for a Chord ring with range \([0, N]\), node profile is an integer in \([0, N]\). Here distance is directional, that is, \(d(a; b)\) is not necessarily equal to \(d(b, a)\). The distance function is defined as

\[ d(a; b) = (a - b) \mod (N+1) \]

**Mesh**

Extending the 1-dimensional distance function for line to two dimensions we can derive distance function for a mesh. The profile for node is two-dimensional real vector. The distance for the mesh is the Manhattan distance, which is the sum of two 1-dimensional distances on two coordinates. Use the same transformation from line to ring, we can get profile and distance function for tube from those for mesh.

### 6.3.1.3 T-Man Protocol

Given an arbitrary overlay network, constructing a target topology is realized via connecting all nodes to the right neighbors. T-Man’s basic idea is there is a general relationship of nodes behind a given topology, which is expressed by a ranking function. The relationship between nodes could be geographical location, semantic description of stored data, storage capacity, etc.
T-Man is based on the gossip communication scheme. After initialization, each node executes the same T-Man protocol concurrently. But no synchronization or coordination is needed; nodes’ running is not synchronous. The protocol consists of two threads: an active thread initiating communication with other nodes; a passive thread waiting for and processing incoming messages.

**Initialization**

\[ \text{view} \leftarrow \text{rnd.view} \cup \{\text{myAddress, myProfile}\} \]

**Active Thread**

\[
\begin{align*}
\text{do} & \quad \text{at a random time once in each consecutive interval of T time units} \\
\text{p} & \leftarrow \text{selectPeer()} \\
\text{myDescriptor} & \leftarrow (\text{myAddress, myProfile}) \\
\text{buffer} & \leftarrow \text{merge(\text{view}, \{\text{myDescriptor}\})} \\
\text{buffer} & \leftarrow \text{merge(\text{buffer, rnd.view})} \\
\text{send buffer to p} \\
\text{receive buffer}_p \text{ from p} \\
\text{buffer} & \leftarrow \text{merge(\text{buffer}_p, \text{view})} \\
\text{view} & \leftarrow \text{selectView(buffer)}
\end{align*}
\]

**Passive Thread**

\[
\begin{align*}
\text{do} & \quad \text{forever} \\
\text{receive buffer}_q \text{ from q} \\
\text{myDescriptor} & \leftarrow (\text{myAddress, myProfile}) \\
\text{buffer} & \leftarrow \text{merge(\text{view}, \{\text{myDescriptor}\})} \\
\text{buffer} & \leftarrow \text{merge(\text{buffer, rnd.view})} \\
\text{send buffer to q} \\
\text{buffer} & \leftarrow \text{merge(\text{buffer}_q, \text{view})} \\
\text{view} & \leftarrow \text{selectView(buffer)}
\end{align*}
\]

As describe in Section 6.2.1.2, each node maintains a view. The view is a set of node descriptors. Function merge(\text{view}_1, \text{view}_2) returns the union of \text{view}_1 and \text{view}_2. In above protocol, two key functions are selectPeer() and selectView(buffer). Function selectPeer() uses the current view to return an address. First, it applies the ranking function to order the elements in the view. Then it returns the first descriptor that belongs to a live node. Function selectView(buffer) applies the ranking function to order the elements in the buffer. Then it returns first \(c\) elements of the buffer.
By using views of their current neighbors, all nodes improve their views, so that their new neighbors will be closer in the target topology. Neighbors will be closer and closer. All nodes run T-Man protocol simultaneously.

6.3.2 T-Chord — An Application of T-Man

6.3.2.1 Advantages of T-Chord

T-Chord efficiently bootstraps Chord from a random unstructured overlay using T-Man. It is one of most successful Chord ring building approaches in terms of thoroughness, speed, and efficiency. Simulation proved that T-Chord is able to create a perfect Chord topology in $O(\log(N))$ steps where $N$ is network size. It also shows optimized message latency. The generated network is immediate operable and could be handed over to the Chord protocol right way.

T-Chord completely breaks away from the old pattern of bootstrapping structured P2P system — that is, using a jumpstart node and node joining procedure in place of bootstrapping. The old joining based method is very inefficient; its fatal problem is: it is unable to parallel node bootstrapping. Nodes have to be booted one by one in a linear manner, which is very unrealistic for large scale network. [DBKKMSB2001] Some methods require booting nodes in a fixed order, which will not only need linear run time but also need complicated synchronization and coordination. One the other hand, without the constraint of single jumpstart node, in T-Chord every node starts its own topology building and optimization simultaneously and concurrently. Furthermore, unlike many other attempts to bootstrapping Chord, T-Chord does not need any a prior configured initial network or jumpstart node.

6.3.2.2 T-Chord Protocol

T-Chord starts from a connected unstructured overlay network with a random topology. In T-Chord simulation, the unstructured random network is generated by a lightweight membership protocol called NEWSCAT. [JGKS2004] However, the bootstrapping of T-Chord does not include node ID automatic generation. Nodes are a prior configured and unique IDs are assigned to nodes from a circular ID space. Next, T-Man is called to build the Chord ring. Only thing special for T-Chord is the ranking function. In fact the ranking function just needs minor revision to be adapted in T-Chord. In T-Man’s running procedure, not only direct successor and predecessor are located as outcome of ring topology, many encountered nodes are also remembered. These buffered nodes happen to be a very good source to build
Chord finger table. Therefore two essential Chord data structures are ready to use after calling T-Man.

6.3.2.3 Deficiencies of T-Chord

The most notable problem with T-Chord is its requirement for a prior configuration for Chord IDs. It ruins its good reputation and great prospective due to its ability to unconditionally bootstrap from arbitrary initial topology.

Another short coming is its distance function, which inherited from T-Man. Its definition

\[ d(u, v) = \min\{(v - u) \mod 2^m, (u - v) \mod 2^m\} \]

is not compatible with the distance definition in Chord, which is

\[ d(u, v) = (v - u) \mod 2^m \]

6.3.3 Ring Network

6.3.3.1 Features of Ring Network

The Ring Network (RN) protocol is an asynchronous message-passing distributed protocol, which fits well the autonomous behavior of peers in a P2P system. [SR2005] Peers do not need to be informed of any global network state. They are not required a grace leave, i.e. to assist in repairing the network topology caused by their leave.

Unlike T-Chord, RN protocol is not gossip based, though all of three perform the similar function. RN uses traditional distributed computing techniques like message passing. Another notable difference is initial condition. RN requires the presence of a weakly connected initial network called minimum bootstrapping system to be able to return a Chord ring, while T-Chord can start at any condition and find any connected component. Two nodes are weakly connected means there is a directed path between them no matter which direction the path is. For our RAN protocol and T-Chord, differentiating weakly connected components from strongly connected components does not make much sense, since we do not have any preliminary requirement about connectivity. In addition, since most devices in MANETs support duplex mode, there is no much pragmatic significance to find this difference. RN does not specify
the scale of the bootstrapping system and how the system is found or configured. From the Proposition 2.1 in the original paper about RN, we guess the bootstrapping system is a subset of all nodes to which every node is connected with at most one hop distance.

6.3.3.2 RN Protocol

The RN protocol is fully distributed. It can quickly adapt to churns in the network. All peers independently and asynchronously run a same set of procedures while they exchange asynchronous messages. Periodically each peer calls the Closer Peer Search procedure to search a closer predecessor in ID space, by which a closer successor candidate is also returned. As shown later in Section 6.3.3.5, authors confuse successor and predecessor in this part of RN algorithm. But, the pseudocode alone is still consistent and correct if we ignore the textual description.

Peers that participate in this search record information in any message they received. After collecting information returned by the predecessor search, returned by bootstrapping process, or gleaned from message propagation, each peer selects a currently closest successor. This process repeats till a complete consistent ring is formed. Local information stored by each peer includes:

- $\Gamma$: the set of current neighbors of the peer.
- $W$: the set of peers returned by Closer Peer Searches.
- $B$: the set of peers that the peer has learned by the Search Monitor while propagating search request messages on behalf of other peers.
- $s$: a peer selected randomly from the current successor, and peers returned by the bootstrapping system.

Three steps of the protocol are described below in more detail.

Closer Peer Search

Each peer $x$ periodically initiate a search for the successor candidate to which it is closer than to its current successor in the ID space. The approach is by first finding the closer predecessor to this current node. Current node $x$ randomly chooses a peer $s$, which is either its current successor $x.\Gamma_0$ or a peer returned by the bootstrapping system, and sends $s$ a Closer-Peer-Request message. $s$ forwards the message to one of its neighbors to which $x$ is closest. The receiver of this request propagates this request in a similar manner. This way $x$ gets closer and closer to the target. When a receiver $u$ finds that the initiator $x$ is closer to itself than any of its neighbors, the search is terminated. $u$ then sends to $x$ the address and ID of its successor $u.\Gamma_0$, which $x$ adds to its set $x.W$. 

58
The result of the Closer Peer Search depends on the current network topology. If the network is already in a ring topology, the search will not be really launched. Note that the search does not necessarily returns the closest node of \( x \) in ID space, because the ending node of the search may have a unvisited descendent node, which is more than one hop away, and \( x \) is closer to it. Furthermore, since the search is actually for a closer predecessor, it does not ensure of finding the successor to which \( x \) is closest. No matter \( x \) is closest to \( u.\Gamma_0 \) or not, since \( u \) is closest to \( x \), \( x \) will be always between \( u \) and \( u.\Gamma_0 \). So \( u.\Gamma_0 \) is a promising candidate for \( x \)'s successor. The frequency of this search only affects the speed of the protocol, not its correctness.

Figure 6.4 illustrates a Closer Peer Search. Left-hand side is the starting situation; right-hand side is the ending situation. Node 50 starts this search at node 30. The search terminates at node 40, which notifies node 50 its successor 60. Node 50 then sets 60 as its new successor. Actually the exact next step for node 50 is adding node 60 to its successor candidate set \( W \). To make it clearer, node 60 is assumed to be selected as new successor of node 50.

**Search Monitor**

Every peer \( u \) records each received Closer Peer Search request message. If a search is initiated by \( x \neq u \) and is terminated at \( u \), then \( x \) is closer to \( u \) than \( u.\Gamma_0 \). \( u \) then adds the address and ID of \( x \) to its set \( B \). In Figure 6.2, peer 40 adds 50 to its set \( B \).

**Neighbor Update**

Periodically every peer \( u \) checks if it has found a closer successor than its current successor \( u.\Gamma_0 \). It examines its current list of neighbors, a bootstrapping peer returned by the bootstrapping system, its set \( W \), and its set \( B \). The peer closest to \( u \) from among the union of these is chosen as the new \( u.\Gamma_0 \). In figure 6.4, after \( W \) and \( B \) have been updated, nodes 40 and 50 update their successors as well.
At the same time \( u \) updates all its other neighbors \( u.\Gamma_1, u.\Gamma_2, \) and so on. \( u \) sends a message to neighbor \( u.\Gamma_i \) asking it to return the ID of \( u.\Gamma_i's ith \) neighbor \( v \). If the ID of \( v \) is between \( u.\Gamma_i \) and \( u \), \( u \) sets it as new \( u.\Gamma_{i+1} \). Similar to finger table used in Chord, the purpose of such an update process is to minimize the number of hops and improve the search speed. \cite{HGS1987} In Figure 6.5 peer 30 updates its third neighbor. Since the order number starts at 0, the third is actually its No. 3 neighbor. It first asks its No. 1 neighbor, peer 50, for the No.1 neighbor. Peer 50 sends back 70. peer 30 then sets peer 70 as its No.2 neighbor. This is because peer 70 is between peer 50 and peer 30 if we look at them in a ring. Eventually peer 30 has discovered closer peer that is 4 hops away from it, using two messages.

### 6.3.3.3 AP Notation

AP notation is tailored pseudocode format for expressing network protocols. \cite{Gouda1998} AP notation is very instrumental for correctness analysis. This analysis model has been proved useful and widely adopted by the distributed computing community. It ignores the execution order of interleaving of actions of nodes in a protocol by assuming arbitrarily random order. It is especially suitable for asynchronous protocols, for it expresses asynchronous protocols clearer by eliminating the need for interrupts.

In AP notation a distributed protocol consists of a series of procedures associated with nodes in a network. A node is the carrier of protocols. Data structures of a node \( p \) are classified into three categories: constants, inputs, and variables, denoted by keywords \texttt{const}, \texttt{input}, and \texttt{var} respectively. The operation procedures of a protocol is put in actions section denoted by \( (a< i >) \), where \( i \) is the order number of procedures. Actions are delimited by two square brackets. An action is expressed in syntax

\[
<\text{guard}> \rightarrow <\text{statement}>
\]
The statement of an action can be executed only if the corresponding guard condition is evaluated to true. At the beginning of every round of running of a protocol, all guards of all actions of all peers are evaluated. Then only one statement of an action whose guard evaluated to true is executed. When there are more than one statements whose guards are evaluated to true, a true guarded statement is selected for execution at random. Every enabled action will eventually be executed, but the order and frequency of execution are arbitrary. RN and our RAN protocols are written in AP notation.

6.3.3.4 RN Algorithm

Here the algorithm pseudocode for the RN protocol is written in the AP notation. [SR2005]

Peer $u$

const
$T$ : set of bootstrapping peers

input
$w$ : a peer (successor candidate)
$x$ : peer being searched for
$c$ : index of received neighbor
$z$ : new neighbor
$s$ : a bootstrapping peer

var
$S$ : Set of peers
$B$ : Set of successor candidates
$W$ : Set of successor candidates
$\Gamma_i$ : $i$th neighbor

(a1) true $\rightarrow$

\[
S := \{s\} \cup W \cup B \cup \Gamma
\]
$\Gamma_0 := \arg\min_{k \in S} d(u, k)$
$B := W := \emptyset$

[]

(a2) true $\rightarrow$

$s := \text{Get random peer from } \{T \cup \Gamma_0\}$
\textbf{send} closerPeerSearch($u$) \textbf{to} $s$
\[\]

(a3) **receive** `closerPeerSearch(x)` **from** `q →`

\[ \text{if } x \text{ is closer to } u \text{ than any neighbor } \in \Gamma \text{ then} \]

\[ B := B \cup \{x\} \]

\[ \text{send } \text{successorCandidate}(\Gamma_0) \text{ to } x \]

\[ \text{else} \]

\[ \text{send } \text{closerPeerSearch}(x) \text{ to } \arg\min_{k \in \Gamma} d(k, x) \]

\[\]

(a4) **receive** `successorCandidate(w)` **from** `q →`

\[ W := W \cup \{w\} \]

\[\]

(a5) **true →**

\[ \text{for each } h \in \Gamma \text{ do} \]

\[ \text{send } \text{getNeighbor(index(h))} \text{ to } h \]

\[\]

(a6) **receive** `getNeighbor(j)` **from** `q →`

\[ \text{if } \Gamma_j \text{ exists} \]

\[ \text{then send } \text{neighbor}(\Gamma_j, j) \text{ to } q \]

\[\]

(a7) **receive** `neighbor(z, c)` **from** `q →`

\[ \text{if } \Gamma_c \leq z < u \]

\[ \text{then } \Gamma_{c+1} := z \]

\[ \text{else } \Gamma_{c+1} := \text{NIL} \]

\[\]

Note that function `\(\arg\min_{k \in S} d(u, k)\)` returns a `k` instead of `d(u, k)` or `(u, k)`.

### 6.3.3.5 Problems with RN

The most serious problem with RN is the minimum bootstrapping system required as a necessary condition of to apply RN protocol. RN does not specify the scale of the minimum bootstrapping system and how the system is configured or generated. Furthermore, RN does not specify: (1) whether and how the minimum bootstrapping system is generated? manually or automatic by a program? from an arbitrary network topology or an a prior configured topology? (2) how many hops away from the
minimum bootstrapping system is any node outside the minimum bootstrapping system? (3) how the RN is interfaced with the minimum bootstrapping system?

Second, RN is not guaranteed to converge to the ideal Chord ring within unbounded finite time. Actually our simulation shows all rings it constructed in MANETs are incomplete. When a connected network has more nodes it is getting more difficult for RN to converge to the ideal ring. Situation in wired network is similar.

Third, as the direct reason for above problem, the basic strategy of RN in searching closer node to the target node — continuously choosing closer neighbor at each step — has no logical support at all. The common sense reasoning is against this strategy. The distribution of node IDs is totally random. The approximity of one node has nothing to do with the approximity of its children nodes. No proof of correctness of RN protocol is presented in [SR2005]. We believe such proof is impossible.

Fourth, in their paper [SR2005], the authors confused some basic concepts and logic. For instance, they mixed up distance from node u to node v, i.e. \( d(u, v) \) with distance from v to u. A subsequent mix-up is the concept u is closer to v when \( d(u, v) \) is smaller. Because the distance is directional and modulus based, suppose here the modulus is m, the following equation always holds

\[
d(u, v) + d(v, u) = m
\]

Obviously, by definition, when \( d(u, v) \) gets smaller, distance from u to v becomes smaller, so u is closer to v. At the same time, v is getting farther to u. However, in [SR2005], the “smaller the value of \( d(u, v) \) the closer v is said to be to u.” It is not just a trivial issue as chopping logic. This mistakes leads to a more serious misuse of concept in following part of the paper. For example, the loser peer search is actually a search for closer predecessor of the current node by interpreting the pseudocode of their algorithm; however, they describe it as a search for closer successor in Section 4.1, which cause a lot more confusion and logical mess-up in RN protocol and algorithm.

Next, the procedure and result of simulation of RN is not very convincing. (Please refer to [SR2005] Section 5) The simulator used for RN simulation is NetLogo. [Wilensky1999] It is not a full-fledged platform which has gone through thorough validation. Not many models and functions for network simulation are included in libraries of NetLogo. For networking simulation the choices and possibilities are limited. In its latest version, i.e. Version 3.1, no model in the integrated library is ready for use for simulation in scenarios like RN. Most important, authors of [SR2005] did not mention any detail about how the simulation is coded, implemented, and deployed. No information for following questions is provided in [SR2005]: (1) whether and how the program is designed? (2) how the RN is terminated in the simulation? what is the ending condition of entire RN protocol? RN
has already given the ending condition of the closer successor search, but nothing has been said about terminating the whole protocol.

Last, in simulation of RN described in [SR2005], no convergence time data or any other data about performance of RN is provided. The simulation is about the quality of Chord ring generated. Authors of [SR2005] used a concept “perfect Chord ring”, however, the perfect ring does not perform best in their simulation. By definition given in Chord position paper [SMKKB2001], it is clear that there could be only one perfect Chord ring, in which all nodes in the networks are linear sorted. No other ring should be target of Chord topology construction, unless Chord is revise to a better version.

6.4 Previous Works on Structured P2P Systems over MANETs

6.4.1 Special Issues on P2P Systems over MANETs

6.4.1.1 Neighbor Relation

In wired networks like Internet, neighbor is defined on overlay layer and low layers such as network layer. We can say being neighbor is equivalent to knowing address. Two nodes \( u \) and \( v \), we say \( v \) is \( u \)’s neighbor only if \( u \) knows \( v \)’s network address and be able to send a message to \( v \). By this definition, neighbor relation is unidirectional and not commutable. When \( u \) knows \( v \)’s address, we have no clue if \( v \) knows \( u \)’s address.

On the contrary, in MANETs, neighbor is only defined on lowest layer, e.g. physics layer or MAC layer. Defining layer could be expanded to network layer. In most cases, it is define by radio range. From this point of view, it has nothing to do with Chord ID space or overlay layer. However, in both wired networks and MANETs, the distance function is defined in the same way.

From above property, a natural extension is: in MANETs, a node’s neighbor set is fixed at a given time, while for a node in a wire network, it could have countless variation. Therefore, in RN protocol in Section 6.3.3, the neighbor update procedure can only be applied in wired networks. It is absolutely not applicable in MANETs.
6.4.1.2 Substituent of IP Address

Substituent of IP address is necessary in MANETs for the purpose of building a P2P overlay. For example, source route in FAPSR, DPSR. The reason is intuitive: overlay layer only makes sense or semantically correct if an underlay layer exists.

6.4.1.3 Connected Component

For Chord or any other Structured P2P networks built on wired networks like Internet, all nodes are actually connected. Even though two nodes can not connect to each other or do not know the existence of the other for they do not know the network address of the other, they are still connected. However, this is not the case in MANETs. Nodes in MANETs are strictly constrained by the radio range in Physical layer. If there is no path from no node to another formed by neighborhood relations in a MANET, these two nodes are not reachable to each other unless their movement establish a path later. A MANET is consisted of a set of connected components, which are disjoint to each other. A component could contain only one node if the node is isolated. If a MANET has more than one component, there is no way to have one comprehensive Chord ring which includes every node like what always happen in Chord over Internet. The best scenario is we can find a Chord ring for each connected component.

6.4.1.4 Proximity

Both P2P over wired networks and P2P over MANETs have proximity concerns, but in MANETs this issue is has more serious impact. The reason is still from the physical layer characteristics. A hop in MANETs is more costly than in Internet. Hence Proximity optimization has more urgent, more realistic significance in MANETs.

6.4.1.5 Comprehensiveness

A P2P network over wired network, especially one over Internet, usually does not cover intermediate nodes of its path on the Network layer. Otherwise the P2P network may cover too many unrelated nodes. In a P2P network over a MANET, the situation is poles apart: all intermediate nodes should be included to secure connectivity on the overlay layer.
6.4.2 Cramer and Fuhrmann’s Pessimistic Verdict

In Cramer and Fuhrmann’s [CF2006], several serious problems could be found.

First, the whole paper is built upon some unrealistic, far-fetched assumptions. For example, they assume that all nodes can reach a common bootstrap node (which is called joint point) immediately after they power up. To make it possible, either all nodes in the MANET have to be only one hop away — which requires very small network or very powerful transmitter/receiver; or every ordinary node already has a route to that super node before power up, which is same as assuming that all nodes already have a pre-configured Chord successor table and finger table — so the network is already initialized, why does it need bootstrapping? Another example is the assumption of single bootstrap node, which is against the definition of MANET and cause the single point of failure.

The most unrealistic assumption is at the time of power up, that is, in their own words, in the first stabilization cycle, a Chord ring has been set up and all nodes have already joined the this ring in ID space. A minor assumption, which is not serious as other assumptions, is every node knows the size of the network n. Another untenable assumption is all nodes on the ring are in a complete sequential order, from 0 to n.

[CF2006] puzzles readers by its obvious detachment with MANETs, except in abstract, introduction, ad hoc networks or MANETs has not been mentioned. In simulation, only AODV is mentioned. How Chord is built upon AODV is not described.

6.5 RAN — An Optimal and Realistic Approach

6.5.1 Introduction

Ring Ad-hoc Network (RAN) is a protocol we developed in this dissertation to build a ring topology over MANETs. RAN has integrated merits from T-Chord, and Ring Network and adapt very well to MANETs. RAN is completely distributed. It uses only neighbors and local information. RAN builds an ideal ring topology for each connected component in the node ID space of a MANET. Upon this ring, entire Chord protocol could run immediately with optimal configuration at full speed; no extra stabilization is necessary unless large scale disturbance occurs. RAN integrates automatic non-IP address configuration into bootstrapping. We believe is the first successful attempt in the filed of bootstrapping structure P2P networks over MANETs.
The basic idea of RAN algorithm is distributed stepwise refinement. Each node treats its connected component as a tree, called component tree. All nodes in the component are included in this tree. It sets itself as the root. If the depth of a node in the tree is \( i \), the node is said at level \( i-1 \). At each step, we compare the current successor with either a random chosen node or all nodes in current level, depending on pattern of the algorithm is random or exhaustive. If a chosen node in current level has small distance to root, we use this closer node as new successor. The process repeats till the tree is traversed. Here the distance function is exactly same as define by Chord, also same as the distance function of RN.

We assume Chord ring is determined by the successor relation among nodes of a connected component. Unlike RN, in RAN the successor of a node is not always its neighbor. If the depth of a node \( n \)'s component tree is \( p > 1 \), the successor of \( n \) is \( n \)'s neighbor only at the first round of RAN execution. As RAN runs into deeper levels, the successor may change. The distance between \( n \) and its successor may be the depth of current level at most.

### 6.5.2 Design Goals and Assumptions

RAN is designed to achieve following goals:
- Have all capability of T-Chord and RN except those incompatible with nature of MANETs
- Pure distributed and decentralized, asynchronous, only use message passing
- Scalable to MANET size
- Very good proximity and optimized for MANETs
- Generate an idea Chord ring for each connected components, which will guarantee the quality of Chord running on these rings.
- No any kind of a priori bootstrapping node or bootstrapping
- Compatible to any working MANET routing protocols, that is, routing independent.

We assume a non-IP node ID configuration is already performed which generates unique random Chord ID in structured P2P layer. No Network layer address is needed. Routing in low layers uses this node ID as well. RAN integrates automatic address configuration into bootstrapping, which is often deliberately ignored in previous approaches by assuming that an ideal IP address configuration has been a priori established from the very beginning.

### 6.5.3 Component Tree

To make following discussion easier, we give a definition of component tree. A component tree is one spanning tree of the connected graph which is derived from the connected component. The rule of construction it is:

1. Select the searching node, which is looking up the closest successor, as the root.
2. Add all neighbors of the root to the first level of the component tree.
3. For all following levels, construct the next level according to the direct neighborhood relation.
4. Delete all edges which connect a lower level node to an upper level node, no matter if the former is a descendent of latter.

![Component Tree](image)

For a complete component tree with $N$ nodes, uniform downward degree $k$, and the depth $d$, following equations hold.

$$N = 1 + k + k^2 + k^3 + \cdots + k^d$$
N = \frac{k^{d+1} - 1}{k - 1} \quad (6.1)

Equivalently,

\[ d = \log_k (kN - N + 1) - 1 \quad (6.2) \]

The component tree is based only upon neighborhood relationship, so it already exists in any connected component. No extra construction is needed.

### 6.5.4 Three Patterns

To compare performance and find out intrinsic mechanism which determines the performance, we designed three patterns for RAN protocol. The primary concern is the balance between effectiveness and efficiency, to be specific, the trade-off between the completeness of the generated ring and the time, message, and storage complexities of construction.

Three patterns are studied in length. Two of them are exhaustive patterns, namely, distributed exhaustive pattern and virtual centralized exhaustive pattern. In virtual centralized exhaustive pattern the searching node acts as a central controller and coordinates the searching procedure. The two exhaustive patterns use unicast in message exchange, exhaustive search at each level of component tree. Output ring is guaranteed to be ideal Chord ring for every connected component. Because it compares all node identifiers in the component, the finding the closest successor is ensured. However, this exhaustion may suffer from high cost in time, message, and storage. We need measures to mitigate the overhead. These two exhaustive patterns are equally excellent in effectiveness. Both keep 100 percent nodes of connected component in ring constructed. The distributed exhaustive pattern has better performance than the virtual centralized exhaustive pattern due to the fact that nodes in distributed exhaustive pattern only exchange messages with parents and children.

The third pattern is random pattern, which has its root in Ring Network (RN) proposed by Shaker and Reeves [SR2005]. To adapt to MANETs environment, we eliminate the minimum bootstrapping system and use a breadth-first search scheme in lieu of it. The search scheme traverses the entire component tree of the searching node in a cascading manner in order to make up the poor effectiveness of RN.
### 6.5.4.1 Distributed Exhaustive Pattern

In the distributed exhaustive pattern, the searching node, which is at root of the component tree, sends a Closest Successor Request message to each direct child. Subsequently each child at the next level concurrently forwards the request message to its children one by one, preferably in a uniform order. At all following levels, nodes keep forwarding the Closest Successor Request message to next level children in the same manner until a leaf node is encountered. Then, from leaf nodes up, the closest successor of the root in the subtree is calculated at the root of the subtree by comparing the distance in node ID space of returned best candidates from the direct children of the subtree root. Then the closest successor information is returned to the parent node in a Closest Successor in Subtree message. This is a distributed cascading. For a complete component tree with uniform downward degree \( k \), except root and leaves, each node in the middle levels sends out \( k \) Closest Successor Request messages to its children, one Closest Successor in Subtree message to its parent, and receives \( k \) Closest Successor in Subtree messages from its children and one Closest Successor Request message from its parent. The message complexity of one node \( M \) satisfies

\[
M = 2(k + 1)N - 2k \times k^d - 2
= 2(k + 1)N - 2k^{d+1} - 2
\]

From (6.2), we know that

\[
k^{d+1} = kN - N + 1
\]

So,

\[
M = 2(k + 1)N - 2k^{d+1} - 2
= (2kN + 2N) - 2(kN - N + 1) - 2
= 4N - 4
\]

Hence, we have following theorem,

**Theorem 6.1**

In the distributed exhaustive pattern, the message complexity of one node \( M \) in a complete component tree with uniform downward degree is independent of the downward degree \( k \). It is only related to the size of network and is given
Obviously, the network message complexity $M_{Net}$ satisfies

$$M_{Net} = MN = (4N - 4)N$$

That is,

$$M_{Net} = 4N^2 - 4N$$

(6.5)

The time complexity $T$ of the distributed exhaustive pattern in a complete component tree with downward degree $k$ is

$$T = T_d + T_u$$

(6.6)

where $T_d$ is time used in downward transmission and $T_u$ is time used in upward transmission.

In a MANET, nodes can run concurrently. Let’s look at the root of the component tree. Suppose it sends the Closest Successor Request message to its children in an order from left to right. All other children except the rightmost one have started their own work when the root finishes its transmission. It is clear that till all operation at the second level is finished the time this approach costs is the sum of the time the root spent in transmission (i.e. $k$) and the time the rightmost node costs after the root completes its transmission, which is $k$ as well. Suppose transmission of a message needs one more time unit. We have

$$T_d = dk$$

(6.7)

and

$$T_u = d$$

(6.8)

From (6.7) and (6.8), we get

$$T = dk + d$$

That is,

$$T = d(k + 1)$$

(6.9)
Component trees of all nodes can run in asynchronous but simultaneous manner. Nodes in above outlined algorithm are idle in most time. So there is little chance of traffic jam caused by contention of messages from different component trees. Therefore, we have following theorem.

**Theorem 6.2**

In the distributed exhaustive pattern, the time complexity of whole network is same as the time complexity of one node, which is $d(k + 1)$.

$$T_{\text{Net}} = d(k + 1) \quad (6.10)$$

Since $d = \log_k (kN - N + 1) - 1$,

$$T_{\text{Net}} = T = (k + 1)d$$

$$= (k + 1)(\log_k (kN - N + 1) - 1)$$

$$\approx (k + 1)\log_k ((k - 1)N) - k - 1$$

$$= (k + 1)\log_k N + (k + 1)\log_k (k - 1) - k - 1$$

Hence, we have

$$T = O(\log_k N)$$

That is

$$T = O(\log N) \quad (6.11)$$

and

$$T_{\text{Net}} = O(\log N) \quad (6.12)$$

**Theorem 6.3**

The distributed exhaustive pattern has the time complexity of $O(\log N)$.

If we use multicast option (see detailed description in Section 6.5.4.2) to send only one Closest Successor Request message to all children, for a complete component tree with uniform downward degree $k$, except root and leaves, each node in the middle levels sends out one Closest Successor Request messages to its children, one Closest Successor in Subtree message to its parent, and receives $k$ Closest Successor
in Subtree messages from its children and one Closest Successor Request message from its parent. The message complexity for one node $M$ satisfies

$$M = (k + 3)N - (k + 1) \times k^d - 2$$
$$= (k + 3)N - \frac{(k + 1)}{k} k^{d+1} - 2$$  (6.13)

From (6.2), we know that

$$k^{d+1} = kN - N + 1$$

So,

$$M = (k + 3)N - 2k^{d+1} - 2$$
$$= (kN + 3N) - \frac{(k + 1)}{k} (kN - N + 1) - 2$$
$$= (kN + 3N) - (1 + 1/k)(kN - N + 1) - 2$$
$$= (kN + 3N) - ((kN + N) - (N + N/k) + (1 + 1/k)) - 2$$
$$= (kN + 3N) - (k - 1/k)N - (1 + 1/k) - 2$$
$$= 3N + N/k - 1/k - 3$$

$$M = (1/k + 3)(N - 1)$$  (6.14)

The network message complexity $M^{\text{Net}}$ follows

$$M^{\text{Net}} = MN = (1/k + 3)(N - 1)N$$  (6.15)

For time complexity,

$$T_d = T_u = d$$  (6.16)

So

$$T = 2d$$  (6.17)

Plug in (6.2),

$$T = 2 \log_k (kN - N + 1) - 2$$

Like the case for unicast,
In the virtual centralized exhaustive pattern, all direct children nodes of the root form the first level set. One by one, the root node sends an All Neighbors Request message to each direct child at the first level and asks them to return their direct downward neighbor set (excluding any node in upper levels) in the component tree. After receiving all direct neighbor sets from nodes at current level, the root sets the union of all these returned neighbor sets as its next level set. Then it sets the next level as current level, and does the same thing to them, till leaves are reached. This algorithm is most expensive in terms of overhead in time and message. However, it gives the root node tremendous power to control whole process upon pure distributed network configuration. In real world application, a user could provide individualized service if virtual centralized exhaustive pattern is used to construct the ring topology. So in our simulation, we did considerable studies for this pattern as well.

For a complete component tree with uniform downward degree $k$, except root and leaves, each node in the middle levels receives a getAllNeighbors message from the root and sends out one AllNeighbors message to the root. If we do not care about the hops of message, the message complexity of one node $M$ satisfies

$$M = 2(N - 1)$$  \hspace{1cm} (6.19)

Subsequently,

$$M^{Net} = 2(N - 1)N$$  \hspace{1cm} (6.20)

However, here messages are not like those in the distributed exhaustive pattern, almost all messages have to go through multi-hops. If we want get precise comparison with the distributed exhaustive pattern, the message per hop should be used.

$$M_{hop} = 2k + 4k^2 + 6k^3 + \cdots + 2dk^d$$

$$= 2\sum_{i=1}^{d} ik^i$$  \hspace{1cm} (6.21)

Obviously,
\[ M_{\text{hop}}^{Net} = NM_{\text{hop}} = N(2k + 4k^2 + 6k^3 + \cdots + 2dk^d) \]

That is,

\[ M_{\text{hop}}^{Net} = 2N \sum_{i=1}^{d} ik^i \quad (6.22) \]

Same as previous assumption, one hop takes one time unit to cross. A round trip of the `getAllNeighbors` message and `AllNeighbors` message at first level takes \( k + 1 \) time units. At second level, it is \( k^2 + 2 \). At third level, it is \( k^3 + 3 \). And so on, till \( k^d + d \). So the time complexity is

\[
T = (k + 1) + (k^2 + 2) + (k^3 + 3) + \cdots + (k^d + d)
\]

\[
= (1 + k + k^2 + k^3 + \cdots + k^d) + (1 + 2 + 3 + \cdots + d) - 1
\]

Since \( N = (1 + k + k^2 + k^3 + \cdots + k^d) \), we get

\[ T = N + d(d + 1)/2 - 1 \quad (6.23) \]

Similarly,

\[ T^{Net} = T = N + d(d + 1)/2 - 1 \quad (6.24) \]

### 6.5.4.3 Random Pattern

Basically our purpose to create random pattern is to simulate Ring Network (RN) in MANETs. To make our version of RN suitable and optimized for MANETs, we made several changes. First, to make it completely decentralized, self-organized, and automatic, we eliminated the minimum bootstrapping system, which is pivotal to RN. In fact, it is impossible to implement the minimum bootstrapping system even though we did wish to do so. Unless we accept the assumption that all nodes in a MANET are connected together all the time, which is impossible, such a minimum bootstrapping system would not be found. And there is no description of it in all media. Second, in place of the minimum bootstrapping system, we employ a breadth-first search scheme, which traverses the entire connected component of the searching node in a cascading manner, level by level from its direct neighbors. This scheme remedies the problem of limited comprehensiveness.

For our exhaustion like random pattern in a complete component tree with uniform downward degree \( k \), the per node message complexity \( M \) consists of \( M_s \) messages to
trigger the component members to take part in the search, and $M_{\text{search}}$ messages do searching for each component member.

The first part is done through a cascading procedure: each node except the root receives a Start message with root information included, so

$$M_S = N - 1$$

The second part is the sum of a node’s distance to bottom and $d$ each node except root,

$$M_{\text{search}} = [((d-1)k + (d-2)k^2 + \cdots + k^{d-1}) + (N-1)]$$

$$= d(k + k^2 + \cdots + k^{d-1}) - [((k+2k^2 + \cdots + (d-1)k^{d-1}) + (N-1)]$$

So

$$M = M_S + M_{\text{search}} = d(k + k^2 + \cdots + k^{d-1}) - [((k+2k^2 + \cdots + (d-1)k^{d-1}) + 2(N-1)$$

The time complexity is

$$T = \frac{k + k + \cdots + k + d}{d}$$

That is,

$$T = (k + 1)d \quad (6.25)$$

Similarly,

$$T^{\text{Net}} = (k + 1)d \quad (6.26)$$

It is exactly same as those of the distributed exhaustive pattern. So we also have

$$T = O(\log N) \quad (6.27)$$

$$T^{\text{Net}} = O(\log N) \quad (6.28)$$

Hence we have following theorem

**Theorem 6.4**
The random pattern has the same time complexity \((k + 1)d\) as the distributed exhaustive pattern. Generally they are in the family of \(O(\log N)\), which is the best in all possibilities.

### 6.5.5 Two Options

Besides three patterns just described, we defined two auxiliary options to improve the efficiency, basically targeting time complexity and message complexity.

#### 6.5.5.1 Approximation Option

The approximation option could be applied to all three patterns. Approximation pattern does not change underlying algorithm. It works by changing the end condition of all patterns. End condition in the approximation option is much looser than normal scenario. Normally, all patterns set the ideal ring as their objective. With approximation option, a small fraction of nodes are allowed to be left out of the final rings if they are in very short line segments attached to rings.

After first running of check_rings() function in the simulator, a connected component breaks down to a ring and a set of lines which are attached to the ring at only one node. With the running of the simulation, the lines gradually shrink and are absorbed by the ring. Finally with sufficient running of our simulator only ring exists. In the plain pattern, we require that all lines are absorbed by the corresponding ring of the connected component. However, this approach becomes so resource demanding when network size increases over 100 nodes. To reduce overhead in time, message, and storage, we revise the ending condition to allow a small fraction of nodes of a component to remain in short lines. Usually the fraction is set to 10 percent, or 15 percent. This approximation tremendously reduced the complexity in time, storage, and message. The growth function of time versus network size dropped from sub exponential to linear. Similar improvement happened to the growth function of the number of sent or received messages versus network size.

#### 6.5.5.2 Multicast Option

Another option is multicast option, in which a node sends message to all direct downward neighbor nodes (its children) at next level by multicasting one message instead of unicasting multiple messages serially. It considerably improves time, message, and storage complexity.
However, as we mentioned above, multicast option cannot be applied to any random pattern.

Please refer to Chapter 7 for detailed algorithm and simulation.
CHAPTER 7 RAN ALGORITHM AND SIMULATION

7.1 Algorithms

7.1.1 Distributed Exhaustive Pattern

7.1.1.1 Message Format

getBestCandidate(id, sender, receiver)

id is the ID of searching node (root of component tree)
sender: sender of this message, not necessarily the searching node
receiver: receiver of this message

BestCandidate(id, candidate, sender, receiver)

id is the ID of searching node (root of component tree)
candidate: best candidate returned
sender: sender of this message, not necessarily the searching node returned
receiver: receiver of this message

7.1.1.2 Algorithm

peer u

constant
maximum: upper bound of ID

input
init: initialization flag, set to true at beginning
size: number of nodes in the connected MANET
in-que-len: length of incoming message queue
out-que-len: length of outgoing message queue
var
  in-queue: incoming message queue
  out-queue: outgoing message queue
  \( \Gamma \): set of one-hop neighbors
  \( \Gamma_0 \): successor
  returned_msg: number of responses returned to getBestCandidate messages sent by this node
  id: ID of searching node (root of component tree)
  root_received: A set of all received
  best_candidate(id): current best candidate for id
  best_candidate_distance(id): ID space distance from id to best_candidate(id)

Library Function
  lookfor(x, y): Return x if \( x \in y \). Return\(<x, *>\) if \(<x, *>\) \( \in y \). Otherwise return NIL.

Action
(a1) init \rightarrow
  construct \( \Gamma \)
  if \( \Gamma \neq \emptyset \) then
    init := false
    returned_msg := 0
    best_candidate(u) := NIL
    best_candidate_distance(u) := maximum
    for each \( h \in \Gamma \) do
      send getBestCandidate(\( u, u, h \)) to \( h \)
  []

(a2) receive getBestCandidate(id, q, u) from q \rightarrow
  if id \in root_received return
  else
    root_received := root_received + \{<id, q>\}
    best_candidate(id) := u
    best_candidate_distance(u) := (u – id) MOD maximum
    for each \( h \in (\Gamma - \{q\}) \) do
      send getBestCandidate(id, u, h) to \( h \)
  []

(a3) receive BestCandidate(id, cd, q, u) from q \rightarrow
  returned_msg ++
\[ d := (cd - id) \mod \text{maximum} \]
\[ \text{if } d < \text{bestcandidate}(id) \]
\[ \text{then} \]
\[ \text{bestcandidate}(id) := cd \]
\[ \text{bestcandidate}_\text{distance}(id) := d \]
\[ \text{if } u \neq id \text{ and } \text{returned_msg} = |T| - 1 \]
\[ \text{then} \]
\[ \text{if } \text{lookfor}(id, \text{rootReceived}) \neq \text{NIL} \]
\[ <id, x> := \text{lookfor}(id, \text{rootReceived}) \]
\[ \text{send BestCandidate}(id, \text{bestcandidate}(id), u, x) \]

7.1.2 Virtual Centralized Exhaustive Pattern

7.1.2.1 Message Format

The format of \text{getAllNeighbors} here is

\[ \text{getAllNeighbors}(\text{originator}, \text{sender}, \text{receiver}, \text{route}) \]

\text{originator}: the root node.
\text{sender}: sender of this message, not necessarily the searching node
\text{receiver}: receiver of this message
\text{route}: the route from the \text{originator}

The format of \text{allNeighbors} here is

\[ \text{allNeighbors}(\text{sender}, \text{originator}, \text{neighbors}, \text{broute}) \]

\text{originator}: \text{originator} of received corresponding \text{mGetAllNeighbors} message
\text{neighbors}: all neighbors except the sender of corresponding \text{mGetAllNeighbors} message
\text{broute}: route from current node to \text{originator}

7.1.2.2 Algorithm

peer \( u \)

input
init: initialization flag, set to true at beginning
size: number of nodes in the connected MANET
in-que-len: length of incoming message queue
out-que-len: length of outgoing message queue
msg-rate: message processing rate

Note: Assuming rates for incoming messages and outgoing messages are same.

var
  in-queue: incoming message queue
  out-queue: outgoing message queue
  T: external timer, simulator by discrete counter
  L: set of peers at the current level
  N: set of neighbors, including all hops
  level: current level of hops from u
  Γ: set of one-hop neighbors
  Γ₀: successor
  R: u’s routing records for this algorithm
  r: a route in R
  s: a node in a set with smallest node ID
  AN_received: number of received allNeighbor messages
  AN_in_queue: number of allNeighbor messages in in_queue
  nodes_last_level: node number in previous level
  current_completed: if all searching is completed at current level

Library Function
  route(a, b): return a route from node a to node b. Actually a route is a string or
  vector. route(a, a) returns a.
  distance(u, h): RAN distance function
  reverse(r): return the reverse path of route r

Action
(a1) init →
  construct Γ
  if Γ ≠ ∅
  then
    find Γ₀
    T := 0
    init := false
    R := ∅
for each $h$ in $\Gamma$ do
  $R := R + \{uh\}$
  $N := L := \Gamma$

Note: Here $route(u, h) = uh$. $uh$ is a series, like string, vector in C++, ArrayList in Java.

(a2) $current\_completed \rightarrow$
  $level \leftarrow \text{false}$
  $T := 0$
  for each $h \in L$ do
    if $route(u, h) \in R$
      then send $\text{getAllNeighbors}(u, u, h, route(u, h))$ to $h$
  $L := \emptyset$

(a3) receive $\text{getAllNeighbors}(o, q, u, r)$ from $q \rightarrow$
  $br := \text{reverse}(r)$
  send $\text{allNeighbors}(u, o, \Gamma - \{\text{all nodes in } r\}, br)$ to $o$

(a4) receive $\text{allNeighbors}(q, o, S, br)$ from $q \rightarrow$
  for each $h$ in $(S - N)$ do
    $route(u, h) := route(u, q) + h$
    $R := R + \{ route(u, h)\}$
  $L := L + (S - N)$
  $N := N + (S - N)$

(a5) $true \rightarrow$

Local

timeout small: lower bound of ending time
timeout big: upper bound of ending time

timeout small := $2 \times level \times \text{nodes\_last\_level} / \text{msg\_rate}$
timeout big := $\max\{4 \times level \times \text{nodes\_last\_level} / \text{msg\_rate}, 6 \times \log(\text{size})\}$
if $(AN\_received = \text{nodes\_last\_level})$
  or $(T >= \text{timeout\_small})$ and $(AN\_in\_queue = 0))$
  or $(T >= \text{timeout\_big})$
then
  $current\_completed := true$
for each \( \text{allNeighbors}(q, u, S, br) \) message still in \textit{in-queue}

take out \( \text{allNeighbors}(q, u, S, br) \)

(a4)

\[ s := \text{argmin}_{k \in L} d(u, k) \]

\[ \text{if } d(u, s) < d(u, \Gamma_0) \text{ then } \Gamma_0 = s \]

[]

(a6) \textbf{receive} a message destined for another node \( \rightarrow \)

\textbf{send} the message to next node on the route

[]

Note: For \textbf{receive} primitives, actual triggering event: message is taken out from \textit{in-queue}.

**7.1.3 Virtual Centralized Exhaustive Pattern with Multicast Option**

**7.1.3.1 Message Format**

Suppose a node always sends multicast messages at low frequency, so there is no need of an out queue for sending multicast messages. Only in-queue is needed for receiving multicast messages from other nodes. We also suppose multicast messages have priority over normal messages, they could use all msg-rate to process multicast in queue if needed.

There is only one kind of multicast messages, that is, \( m\text{GetAllNeighbors} \). The format of \( m\text{GetAllNeighbors} \) message is

\[ m\text{GetAllNeighbors}(\text{originator}, \text{serial\_number}, \text{sender}, \text{depth}, \text{back\_route}) \]

\textit{originator}: the root node.

\textit{serial\_number}: a random number used to find out later repeated coming of a same multicast message from a same originator.

\textit{sender}: the forwarding node of the message.

\textit{depth}: \textit{sender}'s depth in the broadcasting tree, which is a spanning tree converted from the current connected component with root at the querying node.

\textit{back\_route}: the route back to the \textit{originator}

The format of \( \text{allNeighbors} \) here is

\[ \text{allNeighbors}(\text{sender}, \text{originator}, \text{neighbors}, \text{serial\_number}, \text{depth}, \text{route}) \]
originator: originator of received corresponding mGetAllNeighbors message
neighbors: all neighbors except the sender of corresponding mGetAllNeighbors message
serial_number: serial_number of corresponding mGetAllNeighbors message
depth: depth of current node
route: route from current node to originator

7.1.3.2 Algorithm

peer u

Const

\begin{align*}
\text{init:} & \text{ initialization flag, set to true at beginning} \\
\text{in-que-len:} & \text{ length of incoming message queue} \\
\text{out-que-len:} & \text{ length of outgoing message queue}
\end{align*}

input

\begin{align*}
\text{size:} & \text{ number of nodes in the connected MANET} \\
\text{msg\_rate:} & \text{ message processing rate} \\
x: & \text{ the querying node at the root of the broadcasting tree} \\
\text{max\_depth:} & \text{ maximum depth, usually log(size), at most size} \\
\text{timeout:} & \text{ upper bound of running time of whole procedure}
\end{align*}

Note:
Assume rates for incoming messages and outgoing messages are same

var

\begin{align*}
\text{in\_queue:} & \text{ incoming message queue} \\
\text{brd\_in\_queue:} & \text{ incoming multicast message queue} \\
\text{out\_queue:} & \text{ outgoing message queue} \\
T: & \text{ external timer, simulator by discrete counter} \\
N: & \text{ set of neighbors, including all hops} \\
\Gamma: & \text{ set of one-hop neighbors} \\
\Gamma_0: & \text{ successor} \\
R: & \text{ u’s routing records for this algorithm} \\
r: & \text{ a route in } R \\
s: & \text{ a node in a set with smallest node ID} \\
\text{current\_completed:} & \text{ if all searching is completed at current level}
\end{align*}
**Library Function**

- **route(a, b):** return a route from node a to node b. Actually a route is a string or vector. route(a, a) returns a.
- **distance(u, h):** RAN distance function
- **reverse(r):** return the reverse path of route r

**Action**

(a1) **init →**

```
    construct Γ
    if Γ ≠ ∅ then
        Γ0 := 0
        init := false
        rnd := getRandomNum();
        back_route := route(u, u)
        multicast mGetAllNeighbors(u, rnd, u, 0, back_route)
        R := ∅
    for each h in Γ do
        R := R + {uh}
        Γ := Γ - {uh}
    •
```

- Note:
  Here route(u, h) = uh. uh is a series, like string, vector in C++, ArrayList in Java

(a2) **receive mGetAllNeighbors →**

    if (<originator, serial_number> ∉ received_brdcst) and (depth < max_depth)
    back_route := back_route + route(sender, u)
    received_brdcst := received_brdcst + {<originator, serial_number>}
    multicast mGetAllNeighbors(originator, sn, u, depth + 1, back_route)
    route := reverse(back_route)
    send allNeighbors(u, originator, Γ - {sender}, serial_number, depth + 1,
    route) to originator

•

(a3) **receive allNeighbors(q, u, S, sn, d, rt) from q →**

    for each h in (S - N) do
route(u, h) := route(u, q) + h
R := R + { route(u, h)}
if distance(u, h) < distance(u, Γ₀) then Γ₀ := h
N := N + (S − N)

(a4) **receive** a non-multicast message destined for another node → **send** the message to next node on the route

(a5) T > timeout →
    disable (a1), (a3), and (a5)
    stop the whole procedure for u

**Note:**
For all **receive** primitives, actual triggering event is: message is taken out from **in-queue**.

### 7.1.4 Random Pattern

#### 7.1.4.1 Discussion about Random Pattern

The basic idea in random pattern is to seek high efficiency, faster convergence time, i.e. build topology faster, instead of completeness. In another word, random pattern prefers speed to the quality of ring.

In this pattern, at each level of the search tree (a spanning tree rooted at the node which is searching for the closer successor), we do not search every node like in plain pattern. We pick up only one.

One way to do it is: always pick up the closest neighbor of current node. However, since node ID is assigned randomly from a huge ID space, which has nothing to do with a node’s other properties like neighborhood. The implied strategy behind this approach — a node x closer to u may has neighbor even more closer to u — is not tenable.

However, if we use pure random selection, we will lose the ending condition. Maybe we can use the depth of searching path as an end condition.
From this point of view, RN may have chosen a better approach. Their closer peer search actually search a node similar to the predecessor of \( u \), the current node which is searching closer successor. Since the successor of \( u \)’s predecessor definitely has better chance to be close to \( u \).

### 7.1.4.2 Random Pattern Message Format

The format of `closerNodeSearch` message is

\[
\text{closerNodeSearch}(\text{originator, serial\_number, sender, depth, back\_route})
\]

- `originator`: the root node.
- `serial\_number`: a random number used to find out later repeated coming of a same multicast message from a same originator.
- `sender`: the forwarding node of the message.
- `depth`: `sender`’s depth in the component tree, which is a spanning tree converted from the current connected component with root at the querying node.
- `back\_route`: the route back to the `originator`.

The format of `successorCandidate(\Gamma_0)` message is

\[
\text{successorCandidate}(\text{sender, successor, receiver, serial\_number, route})
\]

- `sender`: the sender, i.e. current node.
- `successor`: the successor of current node.
- `receiver`: receiver of this `successorCandidate` message. It should be the `originator` of its received `closerPredecessorSearch` message.
- `serial\_number`: `serial\_number`.
- `route`: the route from current node to the `originator`.

### 7.1.4.3 Random Pattern Algorithm

**Pattern Random**

**Version 3**

`peer u`
**type (class)**

- `component_node`: element of variable `component_queue`
- `component_node.nodeIdx`: node index, internal expression, not node ID.
- `component_node.traversed`: Indicates if a node has been traversed in current searching node’s RAN-Random execution.
- `component_node.in_route`: record route from current searching node to this node.

**input**

- `max_depth`: maximum depth, usually \( \log(size) \), at most \( size \)
- `timeout`: upper bound of running time of whole procedure

**var**

- `init := true`: mark the very beginning of algorithm
- `in_queue`: incoming message queue
- `out_queue`: outgoing message queue
- \( S \): Set of nodes
- \( B \): Set of successor candidates
- \( W \): Set of successor candidates
- \( w \): a node (successor candidate)
- \( x \): peer being searched for
- \( T \): external timer, simulator by discrete counter
- \( \Gamma \): set of one-hop neighbors
- \( \Gamma_0 \): successor
- \( R \): u’s routing records for this algorithm
- \( s \): a node
- `rnd`: a random number
- `new_round`: indicates this round should stop and a new round should be started
- `component_queue`: The BFS connected component queue, used as the source set to feed the `closerNodeSearch`
- `dumped_component`: dumped_component contains those nodes poped out from `component_queue`

**Library Function**

- `route(a, b)`: return a route from node \( a \) to node \( b \). Actually a route is a string or vector.
  - `route(a, a)` returns \( a \).
- `distance(u, h)`: RAN distance function
- `reverse(r)`: return the reverse path of route \( r \)
- `empty(x)`: return true if set or series \( x \) is empty, otherwise return false
Action
(a1) \(\text{init} \rightarrow\)

Local
\(cn: \text{component node}\)

\(\text{init} := \text{false}\)
\(\text{construct } \Gamma\)
\(in\_queue := \emptyset\)
\(out\_queue := \emptyset\)
\(\text{if } \Gamma = \emptyset\)
\(\text{then}\)
\(\text{new\_round} := \text{false}\)
\(\text{return}\)
\(\text{else}\)
\(\text{find } \Gamma_0\)
\(\text{new\_round} := \text{true}\)
\(\text{component\_queue} := \emptyset\)
\(\text{dumped\_component} := \emptyset\)
\(\text{for each } h \in \Gamma\)
\(\text{cn} := \text{new(component\_node)}\)
\(\text{cn}.\text{nodeIdx} := h;\)
\(\text{cn.traversed} := \text{true}\)
\(\text{cn.in\_route} := \{u\} + \{h\}\)
\(\text{push\_back(component\_queue, cn)}\)

[]

(a2) \(\text{new\_round \ and (not empty(component\_queue))} \rightarrow\)
\(\text{new\_round} := \text{false}\)
\(B := W := \emptyset\)
\(cn1 := \text{pop\_front(component\_queue)}\)
\(\text{push\_back(dumped\_component, cn1)}\)
\(\text{rnd} := \text{getRandomNum}();\)
\(\text{back\_route} := \text{route}(u, u)\)
\(\text{send} \text{closerNodeSearch}(u, \text{rnd, u, 0, back\_route)} \text{to} \ cn1\)
\(\text{for each } h \in cn1.\Gamma\)
\(\text{if } h \notin (\text{component\_queue} \cup \text{dumped\_component})\)
\(\text{then}\)
\(\text{cn2} := \text{new(component\_node)}\)
\(\text{cn2}.\text{nodeIdx} := h;\)
\(\text{cn2.traversed} := \text{true}\)
\(\text{cn2.in\_route} := \{u\} + \{h\}\)
\(\text{push\_back(component\_queue, cn2)}\)
(a3) receive closerNodeSearch(x, sn, q, depth, back_route)) from q →

Local

send_candidate := false

if (depth >= max_depth)
    then
    send_candidate := true
    new_round := true
else if u is closer to x than any u’s neighbor ∈ u.Γ
    then send_candidate := true
if send_candidate = true
    then
    B := B ∪ {x}
    back_route := back_route + route(q, u)
    route := reverse(back_route)
    send successorCandidate(u, Γ₀, x, sn, route) to x
else
    back_route := back_route + route(sender, u)
    send closerNodeSearch(x, sn, u, depth + 1, back_route) to argmin_k∈Γ d(k, x)

(a4) receive successorCandidate(q, w, org, sn, route) from q →

if u = org
    then
    W := W ∪ {w}
    S := W ∪ B ∪ Γ
    Γ₀ := argmin_k∈S d(u, k)
    new_round := true
else send the message to next node on the route

7.2 Simulation

To simplify the programming, the simulation is based upon static network. We recommend bootstrapping is launched at a relatively less mobile setting; since mobility will change the composition of connected components. Without stable connected components topology is like a tree without the earth, all efforts becomes futile. All discussion involving mobility must be based upon the premise that the change of components caused by mobility disturbance should be limited to a
reasonable range. The premise validates that early research could be based upon static assumption.

This simulator is not for one single connected component; instead it is for a MANET randomly generated on a 100 × 100 two–dimensional square. Each node is independently generated with node x coordinate and y coordinate uniformly distributed in range [0, 100], and node ID uniformly distributed in range [0, 65535]. The connectivity between two nodes is purely decided by their Euclidean distance and the uniform radio range for all nodes in the MANET. We believe it is much more realistic than generated only one connected component.

Basic parameters tested include completeness, time, number of sent messages, and number of received messages. The completeness examines the effective of algorithm by checking completeness of rings generated. Time is the time used to construct the rings. Two messages measure the message complexity.

### 7.2.1 Completeness

Completeness is the ratio of number of nodes in generated rings to number of all nodes. Our simulation shows that all nodes are either in rings, or in lines. Each line is connected to one and only one ring. Isolated nodes are regarded as rings, so they are always counted as constructed. This is rational for some MANETs which is unfortunately initialized with considerable isolated nodes. Each connected component has one or more rings which are connected by lines. Basic point here is: at any time of the construction, even before anything is done for the construction, all nodes in same component should always in the same component. The construction only changes the number of rings and the nodes in rings in the component. It does not change the component, as the component is defined by the neighborhood relation among nodes, which remains identical in a static network.

<table>
<thead>
<tr>
<th>Network Size</th>
<th>Random Pattern</th>
<th>Distributed Exhaustive Pattern</th>
<th>Centralized Exhaustive Pattern(Plain)</th>
<th>Centralized Exhaustive Pattern(Approximation 0.85)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0.93</td>
<td>1</td>
<td>1</td>
<td>0.94</td>
</tr>
<tr>
<td>40</td>
<td>0.705</td>
<td></td>
<td>1</td>
<td>0.89</td>
</tr>
<tr>
<td>60</td>
<td>0.547</td>
<td></td>
<td>1</td>
<td>0.87</td>
</tr>
<tr>
<td>80</td>
<td>0.395</td>
<td></td>
<td>1</td>
<td>0.88</td>
</tr>
<tr>
<td>100</td>
<td>0.343</td>
<td></td>
<td>0.98</td>
<td>0.855</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0.97</td>
<td>0.86</td>
</tr>
</tbody>
</table>

Table 7.1 Completeness of Algorithms
7.2.2 Time

Table 7.2 Time used in ring construction

<table>
<thead>
<tr>
<th>Network Size</th>
<th>20</th>
<th>40</th>
<th>60</th>
<th>80</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Pattern</td>
<td>11.8</td>
<td>15</td>
<td>32</td>
<td>63.6</td>
<td>91</td>
</tr>
<tr>
<td>Distributed Exhaustive Pattern</td>
<td>8.3</td>
<td>21.8</td>
<td>43</td>
<td>87.4</td>
<td>98.5</td>
</tr>
<tr>
<td>Centralized Exhaustive Pattern (Plain)</td>
<td>5.8</td>
<td>56.4</td>
<td>1091.4</td>
<td>2552</td>
<td>21409</td>
</tr>
<tr>
<td>Centralized Exhaustive Pattern</td>
<td>6</td>
<td>49.6</td>
<td>313.8</td>
<td>632</td>
<td>910</td>
</tr>
</tbody>
</table>

Time is defined as the algorithmic time used in one run of simulation, from beginning to end. The critical question is how the end condition is defined in simulation. As we mentioned in Section 6.5.5.1, normally the end condition is defined by the formation of ideal Chord ring, which is unique and fixed for a given MANET. For a pattern that needs too much time to finish, the end condition could be adapted by using the approximation option. For all combinations of patterns and options, the algorithmic time unit is set as virtually synchronized discrete time unit. In each unit, all nodes are supposed to complete the processing of \( m \) incoming messages and \( n \) outgoing messages. Except in multicast option, \( m \) is assumed to
equal to \( n \). \( m \) and \( n \) are determined by the simulation parameter message processing rate.

![Graph showing time used in ring construction](image)

Figure 7.2 Time used in ring construction

### 7.2.3 Message Complexity

Messages complexity is measured by two parameters: messages sent and messages received. The message sent is defined as total number of messages sent by all nodes in the network during the simulation. The message received is defined as total number of messages received by all nodes in the network during the simulation.

<table>
<thead>
<tr>
<th>Network Size</th>
<th>20</th>
<th>40</th>
<th>60</th>
<th>80</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Pattern</td>
<td>72.8</td>
<td>1117.8</td>
<td>4721</td>
<td>13839</td>
<td>25117.3</td>
</tr>
<tr>
<td>Distributed Exhaustive Pattern</td>
<td>103.6</td>
<td>1355.1</td>
<td>5089.8</td>
<td>16077</td>
<td>26853.5</td>
</tr>
<tr>
<td>Centralized Exhaustive Pattern (Plain)</td>
<td>412</td>
<td>15863.4</td>
<td>162705.6</td>
<td>465754.6</td>
<td>2581659.6</td>
</tr>
<tr>
<td>Centralized Exhaustive Pattern (Approximation 0.85)</td>
<td>226</td>
<td>5556.8</td>
<td>62625.4</td>
<td>186703</td>
<td>364112.2</td>
</tr>
</tbody>
</table>

Table 7.3 Messages Sent
Figure 7.3 Messages sent

Table 7.4 Messages Received

<table>
<thead>
<tr>
<th>Network Size</th>
<th>20</th>
<th>40</th>
<th>60</th>
<th>80</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Pattern</td>
<td>72.8</td>
<td>1117.8</td>
<td>4721</td>
<td>13839</td>
<td>25117.3</td>
</tr>
<tr>
<td>Distributed Exhaustive Pattern</td>
<td>98</td>
<td>1312.8</td>
<td>5045</td>
<td>15938.3</td>
<td>26783</td>
</tr>
<tr>
<td>Centralized Exhaustive pattern</td>
<td>394.6</td>
<td>15639</td>
<td>161335.2</td>
<td>460179.4</td>
<td>2539312.2</td>
</tr>
<tr>
<td>Centralized Exhaustive Pattern (Approximation 0.85)</td>
<td>209.8</td>
<td>5452.2</td>
<td>61693.8</td>
<td>183749.6</td>
<td>358354</td>
</tr>
</tbody>
</table>

Figure 7.4 Messages received
7.2.4 Analysis of Simulation Results

Obviously, the winner is the distributed exhaustive pattern. It shows perfect effectiveness, at the same time and unlike the other two patterns, it has no serious side effect to endanger its advantage. The other two, however, suffer from different fatal problems. For random pattern, it is the effectiveness. For virtual centralized exhaustive pattern, it is efficiency.

As shown in preceding sections, it is clear that in RAN family, random pattern has the best overhead cost in both time complexity and message complexity. However, it is also the worst approach in terms of quality of ring constructed. The reason is behind its searching strategy. The closest first criterion does not make sense in a pure stochastic uniform distribution of ID space. The closest successor could be hidden anywhere in the component tree. It could be child of any node. It may be child of current closest successor, or child of current farthest successor, or child of any other node. The lesson is: in face of such complete randomness, exhaustion in search is necessary. It has been illustrated clearly. Exhaustive approaches almost always returns the best rings, unless we intentionally prohibit it from doing so with an approximation.

Nevertheless, the simulation tests also demonstrated the terrible efficiency of centralized attempt. The virtual centralized exhaustive pattern is worst in terms of cost in time and message. On the other hand, its twin approach, the distributed exhaustive pattern shows tremendous divergent result. It proved that exhaustion is not a synonym of expenditure, as our intuition alarmed. It does not mean inferior efficiency at all. The distributed exhaustive pattern has almost same efficiency as the random pattern.

The third enlightenment is at a high level of abstraction, it is kind of philosophy. As we always advocate, the benefit of decentralization has shown by the distributed exhaustive pattern. This result also raises a question, also at high level of philosophy, that is: could centralization be implemented above decentralized infrastructure?
CHAPTER 8 CONCLUSION

8.1 Conclusion

This dissertation research targeted at the important intersection point of two popular research topics in computer networks, namely, peer-to-peer networks and mobile ad-hoc networks. In this dissertation, we have investigated existing successful and failed research efforts in synergy of P2P networks and MANETs and made two ground-breaking contributions to the specific field of bootstrapping P2P networks over MANETs.

Our first contribution is the proposal of a non-IP automatic address configuration scheme. We analyzed advantages of this scheme comparing to other IP-based dynamic addressing approaches. We found that IP-based schemes waste the precious bits in MANETs. The requirement for IP address unnecessarily makes the addressing task more complicated. The only benefit of keeping IP in MANETs is it is easier to interface with present Internet and various existing applications. However, these excuse hardly forms any obstacle for our non-IP addressing scheme. The non-IP addressing can easily circumvent Internet connection problem by using a few wired network gateway. For application compatibility, we can just use a virtual mapping layer to convert IP address and our non-IP address into each other, such that traditional IP-based applications could be kept in use above our proposed scheme.

Along with the non-IP addressing scheme, a novel protocol suits FAPSR is also developed for building Pastry P2P system over MANETs. The major difference of our approach with existing approaches like DPSR and Ekta is its integration of automatic configuration to routing protocol. With support of our non-IP addressing scheme, fast cryptographical random number generators could be employed directly in address assignment with very low possibility of duplicate addresses.

We believe flat address is a better choice than IP address, not only because it is collision free, but also because of its great simplicity and ease in network operation. More comparison study between flat address approach and IP approach should be conducted to elaborate the advantage of flat address scheme.

Our second contribution is a brand new topology construction protocol, specifically ring construction protocol, for building popular P2P systems like Chord and Pastry over MANETs. Our approach inherited successful topology construction methods in P2P over Internet society. We believe our approach is the first successful attempt to build ring in P2P ID space over MANETs.
Our RAN (Ring Ad-hoc Network) protocol suite is an algorithm family for ring topology construction. RAN builds perfect ring in P2P ID space using only simplest multi-hop unicast and multicast communication primitives. No underlying routing protocols are needed. Upon ring generated by RAN, popular P2P networks like Chord could be immediately started and put into normal running without usual lengthy stabilization. RAN family includes a variety of algorithms for ring building. We showed the pros and cons of these algorithms, both in theory and in simulation experiments, and illustrated that the distributed exhaustive pattern is the best in terms of effectiveness and efficiency in time and messages overhead.

8.2 Suggestions of Future Works

This is a new area of study; many future research topics could be developed. Here we only mention a few as examples.

- Limit the size of DSR route cache. Because using 128-bit node ID will increase route storage size 4 times than 32-bit node ID, and route cache can share with Pastry state tables (routing tables, leaf set, neighborhood set). There should be a considerable overlap between route cache and state tables. We can optimize the storage by avoiding redundancy of overlapped routes. It is also suggested by DPSR. This way, by trade off, we can keep balance of storage usage, such that it is not too higher than IP address scheme.

- Solve the deficiency in network merge and partition. Pastry, like most DHT algorithms, is designed from wired Internet environment, which is unlikely to have partition and mergence, so it has no functionality to handle partition and mergence. DPSR and FAPSR inherit this deficiency. This is also a fundamental problem in P2P (or DHT) over MANETs, which has significant importance. Heer at el [26] gave a solution for mergence, which simply disassembles one network and let all nodes join into the other.

- Gateway configuration which connects the flat addressed pure MANETs to Internet.

- As we mentioned before as explanation for poor effectiveness of both Ring Network and RAN-Random, the very idea to keep tracing the closest node at every round of closer successor search, does not yield best successor, even using its other varieties like searching for closest predecessor. Mathematical analysis and simulation results both show the weakness of this approach is: its guideline of finding closest node limits its range of comparison. To improve this but not going to another extreme of exhaustive search, another approach could be tried. That is: keep the random itinerary but discard the closest standard. However, a follow-up question would be immediately raised, that is: without the closet criterion, what can be our end condition? Simplest answer is search depth or search time, or quality of returned node. In this
direction, we guess the biggest gold mine may be under the way. It is most
prospective follow-up research for RAN.
BIBLIOGRAPHY


Appendix 1.1 Flat Address Simulator

// FlatAddressCLI.cpp
// Use .NET's SHA1 as cryptographic Hash function
// Use CryptGenRandom() in platform SDK to generate random number

#include "stdafx.h"
#include <tchar.h>
#include <string>
#include <stdio.h>  // Needed for file access
#include <stdlib.h>
#include <fstream>
#include <time.h>
#include <math.h>
#include <windows.h>
#include <wincrypt.h>

#define MAX_ID 200000 // Define max node number in the network

#define _tprintf(_T("Fatal Error: size negative"));

using namespace System;
using namespace System::IO;
using namespace System::Security::Cryptography;

template <class T, int aSize> class Byte_string
{
public:
    T Tarray[aSize];
    int ary_size;
    Byte_string() {};
    Byte_string(T aryT[], int size)
    {
        if (size < 0)
            _tprintf(_T("Fatal Error: size negative"));
        else
        {
            ary_size = min(aSize, size);
            for (int i = 0; i < ary_size; i++) Tarray[i] = aryT[i];
        }
    }
};

// This is the entry point for this application
int _tmain(void)
{
    int nRetCode = 0;
    Int32 i = 0, j = 0, k;
    unsigned char hash_out __gc[20] = new unsigned char __gc[20];
unsigned char input_string __gc[] = new unsigned char __gc[3];
UInt32 collision;
bool same;
HCRYPTPROV hCryptProv;
BYTE pbData[3];
unsigned char node_id __gc[,,] = new unsigned char __gc[MAX_ID, 16];
SHA1CryptoServiceProvider *hash1 = new SHA1CryptoServiceProvider();
StreamWriter *out = new StreamWriter(L"result1.dat");

if (CryptAcquireContext(&hCryptProv, NULL, NULL, PROV_RSA_FULL, 0))
{
    Console::WriteLine(L"CryptAcquireContext succeeded.");
}
else
{
    _tprintf(_T("Error during CryptAcquireContext\n"));
}
Console::Write(L"Total number of nodes is ");
Console::WriteLine(MAX_ID);
Console::WriteLine();

collision = 0;
for (i = 0; i < MAX_ID; i++)
{
    // RNG CryptGenRandom() generates 32-bit node IP address
    if (! CryptGenRandom(hCryptProv, 4, pbData))
    _tprintf(_T("Error during CryptGenRandom."));
    Byte_string<BYTE, 4> bString(pbData, 4);
    for (int k = 0; k < 4; k++)
        input_string[k] = static_cast<unsigned char>(bString.Tarray[k]);
    hash1->Initialize();
    hash_out = hash1->ComputeHash(input_string);
    for (k = 0; k < 16; k++)
    { node_id[i, k] = hash_out[k];
    } //hash1->Finalize();
    for (j = 0; j < i; j++)
    { same = true;
        for (k = 0; k < 16; k++)
        { if (node_id[i, k] != node_id[j, k])
            { same = false;
                break;
            }
        } //Console::Write(L"node j: ");
        //Console::WriteLine(j);
        if (same)
        { collision++;
            out->Write(L"collision ");
        }
out->WriteLine(collision);
out->Write(L"node i: ");
out->WriteLine(i);
for (k = 0; k < 16; k++) out->Write(node_id[i,k]);
out->WriteLine();
out->Write(L"node j: ");
out->WriteLine(j);
for (k = 0; k < 16; k++) out->Write(node_id[j,k]);
out->WriteLine();
Console::Write(L"collision ");
Console::WriteLine(collision);
Console::Write(L"node i: ");
Console::WriteLine(i);
for (k = 0; k < 16; k++) Console::Write(node_id[j, k]);
Console::WriteLine();
//Console::Write(L"node i: ");
//Console::WriteLine(i);
//Console::WriteLine();
}

Console::WriteLine();
Console::Write(L"Total number of collision is ");
Console::WriteLine(collision);
Console::WriteLine();
out->WriteLine();
out->Write(L"Total number of collision is ");
out->WriteLine(collision);
out->WriteLine();
out->Flush();
out->Close();
StreamWriter *out1 = new StreamWriter("result2.dat");
out1->Write(L"Total number of collision is ");
out1->WriteLine(collision);
out1->WriteLine();
out1->Flush();
out1->Close();
return 0;
Appendix 1.2 IP Address Simulator

// IPACL1.cpp
// Use .NET's SHA1 as cryptographic Hash function
// Use CryptGenRandom() in platform SDK to generate random number

#include "stdafx.h"
#include <tchar.h>
#include <string>
#include <stdio.h>  // Needed for file access
#include <stdlib.h>
#include <fstream>
#include <time.h>
#include <math.h>
#include <windows.h>
#include <wincrypt.h>

#define ADDRESS_LEN 3
#define MAX_ID 1000 // Define max node number in the network
#define MIN_ID 200 // Define min node number in the network
#define INTERVAL 200 // Define min node number in the network
#define RUNTIMES 5

#include <mscorlib.dll>
using namespace System;
using namespace System::IO;
using namespace System::Security::Cryptography;

// This is the entry point for this application
int _tmain(void)
{
    int nRetCode = 0;
    int runtimes;
    int node_num;

    /*
     * if (File::Exists(L"result.dat")
     *    if (File::Exists(L"data\result.dat")
     *       File::Move(L"result.dat", L"data\result_bkp1.dat");
     *    else
     *       File::Move(L"result.dat", L"data\result.dat");
     *    if (File::Exists(L"brief_result.dat")
     *       if (File::Exists(L"data\brief_result.dat")
     *          File::Move(L"brief_result.dat", L"data\result1-bkp1.dat");
     *       else
     *          File::Move(L"brief_result.dat", L"data\result1.dat");
     *    */
    StreamWriter *out = new StreamWriter(L"result.dat");
    StreamWriter *out1 = new StreamWriter(L"brief_result.dat");

    out->Write(S"Address bit number is ");
    out->WriteLine(ADDRESS_LEN * 8);
    out->Write(S"The simulation repeats ");
for (node_num = MIN_ID; node_num <= MAX_ID; node_num += INTERVAL) {

    double avg_collision = 0;
    UInt32 total_collision = 0;

    for (runtimes = 1; runtimes <= RUNTIMES; runtimes++) {

        Int32 i = 0, j = 0, k;
        BYTE pbData[ADDRESS_LEN];
        unsigned char input_string __gc[] = new unsigned char __gc[ADDRESS_LEN];
        unsigned char hash_out __gc[] = new unsigned char __gc[20];
        unsigned char node_id __gc[,] = new unsigned char __gc[node_num, 16];
        UInt32 collision;
        bool same;
        HCRYPTPROV hCryptProv;
        SHA1CryptoServiceProvider *hash1 = new SHA1CryptoServiceProvider();

        out->Write(S"Total number of nodes is ");
        out->WriteLine(node_num);
        out->WriteLine();
        out1->Write(S"Total number of nodes is ");
        out1->WriteLine(node_num);
        out1->WriteLine();
        Console::Write(S"Total number of nodes is ");
        Console::WriteLine(node_num);
        Console::WriteLine();

        for (i = 0; i < ADDRESS_LEN; i++)
            for (j = 0; j < ADDRESS_LEN; j++)
                pbData[i] = (byte) input_string[i] ^ (byte) input_string[j];

        hash1->ComputeHash((byte*) &pbData[0], ADDRESS_LEN, &hash_out[0]);

        for (k = 0; k < ADDRESS_LEN; k++)
            node_id[node_num, k] = hash_out[k];

        out->Write(S"Run ");
        out->WriteLine(runtimes);
        out->WriteLine();
        out1->Write(S"Run ");
        out1->WriteLine(runtimes);
        out1->WriteLine();
    }
}

out->Write(RUNTIMES);
out->WriteLine(S" times.");
out->WriteLine();
out->WriteLine();
out1->Write(S"Address bit number is ");
out1->WriteLine(ADDRESS_LEN * 8);
out1->Write(S"The simulation repeats ");
out1->Write(RUNTIMES);
out1->WriteLine(S" times.");
out1->WriteLine();
out1->WriteLine();
Console::Write(S"Address bit number is ");
Console::WriteLine(ADDRESS_LEN * 8);
Console::Write(S"The simulation repeats ");
Console::Write(RUNTIMES);
Console::WriteLine(S" times.");
Console::WriteLine();
Console::WriteLine();

for (node_num = MIN_ID; node_num <= MAX_ID; node_num += INTERVAL) {

    double avg_collision = 0;
    UInt32 total_collision = 0;

    for (runtimes = 1; runtimes <= RUNTIMES; runtimes++) {

        Int32 i = 0, j = 0, k;
        BYTE pbData[ADDRESS_LEN];
        unsigned char input_string __gc[] = new unsigned char __gc[ADDRESS_LEN];
        unsigned char hash_out __gc[] = new unsigned char __gc[20];
        unsigned char node_id __gc[,] = new unsigned char __gc[node_num, 16];
        UInt32 collision;
        bool same;
        HCRYPTPROV hCryptProv;
        SHA1CryptoServiceProvider *hash1 = new SHA1CryptoServiceProvider();

        out->Write(S"Total number of nodes is ");
        out->WriteLine(node_num);
        out->WriteLine();
        out1->Write(S"Total number of nodes is ");
        out1->WriteLine(node_num);
        out1->WriteLine();
        Console::Write(S"Total number of nodes is ");
        Console::WriteLine(node_num);
        Console::WriteLine();

        for (i = 0; i < ADDRESS_LEN; i++)
            for (j = 0; j < ADDRESS_LEN; j++)
                pbData[i] = (byte) input_string[i] ^ (byte) input_string[j];

        hash1->ComputeHash((byte*) &pbData[0], ADDRESS_LEN, &hash_out[0]);

        for (k = 0; k < ADDRESS_LEN; k++)
            node_id[node_num, k] = hash_out[k];

        out->Write(S"Run ");
        out->WriteLine(runtimes);
        out->WriteLine();
        out1->Write(S"Run ");
        out1->WriteLine(runtimes);
        out1->WriteLine();
    }
}
Console::Write(S"Run ");
Console::WriteLine(runtimes);
Console::WriteLine();

if (! CryptAcquireContext(&hCryptProv, NULL, NULL,
PROV_RSA_FULL, 0))
_tprintf(_T("Error during CryptAcquireContext!
"));
collision = 0;
for (i = 0; i < node_num; i++)
{
  // RNG CryptGenRandom() generates 32-bit node IP address
  if (! CryptGenRandom(hCryptProv, ADDRESS_LEN, pbData))
    _tprintf(_T("Error during CryptGenRandom.
"));
  for (int k = 0; k < ADDRESS_LEN; k++)
    input_string[k] = static_cast<unsigned char> (pbData[k]);
  hash1->Initialize();
  hash_out = hash1->ComputeHash(input_string);
  for (k = 0; k < 16; k++)
  {
    node_id[i, k] = hash_out[k];
  }
  //hash1->Finalize();
  for (j = 0; j < i; j++)
  {
    same = true;
    for (k = 0; k < 16; k++)
      if (node_id[i, k] != node_id[j, k])
      {
        same = false;
        break;
      }
  }
  //Console::Write(L"node j: ");
  //Console::WriteLine(j);
  if (same)
  {
    collision++;
    /*
      out->Write(S"collision ");
      out->WriteLine(collision);
      out->Write(S"node i: ");
      out->WriteLine(i);
      for (k = 0; k < 16; k++)
        out->Write(node_id[i, k]);
      out->Write(S"node j: ");
      out->WriteLine(j);
      for (k = 0; k < 16; k++)
        out->Write(node_id[j, k]);
      out->WriteLine();
    */
  }
}

Console::Write(S"collision ");
Console::WriteLine(collision);
Console::Write(S"node i: ");
total_collision += collision;
out->WriteLine();
out->Write(S"Number of collision at this run is ");
out->WriteLine(collision);
out->Write(S"Total number of collision in all runs till now is ");
out->WriteLine(total_collision);
out->WriteLine();
out->WriteLine();
out1->WriteLine();
out1->Write(S"Number of collision at this run is ");
out1->WriteLine(collision);
out1->Write(S"Total number of collision in all runs till now is ");
out1->WriteLine(total_collision);
out1->WriteLine();
out1->WriteLine();
out->Flush();
out1->Flush();
Console::WriteLine();
Console::Write(S"Number of collision at this run is ");
Console::WriteLine(collision);
Console::Write(S"Total number of collision in all runs till now is ");
Console::WriteLine(total_collision);
Console::WriteLine();
Console::WriteLine();
out1->WriteLine();
out1->WriteLine();
out1->WriteLine();

avg_collision = (double)total_collision / RUNTIMES;
out->WriteLine();
out->Write(S"When node number is ");
out->WriteLine(node_num);
out->Write(S"Total number of collision in all runs is ");
out->WriteLine(total_collision);
out->Write(S"Average number of collision of one run is ");
out->WriteLine(avg_collision);
out->WriteLine();
out1->WriteLine();
out1->WriteLine();
out1->WriteLine();
out1->WriteLine(node_num);
out1->Write(S"Total number of collision in all runs is ");
out1->WriteLine(total_collision);
out1->Write(S"Average number of collision of one run is ");
out1->WriteLine(avg_collision);
out1->WriteLine();
out1->WriteLine();
out1->Flush();
out1->Flush();
Console::WriteLine();
Console::Write(S"When node number is ");
Console::WriteLine(node_num);
Console::Write(S"Total number of collision in all runs is ");
Console::WriteLine(total_collision);
Console::Write(S"Average number of collision of one run is ");
Console::WriteLine(avg_collision);
Console::WriteLine();
Console::WriteLine();
}
out->Close();
out1->Close();
return 0;
}
APPENDIX 2 RAN SIMULATOR SOURCE CODE

Appendix 2.1 RAN Exhaustive Pattern

/*  ----------------------------------------------------
   RAN-SC(Simplified Collective) Version 1.0
   ----------------------------------------------------

   This version directly inherits RAN 1.3.1. It adds repetition
   running facility to RAN 1.3.1.

   ----------------------------------------------------
   RAN 1.3.1 is considerably different with version 1.2.
   It uses new RAN algorithm 2. It has changed round ending
   judgment, major update is in Node::a5().

   This version switched order of sending and receiving message.
   Actually switched order of processing outgoing queue and incoming
   queue. After switching, receiving first, sending second.

   Including initialization and connected components search
   find_neighbors() function prints only node index in vector.
   */

#include "stdafx.h"
#include <tchar.h>
#include <stdio.h>
#include <math.h>
#include <windows.h>
// Following #undef directives are inserted to solve conflicting
// defines like "#def CreateDirectory CreateDirectoryA" etc.
#undef CreateDirectory
#undef SetCurrentDirectory
#undef GetCurrentDirectory
#undef GetParent

#include <wincrypt.h>
#include <iterator>
#include <vector>
#include <deque>
#include <algorithm>
#include <iostream>
#include <iomanip>
#include <stdexcept>
// #include <exception>

#using <mscorlib.dll>
#define NODEID_LENGTH_IN_BYTES 2
// #define NETWORK_SIZE 60 // Number of nodes in network
#define RANGE 15 // Radio range of all nodes
#define ORDER "Receiving First -- Reverse of Previous Versions"
// #define TIMEOUT1 48 // Ceiling(log 40) * 2 * (2 + 2)
// #define TIMEOUT2 24 // Ceiling(log 40) * 2 * (1 + 1), TIMEOUT2 < TIMEOUT1
#define GET_ALL_NEIGHBORS 1 // getAllNeighbor message type
#define ALL_NEIGHBORS 2
#define OUT_QUEUE_LENGTH 400
#define IN_QUEUE_LENGTH 400
#define MATURE_TIME 40
#define CHECK_INTERVAL 10
#define SMALL_LINE_NUM 3
#define MSG_PROC_RATE 20
#define RUN_TIME 5
#define RATE_IN_LINES 0.15
#define SHORT_LINE_LEN 3

using namespace System;
using namespace System::IO;
using namespace std;

typedef vector<unsigned int> COMPONENT;
typedef vector<int> VEC_INT;

class Message
{
public:
    int type;
    unsigned int sender;
    unsigned int receiver;
    unsigned int next; // Next hop
    vector <unsigned int> path;
    vector <unsigned int> payload;

    Message (int tp, unsigned int sndr, unsigned int rcvr,
        unsigned int nxt, vector <unsigned int> pth, vector <unsigned int> pld)
    {
        type = tp;
        sender = sndr;
        receiver = rcvr;
        next = nxt;
        path = pth;
        payload = pld;
    }

    Message () {}
};

class Node
{  
  public:
      int network_size;
    unsigned int nodeID;
    double x; // node x-coordinate, [0, 100]
    double y; // node x-coordinate, [0, 100]
    int index;
      // node index in vector<Node> (NETWORK_SIZE), like nodes or net
    unsigned int successor; // successor in Chord
    int suc_index; // index of successor, -1 means no successor
    int Timer1;
    int level; // level of current multi-hop neighbors
    int AN_received; // number of received allNeighbor messages
    int AN_in_queue; // number of allNeighbor messages in in_queue
    bool current_completed; // Indicates if current round of searching for better successor is completed
    bool total_completed;
      // total_completed indicates if all searching is completed for this node
      // Usually decided by level
    bool visited; // Used by check_ring() function
    int connect_component_num; // The number of component to which this node belongs
        // connect_component_num indicates the number of component to which
        // this node belongs. -1 means the node 's component has not been found,
        // i.e. connected_components() function has not been invoked.
    bool level_too_deep; // Search level is already too deep
    int nodes_last_level; // node number in previous level

    vector <unsigned int> neighbors;
    vector <unsigned int> L_neighbors; // current level multi-hop neighbors
    vector <unsigned int> LP_neighbors; // a backup of current level multi-hop neighbors
    vector <unsigned int> N_neighbors; // all multi-hop neighbors
    vector <COMPONENT> R_routes; // routing record of this node
    vector <COMPONENT> RL_routes; // routes for nodes in L_neighbors

    deque <Message> out_queue;
    deque <Message> in_queue;

Node (unsigned int id, int idx)  
{
    network_size = 0;
    nodeID = id;
    index = idx;
    suc_index = -1;
    connect_component_num = -1;
    level_too_deep = false;
    AN_received = 0;
AN_in_queue = 0;
current_completed = false;
level_too_deep = false;
}

Node ()
{
    network_size = 0;
suc_index = -1;
connect_component_num = -1;
level_too_deep = false;
AN_received = 0;
AN_in_queue = 0;
current_completed = false;
level_too_deep = false;
}

void a1 (vector<Node>& nodes);
void a2(int net_size);
void a3(Message msg);
void a4(Message msg);
void a5(vector<Node>& nodes);
void a6(Message msg, vector<Node>& nodes);

/* ---------- Begin of the RAN protocol ---------------- */
void Node::a1 (vector<Node>& nodes) // Initialize RAN
{
    if (neighbors.empty()) return;
    if (level_too_deep) return;

    int min_distance = 65536;
    Timer1 = 0;
    int s = neighbors.size();
    int minIndex;
    R_routes.clear();
    RL_routes.clear();
    for (int i = 0; i < s; i++)
    {
        unsigned int &a = neighbors.at(i);
        unsigned int b = a;
        unsigned int nID = nodes.at(b).nodeID;
        int distance = nID - nodeID;
        // d(u, k) = (k.ID - u.ID) mod 2^m, where 2^m is length of Chord
        if (distance < 0) distance = distance + 65536;
        if (distance < min_distance)
{    
    min_distance = distance;
    minIndex = b;
}
vector <unsigned int> route;
route.push_back(index);
route.push_back(b);
R_routes.push_back(route);
RL_routes.push_back(route);
}
suc_index = minIndex;
successor = nodes[suc_index].nodeID;
L_neighbors.assign(neighbors.begin(), neighbors.end());
N_neighbors.assign(neighbors.begin(), neighbors.end());
LP_neighbors.clear();
level = 0;
AN_received = 0;
AN_in_queue = 0;
current_completed = true;
level_too_deep = false;
nodes_last_level = s;
}

void Node::a2(int net_size)
{
    if (neighbors.empty()) return;
    if (level_too_deep) return;
    if (! current_completed) return;

    vector <COMPONENT> RL;
current_completed = false;
    Timer1 = 0;
    AN_received = 0;
    AN_in_queue = 0;
    nodes_last_level = L_neighbors.size();
    LP_neighbors.assign(L_neighbors.begin(), L_neighbors.end());
    level ++;
    if (level > net_size) level_too_deep = true;

    RL.assign(RL_routes.begin(), RL_routes.end());
    int s = RL.size();
    vector <COMPONENT>::iterator Iter;
    Iter = RL.begin();
    int i = 0;

    while (Iter != RL.end())
    {
        vector <unsigned int> route;
        route = *Iter;
        if (route.size() >= 2)
        {
            if (route.size() <= (OUT_QUEUE_LENGTH - out_queue.size()))
            {
                unsigned int &h = route.back();
                unsigned int &next = route.at(1);
                vector <unsigned int> pld;
            }
        }
    }
}
Message msg(GET_ALL_NEIGHBORS, index, h, next, route, pld);
    out_queue.push_back(msg);
    RL.erase(Iter);
}
else Iter++;
else
{
    cout << "Errore: The " << i << "th route in RL_routes has only
    one node.";
    cout << "Errore code is in " << index << "th node's a4() method." << endl;
    exit(EXIT_FAILURE);
} i++;


void Node::a3(Message m)
{
    if (neighbors.empty()) return;
    if (level_too_deep) return;

    if (out_queue.size() < OUT_QUEUE_LENGTH)
    {
        vector<unsigned int> route;
        vector<unsigned int> tmp;

        route.assign(m.path.begin(), m.path.end());
        tmp.assign(neighbors.begin(), neighbors.end());
        int s = route.size();
        if (s >= 2)
        {
            for (int i = 0; i < s; i++)
            {
                unsigned int &a = route.at(i);
                unsigned int b = a;
                vector<unsigned int>::iterator result;
                result = find(tmp.begin(), tmp.end(), b);
                if (result != tmp.end()) tmp.erase(result);
                /*
                else
                {
                    cout << "There is no " << b << " in vector neighbors in
                    ",";
                    cout << index << "th node's a3 method." << endl;
                } */
            }
            unsigned int &c = route.at(s-2);
            unsigned int d = c;
            reverse(route.begin(), route.end());
            Message msg(ALL_NEIGHBORS, index, m.sender, d, route, tmp);
            out_queue.push_back(msg);
        }
    }
void Node::a4(Message m)
{
    if (neighbors.empty()) return;
    if (level_too_deep) return;

    AN_received ++;

    // Following code gets S - N in RAN a4
    int i;
    vector <unsigned int> S; // S in a4 of RAN algorithm
    vector <unsigned int> tmp;
    S.assign(m.payload.begin(), m.payload.end());
    tmp.assign(N_neighbors.begin(), N_neighbors.end());
    int s = tmp.size();
    for (i = 0; i < s; i++) // This for loop gets set S - N_neighbors
    {
        unsigned int &a = tmp.at(i);
        unsigned int b = a;
        vector <unsigned int>::iterator result;
        result = find(S.begin(), S.end(), b);
        if (result != S.end()) S.erase(result);
    } /*
    else
    {
        cout << "There is no " << b << " in vector N_neighbors in ";
        cout << index << "th node's a4() method." << endl;
    }
    */

    // End of code to get S - N

    // Following code gets route(u, q) in RAN algorithm a4
    unsigned int q = m.sender;
    s = R_routes.size();
    for (i = 0; i < s; i++)
    {
        vector <unsigned int> r = R_routes[i];
        if (r.empty())
        {
            cout << "Errore: The " << i << "th route is empty in R_routes in ",
            cout << index << "th node's a4() method." << endl;
            exit(EXIT_FAILURE);
        }
        unsigned int &a = r.back();
        if (q == a) break;
    }
    vector <unsigned int> r = R_routes[i];
    vector <unsigned int> route_to_q;
    route_to_q.assign(r.begin(), r.end());
    // End of code to get route(u, q)
s = S.size();
for (i = 0; i < s; i++)
{
    unsigned int &a = S.at(i);
    unsigned int h = a;
    L_neighbors.push_back(h);
    N_neighbors.push_back(h);
    vector<unsigned int> route_to_h;
    route_to_h.assign(route_to_q.begin(), route_to_q.end());
    route_to_h.push_back(h);
    R_routes.push_back(route_to_h);
    RL_routes.push_back(route_to_h);
}

void Node::a5(vector<Node>& nodes)
{
    if (neighbors.empty()) return;
    if (level_too_deep) return;

    int timeout_small = 2 * level * nodes_last_level / MSG_PROC_RATE;
    int timeout_big = 4 * level * nodes_last_level / MSG_PROC_RATE;
    int t1 = Math::Round(Math::Log(nodes.size(), 2.0) * 2 * 3);
    if (timeout_big < t1) timeout_big = t1;
    if (AN_received == nodes_last_level) current_completed = true;
    else if ((Timer1 >= timeout_small) && (AN_in_queue == 0))
        current_completed = true;
    else if (Timer1 >= timeout_big) current_completed = true;
    else return;

    if (AN_in_queue > 0)
    {
        int is;
        deque<Message>::iterator Iter;
        Iter = in_queue.begin();
        while ((! in_queue.empty()) && (AN_in_queue > 0))
        {
            is = in_queue.size();
            if (Iter == in_queue.end()) break;
            Message m = *Iter;
            if (m.type == ALL_NEIGHBORS)
            {
                a4(m);
                AN_in_queue--;
                in_queue.erase(Iter);
                if (in_queue.empty()) break;
                else if (Iter == in_queue.end()) break;
            }
            else Iter++;
        }
    }

    int s2 = L_neighbors.size();
    for (int i = 0; i < s2; i++)
unsigned int &a = L_neighbors.at(i);
unsigned int b = a;
unsigned int nID = nodes[b].nodeID;
int distance = nID - nodeID;
if (distance < 0) distance += 65536;
int min_distance = nodes[suc_index].nodeID - nodeID;
if (min_distance < 0) min_distance += 65536;
if (distance < min_distance)

successor = nID;
suc_index = b;

}

void Node::a6(Message m, vector<Node>& nodes)
{
if (neighbors.empty()) return;
if (level_too_deep) return;

Message msg;
msg = m;
unsigned int a;
vector < unsigned int> route;
vector < unsigned int>::iterator result;

route.assign(msg.path.begin(), msg.path.end());
result = find( route.begin( ), route.end( ), msg.next);
if (!(result == route.end()))
{
  result++;
  a = *result;
  msg.next = a;
  out_queue.push_back(msg);
}

/* ---------- End of the RAN protocol ---------------- */

// Initialize the ad-hoc network
// nodes is a reference to a vector of all nodes created in _tmain()
// rn is run number
void init_net(vector<Node>& nodes, int rn)
{

int i, j, s;
s = nodes.size();
BYTE pbData[NODEID_LENGTH_IN_BYTES];
Random* rng = new Random();
HCUBLICPROV hCryptProv;

String *file_name = Convert::ToString(rn);
file_name = String::Concat("init_net_", file_name, ".txt");
StreamWriter *out = new StreamWriter(file_name);

if (!CryptAcquireContext(&hCryptProv, NULL, NULL, PROV_RSA_FULL, 0))
    _tprintf(_T("Error during CryptAcquireContext!\n"));

for (i = 0; i < s; i++)
    {
        int tmpID = 0;
        // RNG CryptGenRandom() generates node ID
        if (!CryptGenRandom(hCryptProv, NODEID_LENGTH_IN_BYTES, pbData))
            {
                _tprintf(_T("Error during CryptGenRandom.\n"));
                exit(EXIT_FAILURE);
            }
        for (j = NODEID_LENGTH_IN_BYTES - 1; j >= 0; j--)
            {
                tmpID = tmpID * 256 + pbData[j];
            }
        nodes[i].network_size = s;
        nodes[i].nodeID = tmpID;
        nodes[i].index = i;
        nodes[i].x = rng->NextDouble() * 100;
        nodes[i].y = rng->NextDouble() * 100;
        Console::Write(S"Node ID: ");
        Console::WriteLine(nodes[i].nodeID);
        Console::Write(S"Node Index: ");
        Console::WriteLine(nodes[i].index);
        Console::Write(S"x: ");
        Console::WriteLine(nodes[i].x);
        Console::Write(S"y: ");
        Console::WriteLine(nodes[i].y);
        out->Write(S"Node ID: ");
        out->WriteLine(nodes[i].nodeID);
        out->Write(S"Node Index: ");
        out->WriteLine(nodes[i].index);
        out->Write(S"x: ");
        out->WriteLine(nodes[i].x);
        out->Write(S"y: ");
        out->WriteLine(nodes[i].y);
    }
out->Close();

// Construct neighbor set for every node
// nodes is a reference to a vector of all nodes created in _tmain()
// rn is run number
void find_neighbors(vector<Node>& nodes, int rn)
int i, j, s;

String *file_name = Convert::ToString(rn);
file_name = String::Concat("neighbors ", file_name, ".txt");
StreamWriter *out = new StreamWriter(file_name);

out->WriteLine();
s = nodes.size();
for (i = 0; i < s; i++)
{
    cout << "Neighbors of node " << i << " : " << endl;
    out->Write("Neighbors of node ");
    out->Write(i);
    out->WriteLine(" :");
    for (j = 0; j < s; j++)
    {
        if (j != i)
        {
            double r = sqrt(pow(nodes[i].x - nodes[j].x, 2.0) +
pow(nodes[i].y - nodes[j].y, 2.0));
            if (r <= RANGE)
            {
                nodes[i].neighbors.push_back(j);
                cout << j << " ( " << nodes[j].nodeID << " ) ";
                out->Write(j);
                out->Write(" ( ");
                out->Write(nodes[j].nodeID);
                out->Write(" ) ");
            }
        }
    }
    int a = nodes[i].neighbors.size();
    cout << endl;
    cout << "Size: " << a << endl;
    out->WriteLine();
    out->Write("Size: ");
    out->WriteLine(a);
}
out->Close();
}

// Use graph BFS traversal following the order of Node::neighbors
// vector
// nodes is a reference to a vector of all nodes created in _tmain()
// rn is run number
void connected_components(vector<Node>& nodes, vector<COMPONENT>& components, int rn)
{
    int cc_num = 0; // Current connected component number
    int i = 0;
    int s1 = nodes.size();
    components.clear();

    String *file_name = Convert::ToString(rn);
    file_name = String::Concat("components ", file_name, ".txt");
    StreamWriter *out = new StreamWriter(file_name);
cout << endl << "First Components" << endl;
out->WriteLine("First Components File");
out->WriteLine();
while (i < s1)
{
    if (nodes[i].connect_component_num == -1)
    {
        vector<unsigned int> cp;
        deque < unsigned int> buf_que;
        out->Write("Component ");
        out->WriteLine(cc_num);
        cp.push_back(i);
        cout << i << " ";
        out->Write(i);
        out->Write(" ");
        nodes[i].connect_component_num = cc_num;
        buf_que.clear();
        buf_que.push_back(i);
        while (! buf_que.empty())
        {
            unsigned int &a = buf_que.at(0);
            int b = 0;
            int s2 = nodes[a].neighbors.size();
            int c = a;
            buf_que.pop_front();
            while (b < s2)
            {
                unsigned int &j = nodes[c].neighbors.at(b);
                if (nodes[j].connect_component_num == -1)
                {
                    nodes[j].connect_component_num = cc_num;
                    cp.push_back(j);
                    cout << j << " ";
                    out->Write(j);
                    out->Write(" ");
                    buf_que.push_back(j);
                }
                b++;
            }
        }
        cout << endl;
        out->WriteLine();
        components.push_back(cp);
        cc_num++;
    }
    i++;
}
out->Close();

// Print parameters for whole simulation
void print_parameters(StreamWriter *out, int net_size)
{


/*
StreamReader *in = new StreamReader("run_num.txt");
String *ss = in->ReadLine();
short run_num = Convert::ToInt16(ss);
in->Close();
run_num++;
StreamWriter *out1 = new StreamWriter("run_num.txt");
out1->WriteLine(run_num);
out1->Close();
*/

// Use graph BFS traversal following the order of Node::neighbors
// nodes is a reference to a vector of all nodes created in _tmain()
// rn is run number
void print_successor(int time, vector<Node>& nodes, int rn) {
    String *file_name = Convert::ToString(rn);
    file_name = String::Concat("successor_", file_name, ".txt");
    StreamWriter *out = new StreamWriter(file_name, true);
    cout << endl;
    cout << "-----------------------------------------" << endl;
    cout << "Time is " << time << endl << endl;
    cout << "Node(ID)      Successor(ID)" << endl << endl;
    out->WriteLine();
}
out->WriteLine("-----------------------------------------");
out->Write("Time is ");
out->WriteLine(time);
out->WriteLine();
out->WriteLine("Node(ID)      Successor(ID)");
out->WriteLine();
int s = nodes.size();
for (int i = 0; i < s; i++)
{
    cout << setw(3) << i << " (";
    cout << setw(5) << nodes[i].nodeID << " )    ");
    cout << setw(3) << nodes[i].suc_index << " (";
    cout << setw(5) << nodes[i].successor << ") " << endl;
    out->Write(i);
    out->Write(" (");
    out->Write(nodes[i].nodeID);
    out->Write(" )");
    out->Write(nodes[i].suc_index);
    out->Write(" (");
    out->Write(nodes[i].successor);
    out->WriteLine(" )");
}
out->Close();

// Use graph BFS traversal following the order of Node::neighbors
vector
// time is discrete overall time controled by simulate() function
// nodes is a reference to a vector of all nodes created in _tmain()
// line_num is the pointer which give the current round number of lines
// that are connected to one of rings, but seperated from ring topology.
// rn is run number
bool check_rings(int time, vector<Node>& nodes, int *line_num, int rn)
{
    int i, j;
    int s1, s2;
    vector <VEC_INT> rings;
    vector <VEC_INT> lines;
    int net_size = nodes.size();
    // Lines are connected directional segments, but not circular

    String *file_name = Convert::ToString(rn);
    file_name = String::Concat("rings ", file_name, ".txt");
    StreamWriter *out = new StreamWriter(file_name, true);

    cout << endl;
    cout << "-----------------------------------------" << endl;
    cout << "Time is " << time << endl;
    cout << "rings:" << endl;
    out->WriteLine();
    out->WriteLine("-----------------------------------------");
    out->Write("Time is ");
    out->WriteLine(time);

    // Use graph BFS traversal following the order of Node::neighbors
    vector
    // time is discrete overall time controled by simulate() function
    // nodes is a reference to a vector of all nodes created in _tmain()
    // line_num is the pointer which give the current round number of lines
    // that are connected to one of rings, but seperated from ring topology.
    // rn is run number
    bool check_rings(int time, vector<Node>& nodes, int *line_num, int rn)
    {
        int i, j;
        int s1, s2;
        vector <VEC_INT> rings;
        vector <VEC_INT> lines;
        int net_size = nodes.size();
        // Lines are connected directional segments, but not circular

        String *file_name = Convert::ToString(rn);
        file_name = String::Concat("rings ", file_name, ".txt");
        StreamWriter *out = new StreamWriter(file_name, true);

        cout << endl;
        cout << "-----------------------------------------" << endl;
        cout << "Time is " << time << endl;
        cout << "rings:" << endl;
        out->WriteLine();
        out->WriteLine("-----------------------------------------");
        out->Write("Time is ");
        out->WriteLine(time);
for (i = 0; i < net_size; i++) nodes[i].visited = false;

for (i = 0; i < net_size; i++)
{
    Node n1 = nodes[i];
    Node n2;
    if (! nodes[i].visited)
    {
        vector < int> ring1;
        if (nodes[i].suc_index == -1)
        {
            ring1.push_back(i);
            rings.push_back(ring1);
            // An isolated node is regarded as ring
            // just by sense of connected component
            continue;
        }
        j = i;
        while ((! nodes[j].visited) && (nodes[j].suc_index > -1))
        {
            n2 = nodes[j];
            nodes[j].visited = true;
            ring1.push_back(j);
            j = nodes[j].suc_index;
        }
        n2 = nodes[j];
        if (nodes[j].visited)
        {
            vector < int>::iterator result;
            result = find( ring1.begin( ), ring1.end( ), j);
            if (!(result == ring1.end( )))
            {
                vector < int> ring2;
                ring2.assign(result, ring1.end());
                rings.push_back(ring2);
                if (!(result == ring1.begin( )))
                {
                    ring2.assign(ring1.begin( ), result);
                    lines.push_back(ring2);
                }
            }
            else lines.push_back(ring1);
        }
        if (nodes[j].suc_index == -1)
        {
            _tprintf(_T("Semantics Error in check_ring() function. "));
            _tprintf(_T("A node without successor must have no neighbor. "));
            _tprintf(_T("So it can not be any other node's successor."));
            exit(EXIT_FAILURE);
        }
    }
}

cout << "-----------------------------" << endl;
cout << "Rings" << endl << endl;
out->WriteLine("-----------------------------");
out->WriteLine("Rings");
out->WriteLine();
s1 = rings.size();
for (i = 0; i < s1; i++)
{
    vector<int> ring;
    VEC_INT &r = rings.at(i);
    ring.assign(r.begin(), r.end());
    cout << "Ring" << i << ":" << endl;
    out->Write("Ring");
    out->Write(i);
    out->WriteLine(":");
    s2 = ring.size();
    for (j = 0; j < s2; j++)
    {
        int &a = ring.at(j);
        int b = a;
        cout << " " << b << " (" << nodes[b].nodeID << ")";
        out->Write(" ");
        out->Write(b);
        out->Write(" (");
        out->Write(nodes[b].nodeID);
        out->Write(")");
    }
    int &a = ring.at(0);
    int b = a;
    cout << " " << b << " (" << nodes[b].nodeID << ")";
    out->Write(" ");
    out->Write(b);
    out->Write(" (");
    out->Write(nodes[b].nodeID);
    out->Write(")");
    cout << endl << endl;
    out->Flush();
}

cout << "-----------------------------" << endl << endl;
out->WriteLine("-----------------------------");
out->WriteLine();
cout << "Lines" << endl << endl;
out->WriteLine("Lines");
out->WriteLine();
out->Flush();

s1 = lines.size();
*line_num = s1;
int nodes_in_lines = 0;
bool all_lines_short = true;
for (i = 0; i < s1; i++)
{
    vector<int> line;
    VEC_INT &ln = lines.at(i);
line.assign(ln.begin(), ln.end());
s2 = line.size();
node_in_lines += s2;
if (s2 > SHORT_LINE_LEN) all_lines_short = false;
cout << "Line" << i << ":" << endl;
out->Write("Line");
out->Write(i);
out->WriteLine(":");
for (j = 0; j < s2; j++)
{
    int &a = line.at(j);
    int b = a;
cout << " " << b << " (" << nodes[b].nodeID << ")";
    out->Write(" ");
    out->Write(b);
    out->Write(" ("");
    out->Write(nodes[b].nodeID);
    out->Write(")");
}
cout << endl << endl;
out->WriteLine();
out->WriteLine();
out->Flush();
}

cout << "-----------------------------" << endl << endl;
out->WriteLine("-----------------------------");
out->WriteLine();
out->Close();
if (all_lines_short && (node_in_lines <= nodes.size() * RATE_IN_LINES))
    return true;
else return false;
}

// Primary control of simulation
// nodes is a reference to a vector of all nodes created in _tmain()
// out is the pointer to output.txt file
// rn is run number
void simulate(vector<Node>& nodes, StreamWriter *out, int rn)
{
    int i, j, net_size, T, p, interval, line_num;
    int ms = 0; // number of sent messages
    int mr = 0; // number of received messages, <= ms
    net_size = nodes.size();

    for (i = 0; i < net_size; i++)
    {
        Node &n1 = nodes.at(i);
        n1.a1(nodes);
    }

    T = 0;
p = 0;
i = 0;
line_num = 0;
interval = CHECK_INTERVAL;
while (true)
{
    if (i >= MATURE_TIME)
    {
        if (T % interval == 0)
        {
            if (p % 20 == 0) print_successor(i, nodes, rn);
            p++;
            if (check_rings(i, nodes, &line_num, rn)) break;
            else if (line_num <= SMALL_LINE_NUM) interval = 1;
        }
        T++;
    }
    /* Begin checking if all nodes are too deep */
    bool all_nodes_level_too_deep = true;
    for (j = 0; j < net_size; j++)
    {
        if (! nodes[j].level_too_deep) all_nodes_level_too_deep = false;
    }
    if (all_nodes_level_too_deep) {
        cout << "Error in simaulate() function: successor search too deep, ";
        cout << " level already greater than network size. ";
        cout << "Simulation is stopped." << endl;
        out->WriteLine();
        out->Write("Error in simaulate: successor search too deep, ");
        out->WriteLine(" level already greater than network size");
        out->WriteLine("Simulation is stopped.");
        out->WriteLine();
        out->Flush();
        break;
    }
    /* End checking if all nodes are too deep */

    /* Begin scanning every node repeatedly */
    for (j = 0; j < net_size; j++)
    {
        Node &n2 = nodes.at(j);
        n2.Timer1 ++;
        if (n2.neighbors.empty()) continue;
        if (n2.level_too_deep) continue;
        n2.a2(net_size);

        /* --------- Begin Processing Incoming Messages ----------- */
        int k = 0;
        while (!(n2.in_queue.empty()) && (k < MSG_PROC_RATE))
        {
            Message &m = n2.in_queue.front();
            Message msg = m;
            if (msg.next != n2.index)
            {
                String *file_name = Convert::ToString(rn);
            }
file_name = String::Concat("error_", file_name, ".txt");
StreamWriter *out1 = new StreamWriter(file_name);
cout << "Error in simulate(): message has been ";
cout << "forwarded to wrong node. node ID: ";
cout << n2.index << endl;
out1->Write("Error in simulate(): message has been ");
out1->Write("forwarded to wrong node. node ID: ");
out1->WriteLine(n2.index);
out1->WriteLine();
out1->Close();
exit(EXIT_FAILURE);
}
if (msg.receiver == n2.index)
{
  if (msg.type == GET_ALL_NEIGHBORS) n2.a3(msg);
  else if (msg.type == ALL_NEIGHBORS)
  {
    // Next line is primary place to decrement a node's
    // Another place is in Node::a5()
    n2.AN_in_queue --;
    n2.a4(msg);
  }
  else
  {
    String *file_name = Convert::ToString(rn);
    file_name = String::Concat("error_", file_name, ".txt");

    StreamWriter *out1 = new StreamWriter(file_name);
    cout << "Error in simulate: front message of ";
    cout << " in_queue has wrong type. node " << n2.index;
    cout << endl;
    out1->Write("Error in simulate: front message of ");
    out1->Write(" in_queue has wrong type. node ");
    out1->Write(n2.index);
    out1->WriteLine();
    out1->Close();
    exit(EXIT_FAILURE);
  }
}
else n2.a6(msg, nodes);
 n2.in_queue.pop_front();
 mr++;
 k++;
}
/* ---------- End Processing Incoming Messages ---------- */

/* ---------- Begin Processing Outgoing Messages ---------- */
 k = 0;
while (!(n2.out_queue.empty()) && (k < MSG_PROC_RATE))
{
  Message &m = n2.out_queue.front();
  Message msg = m;
  unsigned int nxt = msg.next;
  Node &n3 = nodes.at(nxt);
  if (n3.in_queue.size() < IN_QUEUE_LENGTH)
    n3.in_queue.push_back(msg);
// This is the entry point for this application
int _tmain(void)
{
    int i;
    int k = RUN_TIME;
    int n; // NETWORK_SIZE
    DirectoryInfo *pinfo;

    for (n = 20; n <= 160; n += 20)
    {
        String *path = Convert::ToString(n);
        Directory::CreateDirectory(path);
        Directory::SetCurrentDirectory(path);
        StreamWriter *out = new StreamWriter("output.txt");
        print_parameters(out, n);
        for (i = 0; i < k; i++)
        {
            vector<Node> net(n);
            vector<COMPONENT> comps;

            init_net(net, i);
            find_neighbors(net, i);
            connected_components(net, comps, i);
            out->WriteLine();
            out->Write("simulation_time ");
            out->Write("sent_messages# ");
            out->WriteLine("received messagess#");
            out->Flush();
        }
    }

    // This is the entry point for this application
    int _tmain(void)
    {
        int i;
        int k = RUN_TIME;
        int n; // NETWORK_SIZE
        DirectoryInfo *pinfo;

        for (n = 20; n <= 160; n += 20)
        {
            String *path = Convert::ToString(n);
            Directory::CreateDirectory(path);
            Directory::SetCurrentDirectory(path);
            StreamWriter *out = new StreamWriter("output.txt");
            print_parameters(out, n);
            for (i = 0; i < k; i++)
            {
                vector<Node> net(n);
                vector<COMPONENT> comps;

                init_net(net, i);
                find_neighbors(net, i);
                connected_components(net, comps, i);
                out->WriteLine();
                out->Write("simulation_time ");
                out->Write("sent_messages# ");
                out->WriteLine("received messagess#");
                out->Flush();
        }
    }
simulate(net, out, i);
}
out->Close();
path = Directory::GetCurrentDirectory();
pinfo = Directory::GetParent(path);
Directory::SetCurrentDirectory(pinfo->FullName);
} return 0;
Appendix 2.2 RAN Random Pattern

/* ----------------------------------------------------
   RAN-Random (RAN Pattern Random) version 3.1
 ----------------------------------------------------
RAN-Random version 3.1 inherits RAN-Random 3.0. This
version also implements RAN Random pattern algorithm
version 3.

 ----------------------------------------------------
RAN-Random version 3.0 inherits RAN-SC 1.x and implements
RAN Random pattern algorithm version 3.

 ----------------------------------------------------
RAN-SC 1.1 inherits RAN-SC 1.0. RAN-SC 1.1 implements
RAN algorithm version 4.

 ----------------------------------------------------
RAN-SC 1.0 inherits RAN 1.3.1. It adds repetitive
running facility to RAN 1.3.1.

 ----------------------------------------------------
RAN 1.3.1 is considerably different with version 1.2.
It uses new RAN algorithm 2. It has changed round ending
judgment, major update is in Node::a5().

   This version switched order of sending and receiving message.
   Actually switched order of processing outgoing queue and incoming
   queue. After switching, receiving first, sending second.

   Including initialization and connected components search
   find_neighbors() function prints only node index in vector.
 */

#include "stdafx.h"
#include <tchar.h>
#include <stdio.h>
#include <math.h>
#include <windows.h>
#include <iterator>
#include <vector>
#include <deque>

// Following #undef directives are inserted to solve conflicting
// defines like "#def CreateDirectory CreateDirectoryA" etc.
#undef CreateDirectory
#undef SetCurrentDirectory
#undef GetCurrentDirectory
#undef GetParent

#include <wincrypt.h>
#include <iterator>
#include <vector>
#include <deque>
```cpp
#include <algorithm>
#include <iostream>
#include <iomanip>
#include <stdexcept>
#include <exception>
#using <mscorlib.dll>

#define NODEID_LENGTH_IN_BYTES 2
#define RANGE 15 // Radio range of all nodes
#define ORDER "Receiving First -- Reverse of Previous Versions"
#define CLOSER_NODE 1 // Closer Node Search message type
#define SUCCESSOR_CANDIDATE 2 // successorCandidate message type
#define OUT_QUEUE_LENGTH 400
#define IN_QUEUE_LENGTH 400
#define MATURE_TIME 40
#define CHECK_INTERVAL 20
#define SMALL_LINE_NUM 1
#define MSG_PROC_RATE 10
#define RUN_TIME 1
#define RATIO_IN_LINES 0.0
#define SHORT_LINE_LEN 1
#define COMPO_QUE_EMPTY_TIME_LIMIT 40

using namespace System;
using namespace System::IO;
using namespace std;

typedef vector<unsigned int> COMPONENT;
typedef vector<int> VEC_INT;

class Message
{
    public:
        int type; // sender, receiver, next, originator, successor are node index
        unsigned int sender; // Direct sender
        unsigned int next; // Next hop node index
        unsigned int originator; // Message originator (first creator) node index
        // For CLOSER NODE message, it is the search originator.
        // For SUCCESSOR_CANDIDATE message, it is the node which
        // is the candidate.
        unsigned int receiver; // Final receiver node index
        vector <unsigned int> path; // route from originator to receiver
        int serial_num; // originator generated serial number, used to identify
        // same search message
        int depth;
        unsigned int succ_index; // succ_index in SUCCESSOR_CANDIDATE message. It is
        // a node index, not node ID.
        int msgID;

        Message (int tp, unsigned int sndr, unsigned int nxt)
        {
```
type = tp;
sender = sndr;
next = nxt;
Random *rnd = new Random();
msgID = rnd->Next();
}
Message (int tp, unsigned int sndr, unsigned int nxt, unsigned int org,
unsigned int rcvr, vector <unsigned int> pth)
{
    type = tp;
    sender = sndr;
    next = nxt;
    originator = org;
    receiver = rcvr;
    path.assign(pth.begin(), pth.end());
    Random *rnd = new Random();
    msgID = rnd->Next();
}
Message () {}
void display ()
{
    StreamWriter *out = new StreamWriter("message.txt", true);
    cout << endl;
    cout << "Message " << msgID << endl;
    cout << "----------" << endl;
    cout << "type: " << type << endl;
    cout << "sender: " << sender << endl;
    cout << "next: " << next << endl;
    cout << "originator: " << originator << endl;
    cout << "receiver: " << receiver << endl;
    cout << "path.size: " << path.size() << endl;
    out->WriteLine();
    out->Write("Message ");
    out->WriteLine(msgID);
    out->WriteLine("----------");
    out->Write("type: ");
    out->WriteLine(type);
    out->WriteLine("sender: ");
    out->WriteLine(sender);
    out->WriteLine("next: ");
    out->WriteLine(next);
    out->WriteLine("originator: ");
    out->WriteLine(originator);
    out->Write("receiver: ");
    out->WriteLine(receiver);
    out->Write("path.size: ");
    out->WriteLine(path.size());
    cout << "path: ";
    out->Write("path: ");
    int sl = path.size();
    for (int i = 0; i < sl; i++)
    {
const unsigned int &a = path.at(i);
cout << a << " ";
out->Write(a);
}
cout << endl;
out->WriteLine();
cout << "serial_num: " << serial_num << endl;
cout << "depth: " << depth << endl;
cout << "succ_index: " << succ_index << endl;
out->Write("serial_num: ");
out->WriteLine(serial_num);
out->Write("depth: ");
out->WriteLine(depth);
out->Write("succ_index: ");
out->WriteLine(succ_index);
out->WriteLine();
out->Close();

// nodes used as elements in the BFS connected component queue
component_queue
class BFS_Node
{
    public:
    unsigned int nodeIdx; // index of node
    bool traversed;
    vector<unsigned int> in_route; // route from current node to this element node

    BFS_Node(unsigned int Idx)
    {
        nodeIdx = Idx;
        traversed = true;
        in_route.clear();
    }

    BFS_Node()
    {
        traversed = true;
        in_route.clear();
    }
};

class Node
{
    public:
    int network_size;
    unsigned int nodeID;
    double x; // node x-coordinate, [0, 100]
    double y; // node x-coordinate, [0, 100]
    int index;
// node index in vector<Node> (NETWORK_SIZE), like nodes or net
unsigned int successor; // successor ID in Chord
int suc_index; // index of successor, -1 means no successor
bool visited; // Used by check_ring() function
int connect_component_num;
    // connect_component_num indicates the number of component to
    // which this node belongs. -1 means the node 's component has
    // not been found, i.e. connected_components() function has
    // not been invoked.
bool new_round;
    // new_round indicates this round should stop
    // and a new round should be started
int compo_queue_empty_timer;
    // timer for counting how long the component_queue has become
    // -1 when component_queue not empty
vector <unsigned int> neighbors;
deque <BFS_Node> component_queue;
    // The BFS connected component queue, used as the
    // source set to feed the closerNodeSearch. It can
    // remember if a node has been traversed
deque <BFS_Node> dumped_component;
    // dumped_component contains those nodes popped out
    // from component_queue
vector <unsigned int> B;
    // B stores node indexes of successor candidates, not node IDs.
    // Candidates are found in Node::a3.
vector <unsigned int> W;
    // W stores node indexes of successor candidates, not node IDs.
    // Candidates are returned in SUCCESSOR_CANDIDATE messages.
vector <COMPONENT> R_routes; // routing record of this node
deque <Message> out_queue;
deque <Message> in_queue;
vector <Message> received_search_msgs;
    // received_search_msgs is this node's set of received
    // different CLOSER_NODE

Node (unsigned int id, int idx)
{
    network_size = 0;
    nodeID = id;
    index = idx;
    suc_index = -1;
    connect_component_num = -1;
    //new_round = false;
}

Node ()
{
    network_size = 0;
    suc_index = -1;
    connect_component_num = -1;
    //new_round = false;
}
```cpp
void a1(vector<Node>& nodes);
void a2(vector<Node> &nodes);
void a3(Message msg, vector<Node> &nodes);
void a4(Message msg, vector<Node> &nodes);
};

/* ---------- Begin of the RAN protocol ---------------- */
void Node::a1(vector<Node>& nodes) // Initialize RAN
{
    if (neighbors.empty())
    {
        suc_index = -1;
        new_round = false;
        return;
    }

    // Following code for finding node's successor //
    int min_distance = 65536;
    int ns = neighbors.size();
    int min_index = -1;
    for (int i = 0; i < ns; i++)
    {
        const unsigned int &a = neighbors.at(i);
        unsigned int nID = nodes.at(a).nodeID;
        int distance = nID - nodeID;
        // d(u, k) = (k.ID - u.ID) mod 2^m, where 2^m is length of Chord ring
        if (distance < 0) distance = distance + 65536;
        if (distance < min_distance)
        {
            min_distance = distance;
            min_index = a;
        }
    }
    if (min_index == -1)
    {
        StreamWriter *out = new StreamWriter("error.txt", true);
        out->WriteLine("Error: min_index == -1 while there must be a successor");
        out->WriteLine("Code in Node::a3");
        out->Close();
        exit(EXIT_FAILURE);
    }
    suc_index = min_index;
    successor = nodes.at(suc_index).nodeID;
    // End of code for finding node's successor //
    in_queue.clear();
    out_queue.clear();
```
component_queue.clear();
new_round = true;

// --------------- Begin ------------------ //
// Following code for building node's BFS //
// component queue       //
for (int i = 0; i < ns; i++)
{
    const unsigned int &a = neighbors.at(i);
    BFS_Node compo_node(a);
    compo_node.in_route.push_back(index);
    compo_node.in_route.push_back(a);
    component_queue.push_back(compo_node);
}
compo_queue_empty_timer = -1;
// ---------------- End ------------------ //

// Source that generates CLOSER_NODE search message
void Node::a2(vector<Node> &nodes)
{
    if (component_queue.empty())
    {
        if (compo_queue_empty_timer == -1) compo_queue_empty_timer = 0;
        else compo_queue_empty_timer ++;
        return ;
    }
    if (! new_round) return;
    // Initialize this round
    new_round = false;
    B.clear();
    W.clear();

    // Take front node from component_queue to cn1
    const BFS_Node &cn1 = component_queue.front();
    unsigned int rcv = cn1.nodeIdx;
    dumped_component.push_back(cn1);
    component_queue.pop_front();

    // Send cn1 a CLOSER_NODE search message
    Message m(CLOSER_NODE, index, rcv, index, rcv, cn1.in_route);
    Random *rnd = new Random();
    m.serial_num = rnd->Next();
    m.depth = 0;
    m.display();
    out_queue.push_back(m);

    // Update component_queue
    // Begin
    const Node &n1 = nodes.at(rcv);
    int s1 = n1.neighbors.size();
    for (int i = 0; i < s1; i++)
    {

const unsigned int h = n1.neighbors.at(i);

// Check if h in component_queue
bool h_found_in_compo = false;
int s2 = component_queue.size();
for (int j = 0; j < s2; j++) {
    const BFS_Node &bn1 = component_queue.at(j);
    if (h == bn1.nodeIdx) {
        h_found_in_compo = true;
        break;
    }
}
if (h_found_in_compo) continue;

// Check if h in dumped_component
bool h_found_in_dumped = false;
s2 = dumped_component.size();
for (int j = 0; j < s2; j++) {
    const BFS_Node &bn1 = dumped_component.at(j);
    if (h == bn1.nodeIdx) {
        h_found_in_dumped = true;
        break;
    }
}
if (h_found_in_dumped) continue;

// Add h to component_queue
BFS_Node cn2(h);
cn2.in_route.assign(cn1.in_route.begin(), cn1.in_route.end());
cn2.in_route.push_back(h);
component_queue.push_back(cn2);
}
// End

void Node::a3(Message msg, vector<Node> &nodes) {
    StreamWriter *out = new StreamWriter("message.txt", true);
    cout << endl << "Node " << index << " receives ";
    cout << "a CLOSER_NODE message in a3." << endl;
    out->WriteLine();
    out->Write("Node ");
    out->Write(index);
    out->WriteLine(" receives a CLOSER_NODE message in a3.");
    out->WriteLine();
    out->Close();
    msg.display();
    bool send_candidate = false;
}
// Check if this message is redundant, i.e. if this node
// already received a node with same serial number and originator.
int s1 = received_search_msgs.size();
for (int i = 0; i < s1; i++)
{
    const Message &m = received_search_msgs.at(i);
    if ((m.originator == msg.originator) && (m.serial_num ==
msg.serial_num))
        return;
}
received_search_msgs.push_back(msg);

const Node &n1 = nodes.at(msg.originator);
unsigned int org_ID = n1.nodeID;
int min = 65536;
in

// If the search message has gone too deep
if (msg.depth >= network_size)
{
    send_candidate = true;
    new_round = true;
}
else if (! neighbors.empty())
{
    // Following code finds current node's closest
    // neighbor to the msg.originator.
    // ------------ Begin -----------------  //
s1 = neighbors.size();
    for (i = 0; i < s1; i++)
    {
        const unsigned int &a = neighbors.at(i);
        if (a == msg.sender) continue;
        const Node &n2 = nodes.at(a);
        unsigned  int nID = n2.nodeID;
        int distance1 = nID - org_ID;
        if (distance1 < 0) distance1 += 65536;
        if (distance1 < min)
        {
            min = distance1;
            min_index = a;
        }
    }
    // --------------- End ----------------  //
    int distance2 = nodeID - org_ID;
    if (distance2 < 0) distance2 += 65536;
    if (distance2 < min) send_candidate = true;
}
else send_candidate = true;

if (send_candidate)
{
    // Send out SUCCESSOR_CANDIDATE message
    vector <unsigned int> tmp;
}
B.push_back(org_ID); // msg.originator is x in algorithm
tmp.assign(msg.path.begin(), msg.path.end());
reverse(tmp.begin(), tmp.end());
// Now tmp is the reverse route to msg.originator
const unsigned int &nxt_node = tmp.at(1);
unsigned int next = nxt_node;
Message m(SUCCESSOR_CANDIDATE, index, next, index, msg.originator,
tmp);
m.succ_index = suc_index;
out_queue.push_back(m);
}
else
{
    // Forward CLOSER_NODE search message
    if (min_index == -1)
    {
        StreamWriter *out = new StreamWriter("error.txt", true);
        out->Write("Error: min_index == -1 while there must be a ");
        out->WriteLine("closer neighbor than current node.");
        out->WriteLine("Code in Node::a3");
        out->Close();
        exit(EXIT_FAILURE);
    }
    vector < unsigned int> tmp;
tmp.assign(msg.path.begin(), msg.path.end());
tmp.push_back(min_index);
Message m(CLOSER_NODE, index, min_index, msg.originator,
min_index, tmp);
m.serial_num = msg.serial_num;
m.depth = msg.depth + 1;
out_queue.push_back(m);
}
}

// Process received SUCCESSOR_CANDIDATE message
void Node::a4(Message msg, vector<Node> &nodes)
{
    StreamWriter *out = new StreamWriter("message.txt", true);
    cout << endl << "Node " << index << " receives ";
    cout << "a SUCCESSOR_CANDIDATE message in a3." << endl;
    out->WriteLine();
    out->Write("Node ");
    out->Write(index);
    out->WriteLine(" receives a SUCCESSOR_CANDIDATE message in a4.");
    out->WriteLine();
    out->Close();
    msg.display();
    if (index == msg.receiver) // This message is for me
    {
        new_round = true;
        /* Following code update successor */
        W.push_back(msg.succ_index);
        vector <unsigned int> tmp;
tmp.insert(tmp.end(), W.begin(), W.end());
int min = successor - nodeID;
if (min < 0) min += 65536;
int minIdx = suc_index;
for (int i = 0; i < s1; i++)
{
    const unsigned int &a = tmp.at(i);

    unsigned int nID = nodes.at(a).nodeID;
    int distancel = nID - nodeID;
    if (distancel < 0) distancel += 65536;
    if (distancel < min)
    {
        min = distancel;
        minIdx = a;
    }
}
suc_index = minIdx;
successor = nodes.at(suc_index).nodeID;
/* End of updating successor */
}
else // This message is not for me. I forward it.
{
    if (msg.next != index)
    {
        StreamWriter *out = new StreamWriter("error.txt", true);
        out->WriteLine("Error: received message m. m.next != current index");
        out->WriteLine("Code in Node::a3");
        out->Close();
        exit(EXIT_FAILURE);
    }
    vector< unsigned int > route;
    route.assign(msg.path.begin(), msg.path.end());
    unsigned int a;
    vector< unsigned int >::iterator result;
    result = find( route.begin( ), route.end( ), msg.next);
    if (!(result == route.end()))
    {
        Message m(msg.type, msg.sender, msg.next, msg.originator, msg.receiver, route);
        result++;
        a = *(result);
        m.next = a;
        m.serial_num = msg.serial_num;
        m.succ_index = msg.succ_index;
        m.display();
        out_queue.push_back(m);
    }
}
/* ----------- End of the RAN protocol --------------- */
// Initialize the ad-hoc network
// nodes is a reference to a vector of all nodes created in _tmain()
// rn is run number
void init_net(vector<Node>& nodes, int rn)
{
    int i, j, s;
    s = nodes.size();
    BYTE pbData[NODEID_LENGTH_IN_BYTES];
    Random* rng = new Random();
    HCRYPTPROV hCryptProv;

    String *file_name = Convert::ToString(rn);
    file_name = String::Concat("init_net_", file_name, ".txt");
    StreamWriter *out = new StreamWriter(file_name);

    if (! CryptAcquireContext(&hCryptProv, NULL, NULL, PROV_RSA_FULL, 0))
        _tprintf(_T("Error during CryptAcquireContext!
"));

    for (i = 0; i < s; i++)
    {
        int tmpID = 0;

        // RNG CryptGenRandom() generates node ID
        if (! CryptGenRandom(hCryptProv, NODEID_LENGTH_IN_BYTES, pbData))
        {
            _tprintf(_T("Error during CryptGenRandom.
"));
            StreamWriter *out = new StreamWriter("error.txt", true);
            out->WriteLine("Error during CryptGenRandom.");
            out->Close();
            exit(EXIT_FAILURE);
        }

        for (j = NODEID_LENGTH_IN_BYTES - 1; j >=0; j--)
        {
            tmpID = tmpID * 256 + pbData[j];
        }
        nodes[i].network_size = s;
        nodes[i].nodeID = tmpID;
        nodes[i].index = i;
        nodes[i].x = rng->NextDouble() * 100;
        nodes[i].y = rng->NextDouble() * 100;
        Console::Write(S"Node ID: ");
        Console::WriteLine(nodes[i].nodeID);
        Console::Write(S"Node Index: ");
        Console::WriteLine(nodes[i].index);
        Console::Write(S"x: ");
        Console::WriteLine(nodes[i].x);
        Console::Write(S"y: ");
        Console::WriteLine(nodes[i].y);
        Console::WriteLine();
        out->Write(S"Node ID: ");
        out->WriteLine(nodes[i].nodeID);
    }
}
// Construct neighbor set for every node
// nodes is a reference to a vector of all nodes created in _tmain()
// rn is run number
void find_neighbors(vector<Node>& nodes, int rn)
{
    int i, j, s;

    String *file_name = Convert::ToString(rn);
    file_name = String::Concat("neighbors ", file_name, ".txt");
    StreamWriter *out = new StreamWriter(file_name);

    out->WriteLine();
    s = nodes.size();
    for (i = 0; i < s; i++)
    {
        cout << "Neighbors of node " « i « ": " « endl;
        out->Write("Neighbors of node ");
        out->Write(i);
        out->WriteLine(":");
        for (j = 0; j < s; j++)
        {
            if (j != i)
            {
                double r = sqrt(pow(nodes[i].x - nodes[j].x, 2.0) +
                        pow(nodes[i].y - nodes[j].y, 2.0));
                if (r <= RANGE)
                {
                    nodes[i].neighbors.push_back(j);
                    cout « j « " ( " « nodes[j].nodeID « " ) " « ;
                    out->Write(j);
                    out->Write(" ");
                    out->Write(nodes[j].nodeID);
                    out->Write(" ");
                }
            }
        }
        int a = nodes[i].neighbors.size();
        cout « endl;
        cout « "Size: " « a « endl;
        out->WriteLine();
        out->Write("Size: ");
        out->WriteLine(a);
    }
    out->Close();
}
// Use graph BFS traversal following the order of Node::neighbors vector
// nodes is a reference to a vector of all nodes created in _tmain()
// rn is run number

void connected_components(vector<Node>& nodes, vector<COMPONENT>& components,
                           int rn, int *comp_num)
{
    int cc_num = 0; // Current connected component number
    int i = 0;
    int sl = nodes.size();
    components.clear();

    String *file_name = Convert::ToString(rn);
    file_name = String::Concat("components_", file_name, ".txt");
    StreamWriter *out = new StreamWriter(file_name);

    cout << endl << "First Components" << endl;
    out->WriteLine("First Components File");
    out->WriteLine();
    while (i < sl)
    {
        if (nodes[i].connect_component_num == -1)
        {
            vector< unsigned int > cp;
            deque < unsigned int > buf_que;
            out->Write("Component ");
            out->WriteLine(cc_num);
            cp.push_back(i);
            cout << i << "   ";
            out->Write(i);
            out->Write("   ");
            nodes[i].connect_component_num = cc_num;
            buf_que.clear();
            buf_que.push_back(i);
            while (! buf_que.empty())
            {
                unsigned int &a = buf_que.at(0);
                int b = 0;
                int s2 = nodes[a].neighbors.size();
                int c = a;
                buf_que.pop_front();
                while (b < s2)
                {
                    unsigned int& j = nodes[c].neighbors.at(b);
                    if (nodes[j].connect_component_num == -1)
                    {
                        nodes[j].connect_component_num = cc_num;
                        cp.push_back(j);
                        cout << j << "   ";
                        out->Write(j);
                        out->Write("   ");
                        buf_que.push_back(j);
                    }
                    b++;
                }
            }
        }
        i++;
    }
}
cout << endl;
out->WriteLine();
components.push_back(cp);
cc_num++;
}
i++;
}
out->Close();
*comp_num = components.size();
}

// Print parameters for whole simulation
void print_parameters(StreamWriter *out, int net_size)
{
  //out->Write("RUN");
  //out->WriteLine(run_num);
  out->WriteLine();
  out->WriteLine("RAN-Random Version 3.0");
  out->Write("NODEID_LENGTH_IN_BYTES: ");
  out->WriteLine(NODEID_LENGTH_IN_BYTES);
  out->WriteLine("NETWORK_SIZE: ");
  out->WriteLine(net_size);
  out->Write("RANGE: ");
  out->WriteLine(RANGE);
  out->Write("ORDER: ");
  out->WriteLine("Receiving First -- Reverse of Previous Versions");
  out->Write("OUT_QUEUE_LENGTH: ");
  out->WriteLine(OUT_QUEUE_LENGTH);
  out->Write("IN_QUEUE_LENGTH: ");
  out->WriteLine(IN_QUEUE_LENGTH);
  out->Write("MATURE_TIME: ");
  out->WriteLine(MATURE_TIME);
  out->Write("CHECK_INTERVAL: ");
  out->WriteLine(CHECK_INTERVAL);
  out->Write("SMALL_LINE_NUM: ");
  out->WriteLine(SMALL_LINE_NUM);
  out->Write("MSG_PROC_RATE: ");
  out->WriteLine(MSG_PROC_RATE);
  out->Write("RATIO_IN_LINES: ");
  out->WriteLine(RATIO_IN_LINES);
  out->Write("COMPO_QUE_EMPTY_TIME_LIMIT: ");
  out->WriteLine(COMPO_QUE_EMPTY_TIME_LIMIT);
  out->Flush();
}

// Use graph BFS traversal following the order of Node::neighbors
// nodes is a reference to a vector of all nodes created in _tmain()
void print_successor(int time, vector<Node>& nodes, int rn) {
    String *file_name = Convert::ToString(rn);
    file_name = String::Concat("successor ", file_name, ".txt");
    StreamWriter *out = new StreamWriter(file_name, true);
    
    cout << endl;
    cout << "-----------------------------------------" << endl;
    cout << "Time is " << time << endl << endl;
    cout << "Node(ID)      Successor(ID)" << endl << endl;
    out->WriteLine();
    out->WriteLine("-----------------------------------------");
    out->Write("Time is ");
    out->WriteLine(time);
    out->WriteLine();
    out->WriteLine("Node(ID)      Successor(ID)");
    out->WriteLine();
    int s = nodes.size();
    for (int i = 0; i < s; i++) {
        cout << setw(3) << i << " (";
        cout << setw(5) << nodes[i].nodeID << ")    ";
        cout << setw(3) << nodes[i].suc_index << " (";
        cout << setw(5) << nodes[i].successor << ")" << endl;
        out->Write(i);
        out->Write(" (";
        out->Write(nodes[i].nodeID);
        out->Write(" ");
        out->Write(nodes[i].suc_index);
        out->Write(" ");
        out->Write(nodes[i].successor);
        out->WriteLine(")");
    }
    out->Close();
}

// Use graph BFS traversal following the order of Node::neighbors vector
// time is discrete overall time controled by simulate() function
// nodes is a reference to a vector of all nodes created in _tmain()
// line_num is the pointer which give the current round number of lines
// that are connected to one of rings, but seperated from ring topology.
// rn is run number
bool check_rings(int time, vector<Node>& nodes, int *line_num, int rn, int comp_num) {
    int i, j;
    int s1, s2;
    vector <VEC_INT> rings;
    vector <VEC_INT> lines;
    int net_size = nodes.size();
    // Lines are connected directional segments, but not circular
}
String *file_name = Convert::ToString(rn);
file_name = String::Concat("rings_", file_name, ".txt");
StreamWriter *out = new StreamWriter(file_name, true);

cout << endl;
cout << "-----------------------------------------" << endl;
cout << "Time is " << time << endl;
cout << "rings:" << endl;
out->WriteLine();
out->WriteLine("-----------------------------------------");
out->Write("Time is ");
out->WriteLine(time);

for (i = 0; i < net_size; i++) nodes[i].visited = false;

for (i = 0; i < net_size; i++)
{
    Node n1 = nodes[i];
    Node n2;
    if (! nodes[i].visited)
    {
        vector <int> ring1;
        if (nodes[i].suc_index == -1) // Isolated node
        {
            ring1.push_back(i);
            rings.push_back(ring1);
            // An isolated node is regarded as ring
            // just by sense of connected component
            continue;
        }

        // --------------- Begin ----------------------- //
        // Following code constructs lines and rings by
        // tracing the successor
        j = i;
        while ((! nodes[j].visited) && (nodes[j].suc_index > -1))
        {
            n2 = nodes[j];
            nodes[j].visited = true;
            ring1.push_back(j);
            j = nodes[j].suc_index;
        }
        n2 = nodes[j];
        if (nodes[j].visited)
        {
            vector <int>::iterator result;
            result = find( ring1.begin(), ring1.end(), j);
            // Below if evaluated true means repeating node in ring1
            // ring1 comes back to itself, i.e. a ring is formed.
            if (!(result == ring1.end()))
            {
                vector <int> ring2;
                ring2.assign(result, ring1.end());
                rings.push_back(ring2);
                if (!(result == ring1.begin()))
                {
                    ring2.assign(ring1.begin(), result);
lines.push_back(ring2);
}
else lines.push_back(ring1);

if (nodes[j].suc_index == -1)
{
  _tprintf(_T("Semantics Error in check_ring() function. "));
  _tprintf(_T("A node without successor must have no neighbor.") );
  _tprintf(_T("So it can not be any other node's successor."));
  StreamWriter *out = new StreamWriter("error.txt", true);
  out->Write("Semantics Error in check_ring() function. ");
  out->Write("A node without successor must have no neighbor.");
  out->WriteLine("So it can not be any other node's successor.");
  out->Close();
  exit(EXIT_FAILURE);
}

// ---------------------- End ------------------------- //</
}
}

// ----------------------- Begin ---------  ------------- //</
// Following code prints rings (including isolated nodes //</
cout << "-----------------------------" << endl;
cout << "Rings" << endl << endl;
out->WriteLine("-----------------------------");
out->WriteLine("Rings");
out->WriteLine();
s1 = rings.size();
for (i = 0; i < s1; i++)
{
  vector <int> ring;
  VEC_INT &r = rings.at(i);
  ring.assign(r.begin(), r.end());
  cout << "Ring" << " " << endl;
  out->Write("Ring");
  out->Write(i);
  out->WriteLine(":");
  s2 = ring.size();
  for (j = 0; j < s2; j++)
  {
    int &a = ring.at(j);
    int b = a;
    cout << " " << b << " (" << nodes[b].nodeID << ")";
    out->Write(" ");
    out->Write(b);
    out->Write(" (";
    out->Write(nodes[b].nodeID);
    out->Write(" )");
  }
  int &a = ring.at(0);
  int b = a;
}
cout << " " << b << " (" << nodes[b].nodeID << ")";  
out->Write(" ");  
out->Write(b);  
out->Write(" (");  
out->Write(nodes[b].nodeID);  
out->Write(" )");  
cout << endl << endl;  
out->WriteLine();  
out->WriteLine();  
out->Flush();

// ----------------------- End ---------------------- //</code>

// ---------------------- Begin -------------------- //</code>
// Following code prints lines

s1 = lines.size();  
*line_num = s1;  
int nodes_in_lines = 0;  
bool all_lines_short = true;  
for (i = 0; i < s1; i++)  
{
    vector <int> line;
    
    VEC_INT &ln = lines.at(i);  
    line.assign(ln.begin(), ln.end());  
    s2 = line.size();  
    nodes_in_lines += s2;  
    if (s2 > SHORT_LINE_LEN) all_lines_short = false;  
    cout << "Line" << i << ":" << endl;  
    out->Write("Line");  
    out->Write(i);  
    out->WriteLine(";");  
    for (j = 0; j < s2; j++)  
    {
        int &a = line.at(j);  
        int b = a;  
        cout << " " << b << " (" << nodes[b].nodeID << ")";  
        out->Write(" ");  
        out->Write(b);  
        out->Write(" (");  
        out->Write(nodes[b].nodeID);  
        out->Write(" )");  
    }
    cout << endl << endl;  
    out->WriteLine();  
    out->WriteLine();  
    out->Flush();
};
cout << "-----------------------------" << endl << endl;
out->WriteLine("-----------------------------");
out->WriteLine();
out->Close();
// ----------------------- End ---------  ------------- //

if (all_lines_short && (nodes_in_lines <= nodes.size() * RATIO_IN_LINES) && (rings.size() == comp_num))
    return true;
else return false;
}

void check_compo_queue(vector<Node>& nodes)
{
    StreamWriter *out = new StreamWriter("component_queue.txt", true);
cout << endl;
cout << "----------------" << endl;
out->WriteLine();
out->WriteLine("----------------");

int s1 = nodes.size();
for (int i = 0; i < s1; i++)
{
    const Node &n1 = nodes.at(i);
cout << "Node " << i << ":" << endl;
out->Write("Node ");
out->Write(i);
out->WriteLine(":");

    // Print node i's component_queue
    // Begin
    int s2 = n1.component_queue.size();
cout << "component queue: ";
out->Write("component queue: ");
if (s2 == 0)
    { 
        cout << "empty for " << n1.compo_queue_empty_timer;
        out->Write("empty for ");
        out->Write(n1.compo_queue_empty_timer);
    }
else
    for (int j = 0; j < s2; j++)
    { 
        const BFS_Node &bn1 = n1.component_queue.at(j);
cout << bn1.nodeIdx << " ";
        out->Write(bn1.nodeIdx);
        out->Write(" ");
    }
cout << endl;
out->WriteLine();
// End

    // Print node i's dumped component_queue
    // Begin
s2 = n1.dumped_component.size();
cout << "dumped component queue: ";
out->Write("dumped component queue: ");
if (s2 == 0)
{
  cout << "empty." << endl;
  out->WriteLine("empty.");
}
else
  for (int j = 0; j < s2; j++)
  {
    const BFS_Node &bn1 = n1.dumped_component.at(j);
    cout << bn1.nodeIdx << " ";
    out->Write(bn1.nodeIdx);
    out->Write(" ");
  }
  cout << endl;
  out->WriteLine();
// End
out->Close();
}

// Primary control of simulation
// nodes is a reference to a vector of all nodes created in _tmain()
// out is the pointer to output.txt file
// rn is run number
void simulate(vector<Node>& nodes, StreamWriter *out, int rn,
int *time, int *sent, int *received, int comp_num)
{
  int i, j, net_size, T, p, interval, line_num;

  int ms = 0; // number of sent messages
  int mr = 0; // number of received messages, <= ms
  net_size = nodes.size();

  for (i = 0; i < net_size; i++)
  {
    Node &n1 = nodes.at(i);
    n1.al(nodes);
  }
  check_compo_queue(nodes);

  T = 0;
  p = 0;
  i = 0;
  line_num = 0;
  interval = CHECK_INTERVAL;
  while (true)
  {
    if (i >= MATURE_TIME)
    {
      if (T % interval == 0)
      {

155
if (p % 20 == 0) print_successor(i, nodes, rn);
p++;
check_compo_queue(nodes);
if (check_rings(i, nodes, &line_num, rn, comp_num)) break;
else if (line_num <= SMALL_LINE_NUM) interval = 1;
} 
T++;

// Begin checking if compo_queue_empty_timers
// of all nodes are too big
/* ---------------- Begin ---------------------- */
bool all_nodes_compo_empty_timer_big = true;
for (j = 0; j < net_size; j++)
if (nodes.at(j).compo_queue_empty_timer <
COMPO_QUE_EMPTY_TIME_LIMIT)
{
    all_nodes_compo_empty_timer_big = false;
    break;
}
if (all_nodes_compo_empty_timer_big) {
    cout << "Error in simaulate() function: compo_empty_timers ";
    cout << "of all nodes are too big.";<< endl;
    out->WriteLine();
    out->Write("Error in simaulate() function: compo_empty_timers ");
    out->WriteLine("of all nodes are too big.");
    out->WriteLine("Simulation is stopped.");
    out->WriteLine();
    out->Flush();
    break;
} /* ---------------- End ---------------------- */

/* --------- Begin scanning every node repeatedly --------- */
for (j = 0; j < net_size; j++)
{
    Node &n2 = nodes.at(j);
    n2.a2(nodes);
}

// Checking if all in_queues, out_queues of all
// nodes are empty.
/* ---------------- Begin ---------------------- */
bool all_queue_empty = true;
for (j = 0; j < net_size; j++)
{
    const Node &n2 = nodes.at(j);
    if (! n2.in_queue.empty())
    {
all_queue_empty = false;
break;
}
if (! n2.out_queue.empty())
{
    all_queue_empty = false;
    break;
}
if (all_queue_empty)
{
    cout << "Error in simaulate() function: all queues are empty ";
    cout << "after all nodes run a2. ";
    cout << "Simulation is stopped." << endl;
    out->WriteLine();
    out->Write("Error in simaulate() function: all queues are empty ");
    out->WriteLine("after all nodes run a2. ");
    out->WriteLine("Simulation is stopped.");
    out->Flush();
    break;
}

for (j = 0; j < net_size; j++)
{
    Node &n2 = nodes.at(j);

    int k = 0;
    while (!(n2.out_queue.empty()) && (k < MSG_PROC_RATE))
    {
        const Message &msg = n2.out_queue.front();
        unsigned int nxt = msg.next;
        Node &n3 = nodes.at(nxt);
        if (n3.in_queue.size() < IN_QUEUE_LENGTH)
            n3.in_queue.push_back(msg);
        n2.out_queue.pop_front();
        ms++;
        k++;
    }
}

for (j = 0; j < net_size; j++)
{
    Node &n2 = nodes.at(j);

    int k = 0;
    while (!(n2.in_queue.empty()) && (k < MSG_PROC_RATE))
    {
        const Message &msg = n2.in_queue.front();
        unsigned int nxt = msg.next;
        Node &n3 = nodes.at(nxt);
        if (n3.in_queue.size() < IN_QUEUE_LENGTH)
            n3.in_queue.push_back(msg);
        n2.in_queue.pop_front();
        ms++;
        k++;
    }
}
if (msg.next != n2.index)
{
    String *file_name = Convert::ToString(rn);
    file_name = String::Concat("error_", file_name, ".txt");
    StreamWriter *out1 = new StreamWriter(file_name, true);
    cout << "Error in simulate(): message has been ";
    cout << "forwarded to wrong node. node ID: ";
    cout << n2.index << endl;
    out1->Write("Error in simulate(): message has been ");
    out1->Write("forwarded to wrong node. node ID: ");
    out1->WriteLine(n2.index);
    out1->WriteLine();
    out1->Close();
    exit(EXIT_FAILURE);
}
if (msg.type == CLOSER_NODE)
{
    if (msg.receiver == n2.index)n2.a3(msg, nodes);
    else
    {
        String *file_name = Convert::ToString(rn);
        file_name = String::Concat("error_", file_name, ".txt");
        StreamWriter *out1 = new StreamWriter(file_name, true);
        cout << "Error in simulate(): CLOSER_NODE message has been ";
        cout << "sent to wrong node. node ID: ";
        cout << n2.index << endl;
        cout << "Error code in ::simulate() function, ";
        cout << "Processing Incoming Messages portion." << endl;
        out1->Write("Error in simulate(): CLOSER_NODE message has been ");
        out1->Write("sent to wrong node. node ID: ");
        out1->WriteLine(n2.index);
        out1->WriteLine("Error code in ::simulate() function, ");
        out1->WriteLine("Processing Incoming Messages portion.");
        out1->WriteLine();
        out1->Close();
        exit(EXIT_FAILURE);
    }
} else if (msg.type == SUCCESSOR_CANDIDATE) n2.a4(msg, nodes);
else
{
    String *file_name = Convert::ToString(rn);
    file_name = String::Concat("error_", file_name, ".txt");
    StreamWriter *out1 = new StreamWriter(file_name);
    cout << "Error in simulate: front message of";
    cout << " in_queue has wrong type. node " << n2.index;
    cout << endl;
    out1->Write("Error in simulate: front message of");
    out1->Write(" in_queue has wrong type. node ");
    out1->Write(n2.index);
    out1->WriteLine();
}
out1->Close();
exit(EXIT_FAILURE);
}
n2.in_queue.pop_front();
mr++;
k++;
}

} /* End Processing Incoming Messages */

///////////////////////////////////////////////////////////////////
* ------------- End scanning every node repeatedly ------------ */

i++;
}
print_successor(i, nodes, rn);
out->Write(i);
out->Write(" ");
out->Write(ms);
out->Write(" ");
out->WriteLine(mr);
out->Flush();
*time = i;
*sent = ms;
*received = mr;
}

// This is the entry point for this application
int _tmain(void)
{
  int i;
  int k = RUN_TIME;
  int n; // NETWORK_SIZE
  DirectoryInfo *pinfo;

  for (n = 20; n <= 20; n += 20)
  {
    String *path = Convert::ToString(n);
    Directory::CreateDirectory(path);
    Directory::SetCurrentDirectory(path);
    StreamWriter *out = new StreamWriter("output.txt");
    print_parameters(out, n);
    int total_time = 0;
    int total_sent = 0;
    int total_received = 0;
    for (i = 0; i < k; i++)
    {
      int time = 0;
      int sent = 0;
      int received = 0;
int comp_num;
vector<Node> net(n);
vector.COMPONENT comps;

init_net(net, i);
find_neighbors(net, i);
connected_components(net, comps, i, &comp_num);
out->WriteLine();
out->Write("simulation_time ");
out->Write("sent_messages# ");
out->WriteLine("received messages#");
out->Flush();
simulate(net, out, i, &time, &sent, &received, comp_num);
total_time += time;
total_sent +=sent;
total_received += received;
}
out->Write((double) total_time / k);
out->Write(" ");
out->Write((double) total_sent / k);
out->Write(" ");
out->WriteLine((double) total_received / k);
out->Flush();
out->Close();
path = Directory::GetCurrentDirectory();
pinfo = Directory::GetParent(path);
Directory::SetCurrentDirectory(pinfo->FullName);
)

return 0;
}
VITA

Wei Ding received his Master of Science degree in computer science and technology from the University of Science and Technology of China, Hefei, People’s Republic of China, in 1996. He is a registered engineer in China. He is working toward his doctoral degree in computer science at the Louisiana State University, Baton Rouge, Louisiana. He just successfully passed his dissertation defense on July 10, 2006. He is a member of the Institute of Electrical and Electronics Engineers and the Association for Computing Machinery. His research interests include wireless networks, network security, computer simulation, intelligent control, and artificial intelligence.