การจัดการระบบรับประกันคุณภาพของระบบเครือข่ายไร้สาย ที่แอ็คเซสพอยท์ภายใต้ระบบปฏิบัติการลีนุกซ์

QOS MANAGEMENT OF WIRELESS NETWORK WITH LINUX ACCESS POINT

โดย

ศิริศักดิ์ โลหะพิริยกุล

เอกนันท์ นันทะชิน

อาจารย์ที่ปรึกษา

อาจารย์ลภัส ประดิษฐ์ทัศนีย์

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรวิทยาศาสตรบัณฑิต

สาขาวิชาเทคโนโลยีสารสนเทศ

คณะเทคโนโลยีสารสนเทศ

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ภาคเรียนที่ 2 ปีการศึกษา 2549

QOS MANAGEMENT OF WIRELESS NETWORK
WITH LINUX ACCESS POINT

SIRISAK LOHAPHIRIYAKUL

EAKKANAN NANTHACHIN

A PROJECT SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENT FOR THE DEGREE OF

BACHELOR OF SCIENCE PROGRAM IN INFORMATION TECHNOLOGY

FACULTY OF INFORMATION TECNOLOGY

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

2/2006

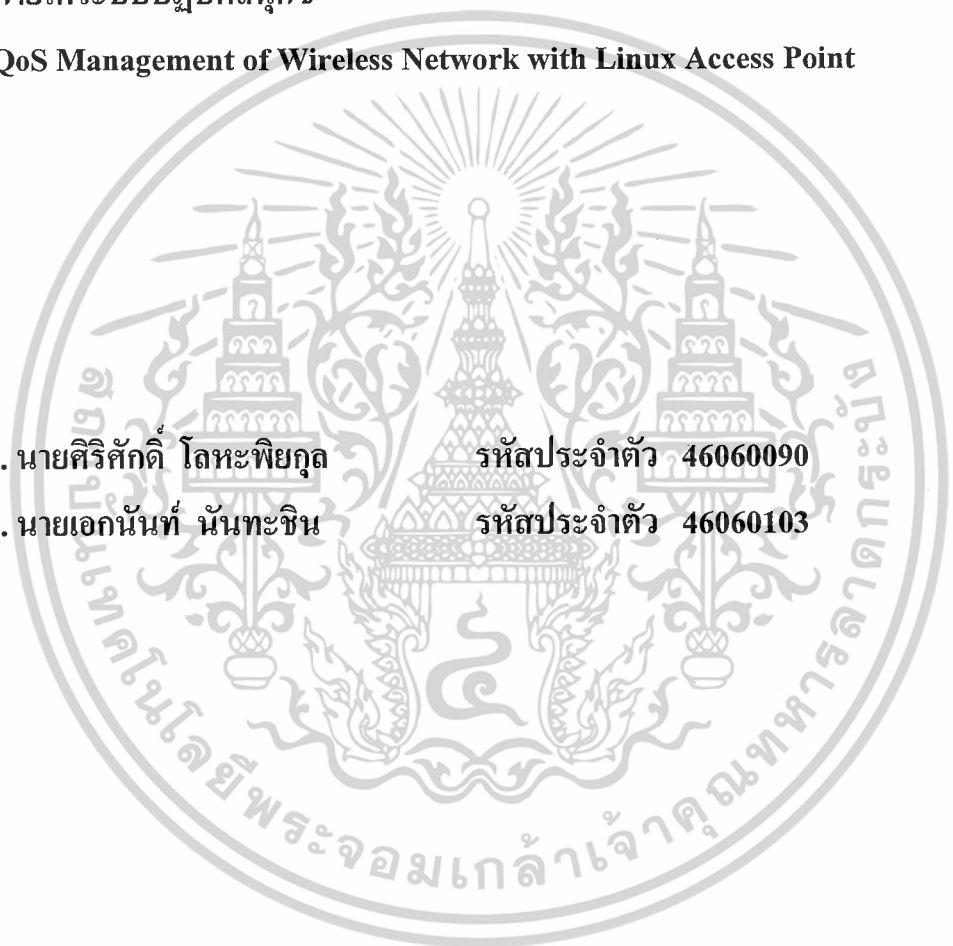
ใบรับรองปริญญานิพนธ์ ประจำปีการศึกษา 2549

คณะเทคโนโลยีสารสนเทศ

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง การจัดการระบบรับประกันคุณภาพของระบบเครือข่ายไร้สายที่แอคเซสพอยต์ภายใต้ระบบปฏิบัติลีนุกซ์

QoS Management of Wireless Network with Linux Access Point



ผู้จัดทำ

1. นายศิริศักดิ์ โลหะพิยกุล รหัสประจำตัว 46060090
2. นายเอกนันท์ นันทะชิน รหัสประจำตัว 46060103

.....อาจารย์ที่ปรึกษา

(อาจารย์ลภัส ประดิษฐ์ทัศนีย์)

หัวข้อ	การจัดการระบบรับประกันคุณภาพของระบบเครือข่ายไร้สายที่แอ็คเซสพอยท์ภายใต้ระบบปฏิบัติการลีนุกซ์	
นักศึกษา	นายศิริศักดิ์ โลหะพิริยกุล	46060090
	นายเอกนันท์ นันทะชิน	46060103
ปริญญา	วิทยาศาสตรบัณฑิต	
สาขาวิชา	เทคโนโลยีสารสนเทศ	
ปีการศึกษา	2549	
อาจารย์ที่ปรึกษา	อาจารย์ลภัส ประดิษฐ์ทัศนีย์	

บทคัดย่อ

เนื่องจากการใช้งานระบบเครือข่ายไร้สายในปัจจุบันนั้นได้รับความนิยมเป็นอย่างสูง ทำให้มีข้อมูลต่างๆ หลากหลายชนิดถูกส่งผ่านในระบบเครือข่ายไร้สาย และเมื่อมีผู้ใช้งานมากการส่งข้อมูลผ่านไวร์เลสแอ็กเซสพอยท์จึงมีมากตามไปด้วยเช่นกัน ทำให้เกิดการแย่งชิงช่องสัญญาณที่จะใช้ส่งข้อมูลจนส่งผลให้คุณภาพของสัญญาณที่ให้บริการกับผู้ใช้ไม่มีประสิทธิภาพมากเพียงพอ ซึ่งปัจจัยสำคัญมาจากไวร์เลสแอ็คเซสพอยท์โดยมากที่มีขายอยู่ในท้องตลาดนั้นไม่มีความยืดหยุ่นในการที่จะให้ผู้ใช้สามารถทำการปรับแต่งค่าต่างๆ ให้เหมาะสมกับพฤติกรรมการใช้งานระบบเครือข่ายไร้สายของผู้ใช้ได้

ดังนั้น โครงงานนี้จึงได้มีการนำเครื่องคอมพิวเตอร์ที่ใช้ระบบปฏิบัติลีนุกซ์ซึ่งประกอบด้วยการ์ดอินเตอร์เฟส 2 ใบ คือ การ์ดอีเทอร์เน็ตและการ์ดไวร์เลสมาทำงานเป็นแอ็คเซสพอยต์ที่สนับสนุนระบบรับประกันคุณภาพบนเครือข่ายไร้สาย โดยทำการรับรองคุณภาพของสัญญาณให้เป็นไปตามแบนวิธที่ผู้ดูแลระบบกำหนดให้ตามแต่ละโปรแกรมประยุกต์ใช้งาน ซึ่งการทำงานของแอ็คเซสพอยต์นั้นจะทำงานอยู่ในระดับของเดทาลิงค์ จึงได้ทำการพัฒนาโปรแกรมเป็นโมดูลและทำการใส่โมดูลนั้นลงไปในระดับชั้นเคอร์เนลของระบบปฏิบัติการ เพื่อทำการสร้างช่องทางการรับส่งข้อมูลระหว่างระดับชั้นเดทาลิงท์ระหว่างการ์ดอีเทอร์เน็ตและการ์ดไวร์เลสโดย นอกจากนี้ตัวโมดูลยังมีหน้าที่ในการจัดสรรแบนวิธให้ข้อมูลของแต่ละโปรแกรมประยุกต์ให้เป็นไปตามที่กำหนดไว้ อีกทั้งได้ทำการพัฒนาในส่วนของเว็บอินเตอร์เฟสเพื่อให้ผู้ใช้สามารถปรับแต่งการทำงานของไวร์เลสแอ็คเซสพอยท์ได้สะดวก รวดเร็ว และมีประสิทธิภาพ

Title	QoS Management of Wireless Network with Linux Access Point	
Student	Mr. Sirisak Lohaphiriyakul	46060090
	Mr. Eakkanan Nanthachin	46060103
Degree	Bachelor of Science	
Program	Information Technology	
Academic Year	2006	
Advisor	Lapas Pradittasnee	

ABSTRACT

Nowadays, the popularity of Wireless Local Area Network (WLAN) is increasing every year. Leading to various types of data are traversed through the wireless network. Access Points have to serving high number of users, which create the contention for available bandwidth in the network. In many cases, users do not receive the acceptable level of service they need. One of the important factors to this problem is most of the access point, which available in the market today, do not provide the flexibility for users to change the configuration to fit the traffic behavior of their network they manage.

This project use the computer that run Linux Operating system that included 2 network interface cards, Ethernet and Wireless adapters, to build the access point that can support Quality of Service (QoS) in wireless network. QoS is provided based on the value of the bandwidth of each application program, which can be configured by user. The project development is based on creating program module that work at Data Link layer of OSI model and insert the module in the kernel of operating system. The module will act as the bridge between Ethernet and Wireless adapter. Furthermore the module also responsible for allocate the bandwidth for each application program as the rules set. The web interface also includes helping user to configuring access point parameters more easily and effective.

กิตติกรรมประกาศ

ปริญญานิพนธ์เล่มนี้สำเร็จได้ด้วยความกรุณาจากอาจารย์ที่ปรึกษา อาจารย์ลภัส ประดิษฐ์ทัศนีย์ ที่ให้ความช่วยเหลือ ให้คำแนะนำช่วยแก้ปัญหาตลอดจนให้ความรู้และประสบการณ์ที่ดีแก่ข้าพเจ้า

ขอขอบพระคุณ รศ.ดร. โชติพัชร์ ภรณวลัย ผศ. อัครินทร์ คุณกิตติ และ อาจารย์สุเมธ ประภาวัต กรรมการสอบปริญญานิพนธ์ที่ได้กรุณาให้คำแนะนำตลอดจนข้อชี้แนะ จนในที่สุดทำให้ปริญญานิพนธ์ฉบับนี้สำเร็จลงได้

สำหรับคุณงามความดีอันใดที่เกิดจากวิทยานิพนธ์ฉบับนี้ ผู้จัดทำขอมอบให้กับบิดามารดา ซึ่งเป็นที่รักและเคารพยิ่ง ตลอดจนครูอาจารย์ที่เคารพทุกท่านที่ได้ประสิทประสาทวิชาความรู้และถ่ายทอดประสบการณ์ที่ดีให้

ศิริศักดิ์ โลหะพิริยกุล

เอกนันท์ นันทะชิน

สารบัญ

หน้า

บทคัดย่อภาษาไทย I

บทคัดย่อภาษาอังกฤษ II

กิตติกรรมประกาศ III

สารบัญ IV

สารบัญตาราง VII

สารบัญภาพ VIII

บทที่1 บทนำ 1

1.1 ความเป็นมาและความสำคัญของปัญหา 1

1.2 ความมุ่งหมายและวัตถุประสงค์ของการศึกษา 1

1.3 ประโยชน์ของโครงงาน 1

1.4 ขอบเขตของโครงงาน 2

บทที่ 2 ทฤษฎีเกี่ยวกับเครือข่ายและลีนุกซ์ 3

2.1 TCP/IP Model 3

2.2 ระบบเครือข่ายเฉพาะที่ 4

2.2.1 การเชื่อมต่อแบบบัส 5

2.2.2 การเชื่อมต่อแบบวงแหวน 5

2.2.3 การเชื่อมต่อแบบดาว 5

2.3 การทำงานของเครือข่ายไร้สาย 6

2.3.1 ชนิดของเครือข่ายไร้สาย 8

2.3.1.1 Independent network (IBSS) 8

2.3.1.2 Infrastructure network 8

2.3.1.3 Extended service areas 9

2.3.2 การทำงานของเครือข่าย 802.11 10

2.3.3 MAC Frame format 11

2.3.4 การค้นหาเครือข่ายไร้สาย 13

2.3.4.1 Beaconing 13

2.3.4.2 Scanning 14

2.3.5 การเข้าร่วมเครือข่าย 14

สารบัญ(ต่อ)

หน้า

2.3.5.1การรับรองสิทธิ์ 14

2.3.5.2 การสร้างความสัมพันธ์ 15

2.4 ลีนุกซ์ 16

2.4.1 การใช้งาน 17

2.4.2 โครงสร้างของแฟ้มข้อมูล 17

2.4.3 BASH Shell 18

2.4.4 การติดต่อระหว่างอุปกรณ์กับระบบปฏิบัติการลีนุกซ์ 19

2.5 การประกันคุณภาพ 23

บทที่ 3 การสร้างไวร์เลสแอ็คเซสพอยต์ 26

3.1 การติดตั้งระบบปฏิบัติการลีนุกซ์ 26

3.2 การติดตั้งไดร์เวอร์ของการ์ดเครือข่ายไร้สาย 26

3.2.1 ความต้องการ 26

3.2.2 การติดตั้ง 27

3.3 การทำ Host-AP บนลีนุกซ์ ด้วย MADWiFi 28

บทที่4 การพัฒนาโครงงาน 31

4.1 ภาพรวมของโครงงาน 31

4.2 การพัฒนาโครงงานในระดับเคอร์เนล 32

4.2.1 การพัฒนาส่วนติดต่อระหว่างอินเตอร์เฟส 32

4.2.1.1 ไลบารีที่สำคัญ 33

4.2.1.2 การพัฒนาโมดูลเชื่อมอินเตอร์เฟสเบื้องต้น 40

4.2.2 การพัฒนาในส่วนการรับประกันคุณภาพ 43

4.2.2.1 การพัฒนาในส่วนการจัดคิว 43

4.2.2.2 เธร็ด 51

4.3 โปรแกรมในส่วนของการปรับแต่งการทำงานของแอ็คเซสพอยท์ 59

4.3.1 ส่วนที่ทำงานอยู่ในระดับยูเซอร์ 59

4.3.2 แนะนำภาษา PHP 59

4.3.3 การใช้งาน PHP ร่วมกับ HTML 59

สารบัญ(ต่อ)

หน้า

4.3.4 ตัวอย่างฟังก์ชันของ PHP ที่ใช้ในการสร้างโปรแกรมติดต่อกับผู้ใช้ 60

4.3.5 ตัวอย่างหน้าจอโปรแกรม 61

บทที่ 5 การทดลอง 63

5.1 อุปกรณ์ที่ใช้ในการทดลอง 63

5.2 การจัดเตรียมด้านฮาร์ดแวร์ 63

5.3 การจัดเตรียมด้านซอฟท์แวร์ 63

5.4 วิธีการทดลอง 64

5.4.1 การทดลองที่ 1 64

5.4.2 การทดลองที่ 2 65

5.5 ผลการทดลอง 66

5.5.1 ผลการทดลองที่ 1.1 66

5.5.2 ผลการทดลองที่ 1.2 69

5.5.3 ผลการทดลองที่ 1.3 72

5.5.4 ผลการทดลองที่ 2 75

5.6 สรุปผลการทดลอง 75

บทที่ 6 บทสรุป 76

6.1 บทสรุป 76

6.2 ปัญหาและอุปสรรคในการพัฒนา 76

6.3 แนวทางในการพัฒนาต่อ 77

บรรณานุกรม 78

ภาคผนวก ก 80

ภาคผนวก ข 83

สารบัญตาราง

ตารางที่ หน้า

2.1 ข้อมูลบนอินเตอร์เน็ตที่มีความไวต่างๆ 24

5.1 อัตราส่วนน้ำหนักของข้อมูลของแต่ละพอร์ตในแต่ละการทดลอง 61

5.2 ค่าน้ำหนักที่กำหนดให้แต่ละพอร์ทตลอดทั้งการทดลอง 66

5.3 อัตราทรูพุทที่ใช้ในการส่งของแต่ละพอร์ทในแต่ละการทดลอง 66

5.4 ตารางแสดงค่าทรูพุทของแต่ละพอร์ตในการทดลองที่ 1.1 64

5.5 ตารางแสดงค่าทรูพุทของแต่ละพอร์ตในการทดลองที่ 1.2 67

5.6 ตารางแสดงค่าทรูพุทของแต่ละพอร์ตในการทดลองที่ 1.3 69

5.7 ตารางแสดงค่าทรูพุทของฝั่งรับที่ได้จากแต่ละการทดลองในการทดลองที่ 2 75

สารบัญภาพ

รูปที่ หน้า

2.1 แบบจำลองทีซีพี/ไอพี (TCP/IP model) 3

2.2. การเชื่อมต่อแบบบัส 5

2.3 การเชื่อมต่อแบบวงแหวน 5

2.4 การเชื่อมต่อแบบดาว 6

2.5 IEEE802.11 specification 6

2.6 องค์ประกอบฟิสิคอล 7

2.7 เครือข่ายแอดฮ็อก 8

2.8 Infrastructure networks 9

2.9 Extended Service areas 10

2.10 โครงสร้างของเมเนจเมนต์เฟรม 12

2.11 เฟรมควบคุม 12

2.12 เฟรมข้อมูล 12

2.13 การรับรองสิทธิ์แบบใช้กุญแจร่วม 15

2.14 การใช้งานบนลีนุกซ์ 17

2.15 ตัวอย่างโครงสร้างไฟล์บนลีนุกซ์ 17

2.16 โครงสร้างการทำงานในระดับเคอร์แนล 19

2.17 การเชื่อมโมดูลกับเคอร์เนล 23

4.1 แสดงภาพรวมของโครงงาน 32

4.2 แสดงโครงสร้างของข้อมูลชนิด sk_buff 32

4.3 แสดงการทำสำเนาโครงสร้างข้อมูลจากฟังก์ชัน skb_copy() 34

4.4 การทำงานของฟังก์ชัน skb_clone() 35

4.5 แสดงโครงสร้างข้อมูลของตัวแปรชนิด net_device 36

4.6 แสดงถึงโครงสร้างข้อมูลของตัวแปรชนิด sk_buff_head 44

4.7 แสดงขั้นตอนการใส่ข้อมูลลงคิว 46

4.8 แสดงวิธีในการเปลี่ยนพอร์ทปลายทาง 47

4.9 แสดงการถึงข้อมูลออกจากคิว 49

4.10 หน้าจอดูสถานะต่างๆของแอ็กเซสพอยต์ 61

4.11 หน้าจอที่ใช้ในการปรับแต่งค่าของแอ็กเซสพอยต์ 61

4.12 หน้าจอในส่วนของการรับประกันคุณภาพ 62

สารบัญภาพ(ต่อ)

รูปที่ หน้า

5.1 รูปการทดลองที่ 1 64
5.2 รูปการทดลองที่ 2 65
5.3 กราฟแสดงทรูพุธในแต่ละพอร์ตที่ได้จากโปรแกรม LanTraffic V2 ในการทดลองที่ 1.1 68
5.4 แสดงทรูพุธในแต่ละพอร์ตที่ได้จากโปรแกรม Ethereal ในการทดลองที่ 1.1 68
5.5 กราฟแสดงทรูพุธในแต่ละพอร์ตที่ได้จากโปรแกรม LanTraffic V2 ในการทดลองที่ 1.2 71
5.6 กราฟแสดงทรูพุธในแต่ละพอร์ตที่ได้จากโปรแกรม Ethereal ในการทดลองที่ 1.2 71
5.7 กราฟแสดงทรูพุธในแต่ละพอร์ตที่ได้จากโปรแกรม LanTraffic V2 ในการทดลองที่ 1.3 73
5.8 กราฟแสดงทรูพุธในแต่ละพอร์ตที่ได้จากโปรแกรม Ethereal ในการทดลองที่ 1.3 74
ก.1 แสดงเมื่อเข้าหน้า properties ของอินเตอร์เฟสไร้สาย 81
ก.2 แสดงหน้า internal protocol โดยต้องเลือกในช่อง “obtain an IP address au tomatically” 82
ก.3 แสดงเมื่อกำหนดค่าต่างๆ ได้ถูกต้องก็สามารถเข้าใช้เครือข่ายปกติได้ 82

บทที่ 1
บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

เนื่องจากในปัจจุบันอุปกรณ์เครือข่ายแบบไร้สายใน ที่มีวางขายในท้องตลาดส่วนใหญ่เป็นระบบปิด ซึ่งไม่อนุญาตให้บุคคลภายนอกสามารถปรับปรุงแก้ไของค์ประกอบใด ๆ ได้ และการปรับแต่งค่าการทำงานของตัวอุปกรณ์นั้นยังทำได้ในระดับพื้นฐานเท่านั้น ทำให้การศึกษาและพัฒนาระบบการทำงานของระบบการสื่อสารไร้สาย จำเป็นต้องมีการนำเครื่องคอมพิวเตอร์มาทำงานเป็นไวร์เลสแอ็คเซสพอยท์เพื่อให้สามารถทำการปรับแต่งค่าพารามิเตอร์ต่าง ๆ ให้สามารถทำงานตามความต้องการของนักวิจัยได้ นอกจากนี้ในโครงงานนี้ยังทำการพัฒนาโปรแกรมอีกโปรแกรมหนึ่งมาเพื่อให้ผู้ใช้สามารถควบคุมการรับส่งข้อมูลในระบบเครือข่ายแบบไร้สายให้มีประสิทธิภาพสูงที่สุดตามเงื่อนไข ณ เวลานั้น ๆ หรือ เพื่อให้เป็นไปตามความต้องการของผู้ใช้

1.2 ความมุ่งหมายและวัตถุประสงค์ของการศึกษา

- เพื่อพัฒนาสร้างซอฟต์แวร์ที่ทำงานบนระบบปฏิบัติการลีนุกซ์ในการทำให้เครื่องคอมพิวเตอร์ธรรมดาสามารถทำงานเป็นไวร์เลสแอ็คเซสพอยท์ได้ใกล้เคียงกับไวร์เลสแอ็คเซสพอยท์จริง ๆ ให้มากที่สุด
- เพื่อพัฒนาโปรแกรมในการปรับแต่งค่าพารามิเตอร์ต่าง ๆ ที่จะช่วยให้การรับส่งข้อมูลที่เดินทางผ่านแอ็คเซสพอยท์มีประสิทธิภาพมากที่สุด หรือ เป็นไปตามความต้องการของผู้ใช้ได้
- สร้างความสะดวกให้กับผู้ใช้ทั่วๆ ไปที่มีความความรู้ทางด้านเครือข่ายไร้สายไม่มากนักสามารถใช้งานและปรับแต่งการทำงานให้เหมาะสมกับเครือข่ายที่ใช้อยู่ได้

1.3 ประโยชน์ของโครงงาน

- ประยุกต์เครื่องคอมพิวเตอร์ให้สามารถทำหน้าที่เป็นแอคเซสพอยต์ตามที่ผู้ใช้ต้องการได้
- ประหยัดค่าใช้จ่ายในการติดตั้งหรือค่าอุปกรณ์แอคเซสพอยต์
- สามารถตรวจสอบการทำงานจากเครื่องคอมพิวเตอร์ที่เป็นแอคเซสพอยต์ได้
- เพื่อให้ผู้ใช้ปรับแต่งค่าต่างๆ ได้ง่าย

- เพื่อพัฒนาโปรแกรมที่ใช้จัดการเกี่ยวกับการประกันคุณภาพ (QoS) เบื้องต้นที่ตัวแอคเซสพอยท์ให้สามารถจัดการกราฟิกในระบบเครือข่ายไร้สายให้เป็นไปตามที่ผู้ใช้ต้องการได้

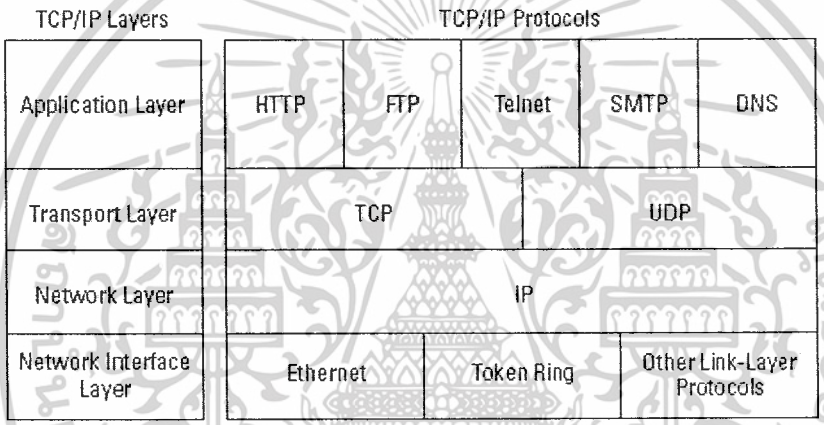
1.4 ขอบเขตของโครงงาน

- ประยุกต์ให้คอมพิวเตอร์มีคุณสมบัติเหมือนเป็นอุปกรณ์แอคเซสพอยต์จริงๆ
- แอคเซสพอยต์จำลองที่สร้างขึ้นสามารถรองรับการให้บริการได้อย่างมีคุณภาพ
- ปรับแต่งในส่วนการประกันคุณภาพได้ตามความต้องการของผู้ใช้

บทที่ 2
ทฤษฎีเกี่ยวกับเครือข่ายและลีนุกซ์

2.1 TCP/IP Model

เป็นรูปแบบข้อตกลงที่ใช้ในการออกแบบเครือข่ายที่ใช้สื่อสารกันทั่วไป ระหว่างอุปกรณ์คอมพิวเตอร์กับอุปกรณ์ทางเครือข่าย โดยโมเดลนี้จะทำการแบ่งการทำงานออกเป็นระดับชั้น โดยมีการติดต่อกันในระดับชั้นที่อยู่ติดกัน โดยประกอบไปด้วยทั้งหมด 5 ชั้นได้แก่



รูปที่ 2.1 แบบจำลองทีซีพี/ไอพี (TCP/IP model)

- ชั้นแอพพลิเคชั่น เป็นระดับชั้นที่สนับสนุนการสื่อสารระหว่างแอพพลิเคชั่นกับแอพพลิเคชั่น ตัวอย่างแอพพลิเคชั่น เช่น การขนส่งไฟล์ (File Transfer), จดหมายอิเล็กทรอนิกส์ (Electronic mail), ข่าวสารเกี่ยวกับโฮสต์, การบริหารจัดการเครือข่าย (Network Management)
- ชั้นทรานสปอร์ต จะรับผิดชอบในการเคลื่อนย้ายข้อมูลระหว่างโปรเซสของผู้รับและโปรเซสของผู้ส่ง โดยในขณะใดขณะหนึ่งอาจจะมีหลายโปรเซสกำลังทำงานอยู่ ดังนั้นในชั้นนี้จะรับผิดชอบการรับส่งข้อมูลไปให้ถึงโปรเซสที่ต้องการ โปรโตคอลในชั้นนี้สามารถให้บริการได้หลายๆ แอพพลิเคชั่นในเวลาเดียวกัน โดยการกำหนดที่อยู่พอร์ต เพื่อใช้เป็นตำแหน่งอ้างอิงสำหรับการติดต่อกับแต่ละ

แอพพลิเคชั่นที่อยู่อีกฝั่งหนึ่ง ซึ่งการเชื่อมต่อกับพอร์ตจะอิงกับกลไกของซ็อกเก็ต (socket)

- ชั้นเนตเวิร์ก รับผิดชอบในการเลือกเส้นทางและจัดการส่งข้อมูลระหว่างโฮสต์ ซึ่งสิ่งที่ใช้ในการส่งในชั้นนี้คือ ลอจิคัลแอดเดรส(โดยมากเป็นไอพีแอดเดรส) ข้อมูลที่ถูกสร้างขึ้นนี้จะถูกเรียกว่าแพ็กเก็ต
- ชั้นดาต้าลิงค์ เป็นชั้นที่ทำหน้าที่เชื่อมต่อการรับส่งข้อมูลในระดับฮาร์ดแวร์ โดยเมื่อมีการสั่งให้รับข้อมูลจากในชั้นเน็ตเวิร์กลงมา ชั้นดาต้าลิงค์จะทำหน้าที่แปลคำสั่งนั้นให้เป็นคำสั่งควบคุมฮาร์ดแวร์ที่ใช้รับส่งข้อมูล ทำการตรวจสอบข้อผิดพลาดในการรับส่งข้อมูลของระดับฮาร์ดแวร์ และแก้ไขข้อผิดพลาดที่ตรวจพบนั้น ข้อมูลที่อยู่ในชั้นดาต้าลิงค์นี้จะอยู่ในรูปของเฟรม(Frame) คือกลุ่มของข้อมูลที่มีรูปร่างตามข้อบังคับของฮาร์ดแวร์ที่ใช้ในการรับส่งข้อมูล เช่น ถ้าฮาร์ดแวร์ที่ใช้เป็นอีเทอร์เน็ตแลน(Ethernet LAN) ข้อมูลก็จะมีรูปร่างของเฟรมตามที่ระบุไว้ในมาตรฐานของอีเทอร์เน็ต หากว่าฮาร์ดแวร์ที่ใช้รับส่งข้อมูลเป็นชนิดอื่น เช่น เครือข่ายวงแหวนโทเค็น(Token Ring LAN) หรือเอฟดีดีไอ(Fiber Distributed Data Interface) รูปร่างของเฟรมที่ใช้ในการรับส่งข้อมูลก็จะเปลี่ยนไปตามมาตรฐานนั้นๆ
- ชั้นฟิสิคัล เป็นชั้นที่อยู่ล่างสุด ระดับชั้นนี้จะรับผิดชอบเกี่ยวกับการส่งข้อมูลที่เป็นบิต ซึ่งก็คือเลข 0 กับ 1 ในระบบเลขฐานสอง (Binary) โดยในฝั่งผู้ส่งชั้นนี้จะทำการรับข้อมูลจากชั้นดาต้าลิงค์ ซึ่งข้อมูลที่รับมาจะเป็นเฟรม (Frame) แล้วทำการส่งข้อมูลนี้ออกไปทีละบิตแบบเรียงลำดับ ส่วนในฝั่งผู้รับ ชั้นฟิสิกคัลก็จะรับข้อมูลที่ส่งมาทีละบิตและจัดส่งผ่านข้อมูลที่เป็นบิตนี้ให้ให้ชั้นดาต้าลิงค์ประมวลผลต่อไป

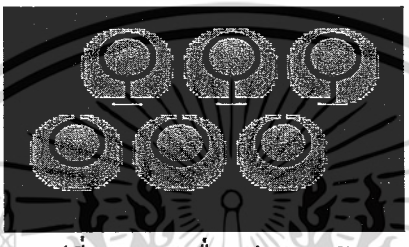
2.2 ระบบเครือข่ายเฉพาะที่ (Local Area Network : LAN)

เป็นการเชื่อมโยงคอมพิวเตอร์เข้าด้วยกันให้เป็นเครือข่ายโดยมีอุปกรณ์และสื่อกลางต่างๆ ใช้เป็นตัวเชื่อมโยงคอมพิวเตอร์เข้าด้วยกัน โดยเครือข่ายท้องถิ่นเป็นเครือข่ายที่มีขนาดไม่ใหญ่มาก มักอยู่ในองค์กรๆ หนึ่งโดยอาจจะมีการเชื่อมโยงกลับเครือข่ายภายนอกด้วยหรือไม่ก็ได้ โดยรูปแบบในการเชื่อมอุปกรณ์เข้าด้วยมีอยู่หลากหลายรูปแบบ ไม่ว่าจะเป็น แบบเม็ต(mesh) , แบบดาว(star) , แบบริง

(ring), แบบบัส(bus) และแบบไฮบริดจ์(hybrid) ส่วนมาตรฐานของเทคโนโลยีที่นิยมใช้ในเครือข่ายท้องถิ่นคือ อีเทอร์เน็ต

2.2.1. การเชื่อมต่อแบบบัส (Bus Topology)

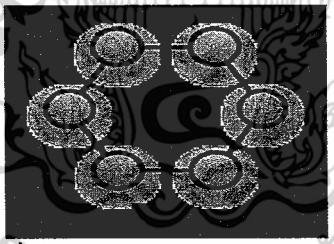
ทอพอลอยีชนิดนี้เป็นการเชื่อมต่อกันของคอมพิวเตอร์เป็นเครือข่าย โดยใช้สายสัญญาณเพียงเส้นเดียวเป็นเส้นหลัก (Trunk) ข้อดีของทอพอลอยีแบบนี้คือมีความสะดวกและยืดหยุ่นหากต้องการขยายขนาดเครือข่าย แต่ก็มีข้อจำกัดในเรื่องความยาวของสายสัญญาณ



รูปที่ 2.2. การเชื่อมต่อแบบบัส

2.2.2. การเชื่อมต่อแบบวงแหวน (Ring Topology)

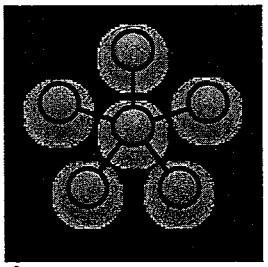
เป็นการเชื่อมต่อแบบวงแหวนหรือลูป (Loop) โดยที่แต่ละโหนด (Node) บนเครือข่ายมีจุดเชื่อมต่อทั้งสองด้านเพื่อเชื่อมต่อโหนดต้นทางและปลายทางเข้าหากันเป็นลูป ตัวอย่างของเครือข่ายชนิดนี้คือ โทเค่นริง (Token Ring)



รูปที่ 2.3 การเชื่อมต่อแบบวงแหวน

2.2.3. การเชื่อมต่อแบบดาว (Star Topology)

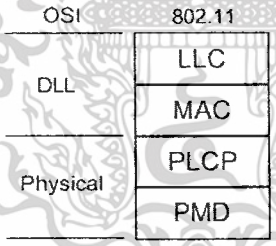
ลักษณะนี้เป็นการเชื่อมต่อที่มีคอมพิวเตอร์หรือโฮสต์ (Host) ต่างๆ ไปที่ศูนย์กลาง (Centralized) โดยในระบบเครือข่ายแลนจะใช้ฮับ (Hub) หรือสวิทช์ (Switch) เป็นจุดศูนย์กลาง



รูปที่ 2.4 การเชื่อมต่อแบบดาว

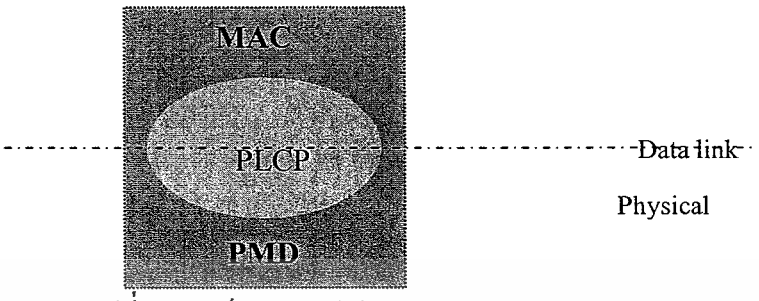
2.3 การทำงานของเครือข่ายไร้สาย

เครือข่ายไร้สายได้ใช้เทคโนโลยีตามมาตรฐาน IEEE 802.11 ซึ่งเป็นหนึ่งในสมาชิกของมาตรฐาน IEEE 802 ซึ่งเป็นมาตรฐานที่กำหนดรายละเอียดที่เกี่ยวกับเครือข่ายแลน โดย IEEE 802 จะดูแลการทำงานในสองเลเยอร์ล่างสุดของแบบจำลองโอเอสไอ (OSI model) ซึ่งก็คือ ระดับชั้นดาต้าลิงค์ (Data Link layer) และ ระดับชั้นฟิสิคัล(Physical layer) ซึ่งในระดับชั้นดาต้าลิงค์จะมีแมค (MAC: Media Access Control) เป็นตัวควบคุมการเข้าถึงตัวกลางและการส่งข้อมูล และแอลแอลซี(LLC: Logical Link Control) จะสร้างและดูแลการเชื่อมต่อระหว่างอุปกรณ์ ส่วนรายละเอียดในการส่งและการรับข้อมูลนั้นเป็นหน้าที่ของระดับชั้นฟิสิคัล



รูปที่ 2.5 IEEE802.11 specification

การที่ระดับชั้นฟิสิคัลของเครือข่ายไร้สายใช้คลื่นวิทยุเป็นตัวกลางนั้น เป็นเรื่องที่ซับซ้อน 802.11 จึงได้ทำการแบ่งฟิสิคัลคอมโพเนนท์ (Physical component) ออกเป็นสองชนิดคือ Physical Layer Convergence Procedure (PLCP) ซึ่งจะใช้สำหรับแปลงฟอร์แมตของข้อมูลผู้ใช้ให้สามารถส่งไปบนระบบรับส่งสัญญาณได้ และคอมโพเนนท์อีกตัวคือ Physical Medium Dependent (PMD) ซึ่งใช้สำหรับรับส่งข้อมูลซึ่งจะขึ้นอยู่กับเทคโนโลยีที่ใช้



รูปที่ 2.6 องค์ประกอบฟิสิคอล

เครือข่าย 802.11 ประกอบด้วย 4 องค์ประกอบหลัก คือ

- ระบบกระจาย (Distribution system)

  Distribution system จะให้บริการโดยการเชื่อมต่อแอคเซสพอยต์ (access point) ต่างๆ เข้าด้วยกัน เมื่อมีเฟรมข้อมูลมาถึงระบบกระจาย (distribution system) มันจะทำการส่งเฟรมนั้นไปยังแอคเซสพอยต์ที่ดูแลสถานี (station) ปลายทางอยู่และ แอคเซสพอยต์ปลายทางจะทำการส่งเฟรมข้อมูลที่ได้รับมาไปให้ สถานีปลายทาง หน้าที่ของระบบกระจายคือการตรวจสอบว่า สถานีอยู่ที่ใด เพื่อที่จะทำการส่งเฟรมไปให้ได้อย่างถูกต้อง

- แอคเซสพอยต์ (access point)

  การที่เครือข่ายไร้สายจะสามารถติดต่อกับเครือข่ายอื่นๆ ได้นั้น ต้องทำการแปลงเฟรมข้อมูลในรูปแบบ 802.11 ให้กลายเป็นเฟรมข้อมูลรูปแบบอื่นที่เหมาะสมในการสื่อสารกับเครือข่ายนั้นๆ โดย แอคเซสพอยต์จะมีหน้าที่หลักในการแปลงเฟรมข้อมูลนี้

- ตัวกลางไร้สาย (Wireless medium)

  IEEE 802.11 สามารถรองรับระดับชั้นฟิสิคัล (Physical layer) ได้หลายรูปแบบแต่ที่รู้จักและนิยมโดยทั่วไปคือ คลื่นความถี่วิทยุ (RF)

- สถานี (station)

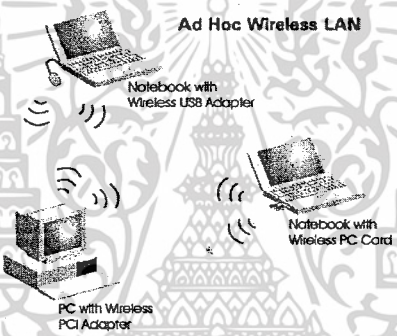
  อุปกรณ์ลูกข่ายโดยจะต้องติดตั้งส่วนเชื่อมต่อกับเครือข่ายไร้สาย

2.3.1 ชนิดของเครือข่ายไร้สาย

IEEE 802.11 ได้มีการกำหนดรูปแบบพื้นฐานของการเชื่อมต่อคือ Basic service set (BSS) ซึ่งเป็นการติดต่อสื่อสารระหว่างสถานี (station) โดยการที่สถานี (station) ที่อยู่ภายใน BSS เดียวกันจะสามารถติดต่อสื่อสารกันได้นั้นมี 3 รูปแบบคือ

2.3.1.1. Independent network (IBSS)

สถานีต่างๆที่อยู่ใน BSS เดียวกันจะสามารถติดต่อกันได้โดยตรงตราบเท่าที่สัญญาณจะไปถึง การเชื่อมต่อในลักษณะนี้จะนิยมใช้ในการเชื่อมต่อแบบชั่วคราว เช่นในการประชุม จะทำการเชื่อมต่อสถานีต่างๆเข้าด้วยกันเพื่อใช้ในการแลกเปลี่ยนข้อมูล เมื่อการประชุมเสร็จสิ้นเครือข่ายนั้นก็จะถูกยกเลิก โดยอาจจะเรียกการเชื่อมต่อแบบนี้ว่าเครือข่ายแอดฮ็อค (ad hoc network)



รูปที่ 2.7 เครือข่ายแอดฮ็อก

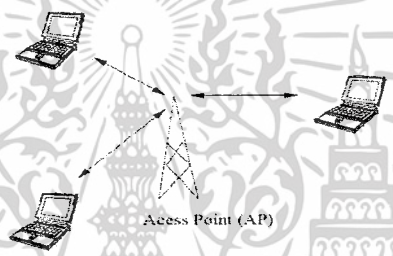
2.3.1.2. Infrastructure network

การเชื่อมต่อแบบนี้จะนำแอคเซสพอยต์เข้ามาเป็นตัวกลางในการติดต่อสื่อสารระหว่างสถานี แม้ว่าสถานีที่ติดต่อสื่อสารกันจะอยู่ใน BSS เดียวกันก็ตาม เช่น สถานี A ต้องการส่งข้อมูลไปให้สถานี B (โดยทั้งสถานี A และสถานี B อยู่ใน BSS เดียวกัน) สามารถทำได้โดย

- A จะส่งข้อมูลไปให้แอคเซสพอยต์
- แอคเซสพอยต์จะส่งข้อมูลที่ได้รับมาจาก A นั้นไปให้ สถานีปลายทางซึ่งก็คือ B

แม้ว่าการเชื่อมต่อแบบนี้จะต้องทำการส่งข้อมูลหลายๆทอดกว่าจะถึงผู้รับและมีการจราจร (traffic) ที่เกิดขึ้นในเครือข่ายมากกว่าการเชื่อมต่อแบบแอดฮ็อก (ad hoc) แต่ถึงอย่างไรการเชื่อมต่อแบบนี้มีข้อดีกว่าการเชื่อมต่อแบบแอดฮ็อก คือ

- การเชื่อมต่อแบบ infrastructure BSS จะถูกกำหนดด้วยระยะทางระหว่างสถานีถึงแอคเซสพอยต์ โดยสถานีทุกตัวต้องอยู่ในรัศมีสัญญาณของแอคเซสพอยต์ แต่ไม่ได้มีการกำหนดระยะทางระหว่างสถานีแต่ละตัวว่าห้ามเกินเท่าไหร่ ทำให้เป็นการประหยัดต้นทุนในเรื่องของความซับซ้อนของชั้นฟิสิคัล ในการจัดการความสัมพันธ์ที่เกิดขึ้นในแต่ละสถานีระหว่างตัวมันเองกับสถานีอื่นๆทั้งหมดที่อยู่ใน BSS เดียวกับมัน
- ตำแหน่งของการติดตั้ง แอคเซสพอยต์ให้ใกล้กับ สถานีจะทำให้สถานีประหยัดพลังงานที่ต้องใช้ในการเชื่อมต่อกับเครือข่ายเพราะส่วนใหญ่ สถานีในเครือข่ายไร้สายจะใช้พลังงานจากแบ็ตเตอรี่เป็นหลักเช่น Laptop, PDA เป็นต้น

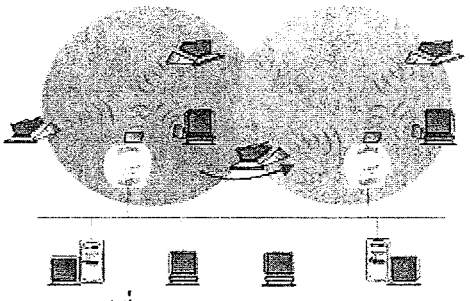


รูปที่ 2.8 Infrastructure networks

2.3.1.3. Extended service areas

Extended service areas(ESS) คือการเชื่อมต่อ BSS หลายๆ BSS เข้าด้วยกันเพื่อให้กลายเป็นเครือข่ายขนาดใหญ่ครอบคลุมพื้นที่การใช้งานที่กว้างขึ้น

สถานีต่างๆที่อยู่ใน ESS เดียวกันจะสามารถติดต่อสื่อสารกันได้แม้ว่าแต่ละสถานีจะอยู่คนละ BSS หรืออยู่ในขณะที่กำลังเปลี่ยนจาก BSS หนึ่งไปเป็นอีก BSS หนึ่งก็ตาม การที่สถานีต่างๆใน ESS จะสามารถติดต่อสื่อสารกันได้นั้น ตัวกลางไร้สายจะต้องทำงานเสมือนการติดต่อในระดับดาต้าลิงค์โดยมี แอคเซสพอยต์เปรียบเสมือนบริดจ์ (bridge) ที่คอยเชื่อมต่อระหว่างสถานีต่างๆใน ESS โดย แอคเซสพอยต์ในแต่ละ BSS จะเชื่อมต่อกับฮับ (hub) หรือสวิทช์ (switch) ของเครือข่าย



รูปที่ 2.9 Extended service areas

2.3.2 การทำงานของเครือข่าย 802.11

บริการในเครือข่าย

802.11 ได้กำหนดให้เครือข่ายไร้สายมีบริการ 9 รูปแบบ โดยเป็นบริการที่ใช้ในการส่งข้อมูล 3 บริการและอีก 6 บริการที่เหลือจะใช้ในการจัดการเครือข่าย

- Distribution

  เป็นบริการในการส่งเฟรมข้อมูลที่แอคเซสพอยต์ได้รับมาให้ไปถึง สถานีปลายทางที่ถูกต้อง การติดต่อสื่อสารที่กระทำผ่านแอคเซสพอยต์ทุกครั้งต้องใช้บริการนี้แม้ว่าทั้งผู้รับและผู้ส่งจะอยู่ในความดูแลของแอคเซสพอยต์ตัวเดียวกันก็ตาม

- Integration

  เป็นบริการที่ทำให้ระบบกระจาย (Distribution system) สามารถติดต่อสื่อสารกับเครือข่ายอื่นที่ไม่ใช่ IEEE802.11 ได้

- Association

  การที่สถานีจะสามารถใช้งานเครือข่ายไร้สายได้นั้น จะต้องทำการลงทะเบียนกับแอคเซสพอยต์ก่อน เพื่อให้ระบบกระจายทราบว่าในแต่ละแอคเซสพอยต์นั้นมีสถานีใดอยู่ในความดูแลของมันบ้างเพื่อที่จะสามารถส่งเฟรมไปให้ได้อย่างถูกต้อง

- Reassociation

  เมื่อสถานีได้ย้ายจาก BSS หนึ่งไปสู่ BSS หนึ่งที่อยู่ใน ESS เดียวกัน สถานีจะต้องทำการประเมินความแรงของสัญญาณที่ใช้ในการติดต่อกับแอคเซสพอยต์ ระหว่างความแรงสัญญาณของ BSS เก่ากับความแรงสัญญาณใน BSS ใหม่ หากพบว่าความแรงของสัญญาณของ แอคเซสพอยต์ ใน BSS ใหม่นั้นดีกว่า บริการ Reassociation ก็จะทำการเปลี่ยนแอคเซสพอยต์ที่ดูแลสถานีนั้นไปเป็นแอคเซสพอยต์ใน BSS ใหม่ จากนั้นระบบกระจายจึงทำการอัพเดทข้อมูลว่าสถานีนั้นได้อยู่ในความดูแลของ

แอคเซสพอยต์ตัวใหม่แล้ว

- Disassociation

  เป็นบริการที่ใช้ในการยกเลิกการเชื่อมต่อกับเครือข่ายไร้สาย เมื่อสถานีได้ทำการยกเลิกการเชื่อมต่อ ข้อมูลทั้งหมดเกี่ยวกับสถานีนั้นที่อยู่ในระบบกระจายจะถูกลบออกทั้งหมด

- Authentication

  เป็นบริการที่ใช้ในการยืนยันการมีสิทธิ์เข้าใช้เครือข่ายไร้สาย โดยบริการการยืนยันสิทธิ์นี้ (Authentication service) นี้จะถูกเรียกใช้ก่อนบริการ association เนื่องจากระบบจะอนุญาตให้เฉพาะผู้ที่ได้รับอนุญาตเท่านั้นที่จะสามารถใช้งานเครือข่ายนั้นได้

- Disauthentication

  เป็นบริการที่ใช้ในการยกเลิกการยืนยันสิทธ์

- MSDU delivery

  เป็นบริการที่ใช้ในการส่งข้อมูลให้ถึงผู้รับปลายทาง

- Privacy

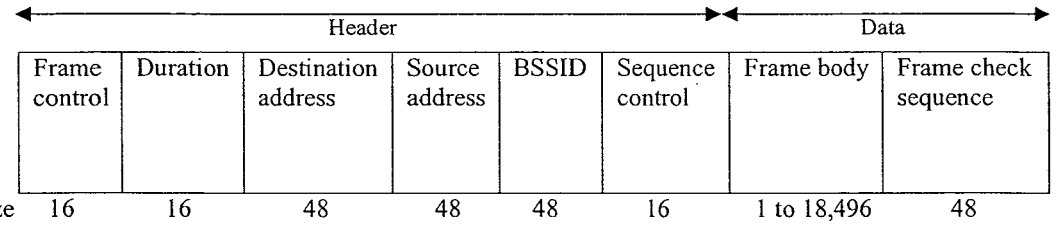
  เนื่องจากการที่เครือข่ายไร้สายสามารถถูกโจมตีได้ง่ายกว่าเครือข่ายแบบมีสาย IEEE

802.11 จึงได้มีการกำหนดนโยบายในการให้บริการด้านความปลอกภัยขึ้นมาซึ่งก็คือ การเข้าและถอดรหัสข้อมูล (WEP Encryption/Decryption) โดย WEP ใช้หลักการในการเข้าและถอดรหัสข้อมูลที่เป็นแบบสมมาตร (symmetrical) นั่นคือรหัสที่ใช้ในการเข้ารหัสข้อมูลจะเป็นตัวเดียวกันกับรหัสที่ใช้สำหรับการถอดรหัสข้อมูล และเมื่อไม่นานมานี้ ได้มีการนำเทคโนโลยีการรักษาความปลอดภัยแบบใหม่มาใช้ คือ WPA(Wi-Fi Protected Access) ซึ่งจะเป็นการใช้คีย์แบบไดนามิกในการเข้ารหัสข้อมูล คือคีย์จะเปลี่ยนไปเรื่อยๆทำให้ยากต่อการคาดเดา ซึ่งเมื่อเทียบกับคีย์แบบสแตติกกับแบบ WEP นั้นจะทำให้การตรวจสอบเป็นไปอย่างมีประสิทธิภาพมากขึ้น

2.3.3. MAC Frame Formats

มาตรฐาน IEEE 802.11 ได้กำหนดชนิดของเฟรมไว้ 3 ชนิดคือ management frames, control frames, and data frames

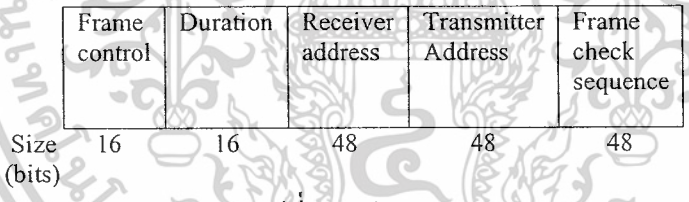
เมเนจเม็นต์เฟรม(Management frames) ใช้สำหรับกำหนดค่าเริ่มต้นในการติดต่อสื่อสารระหว่างอุปกรณ์และแอ็คเซสพอยต์(สำหรับโหมดอินฟราสตรักเจอร์) หรือระหว่างอุปกรณ์(สำหรับโหมดแอดฮ็อก) และใช้ในการรักษาการติดต่อสื่อสาร



รูปที่ 2.10 โครงสร้างของเมเนจเมนต์เฟรม

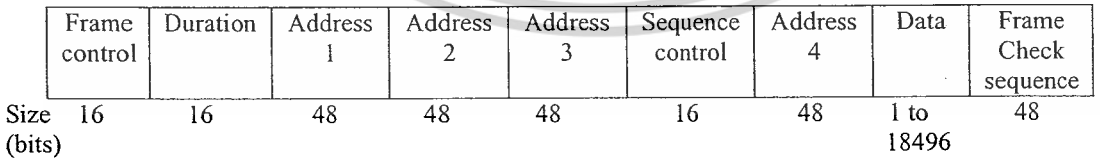
จากรูปแบบของเมเนจเม็นต์เฟรม ฟิลด์เฟรมคอนโทรล(Frame control) จะประกอบด้วยข้อมูลต่างๆ เช่น หมายเลขเวอร์ชันของมาตรฐาน ฟิลด์ระยะเวลา(Duration) จะประกอบด้วยตัวเลขที่ระบุระยะเวลาเป็นไมโครวินาทีที่จะต้องการส่งข้อมูล โดยตัวเลขนี้จะขึ้นอยู่โหมดที่ใช้ในการส่งข้อมูล ที่อยู่ปลายทางและที่อยู่ต้นทางคือที่อยู่ของอุปกรณ์ผู้รับและอุปกรณ์ผู้ส่งตามลำดับ บีเอสเอสไอดี(BSSID) จะเก็บที่อยู่ของบีเอสเอสไอดี ฟิลด์ควบคุมลำดับ(Sequence control) จะเป็นฟิลด์ที่ใช้เก็บเลขลำดับสำหรับแพ็กเก็ตและหมายเลขการแตกแพ็กเก็ต(Fragment number)

เฟรมควบคุม(Control Frame) โดยหลังจากที่การเชื่อมต่อระหว่างอุปกรณ์ไร้สายถูกสร้างขึ้นเรียบร้อยแล้ว เฟรมควบคุมจะเป็นตัวช่วยในการส่งเฟรมข้อมูล



รูปที่ 2.1 เฟรมควบคุม

เฟรมข้อมูล(Data frame) จะใช้ในการขนส่งข้อมูลที่ต้องการจะส่งไปให้อุปกรณ์ปลายทาง



รูปที่ 2.12เฟรมข้อมูล

จากรูป ฟิลด์

- Address 1 คือ แมคแอดเดรสของอุปกรณ์ที่จะรับเฟรมนี้
- Address 2 คือ แมคแอดเดรสของอุปกรณ์ที่ทำการส่งเฟรมนี้

- Address 3 คือ แมคแอดเดรสของเราท์เตอร์อินเตอร์เฟส ที่ BSS ของเราเชื่อมต่ออยู่ (ในกรณีที่เป็นการเชื่อมต่อในลักษณะเครือข่าย ESS)
- Address 4 จะใช้ในเครือข่ายแอดฮอค

2.3.4 การค้นหาเครือข่ายไร้สาย(Discovering the WLAN)

หน้าที่หลักอันดับหนึ่งของแมคเลเยอร์(MAC layer) จะเกี่ยวข้องกับกระบวนการค้นหาที่ทำให้อุปกรณ์ไร้สายสามารถค้นหาเครือข่ายไร้สายได้ จะเกี่ยวข้องกับกระบวนการที่ทำอุปกรณ์ไร้สายสามารถค้นหาเครือข่ายไร้สายที่มีอยู่ได้ โดยการค้นหานี้จะมีอยู่ 2 กระบวนการ คือ การที่แอ็คเซสพอยต์หรืออุปกรณ์ไร้สายอื่นๆ ต้องส่งเฟรมที่เหมาะสม และการที่อุปกรณ์ไร้สายต้องมองหาเฟรมนั้นๆ ซึ่งทั้งสองกระบวนการดังกล่าวมีรายละเอียดดังนี้

2.3.4.1 Beaconing

โดยปกติแอ็คเซสพอยต์ในเครือข่ายอินฟราสตรักเจอร์หรืออุปกรณ์ไร้สายในในเครือข่ายแอดฮ็อคส่งบีคอนเฟรม(Beacon frame) ออกมาเป็นระยะๆ เพื่อเป็นการแนะนำตัวมันเองว่ามีแอ็คเซสพอยต์ตัวนี้อยู่ และยังเป็นการส่งข้อมูลที่จำเป็นต่างๆออกไปให้กับอุปกรณ์ไร้สายอื่นที่ต้องการจะเข้าใช้เครือข่ายไร้สายนั้น

บีคอนเฟรมเป็นเมเนจเม็นต์เฟรมชนิดหนึ่งโดยในส่วนของฟิลด์ที่อยู่ปลายทาง(Destination address) จะถูกกำหนดเป็นที่อยู่แบบบรอดแคส(Broadcast address) ซึ่งทำให้อุปกรณ์ไร้สายทุกตัวที่อยู่ในพื้นที่นั้นจะได้รับบีคอนเฟรมนั้นไปประมวลผล โดยเนื้อหาภายในบีคอนเฟรมมีดังนี้

- Beacon interval –จะบอกระยะเวลาระหว่างการส่งบีคอน
- Timestamp – เป็นค่าที่บังคับให้อุปกรณ์ไร้สายทุกตัวปรับนาฬิกาของตนเองให้ตรงกับแอ็คเซสพอยต์
- Service Set Identifier (SSID) –ระบุหมายเลข SSID ของเครือข่ายไร้สายนั้น
- Supported rate – จะเป็นการบอกอัตราการส่งข้อมูลที่เครือข่ายไร้สายนั้นสนับสนุน
- Parameter sets – จะเป็นข้อมูลเกี่ยวกับหลักการมอดูลเลชั่นที่เครือข่ายไร้สายนั้นใช้อยู่
- Capability information - จะเป็นการบอกข้อมูลที่จำเป็นให้แก่อุปกรณ์ไร้สายหากอุปกรณ์นั้นต้องการเชื่อมต่อกับเครือข่ายไร้สายนี้

2.3.4.2 Scanning

การสแกนมี 2 แบบคือ แพสซิฟสแกนนิ่ง(passive scanning) และแอ็กทิฟสแกนนิ่ง (active scanning)

- Passive scanningในการสแกนแบบแพสซิฟนั้นอุปกรณ์ไร้สายจะรอฟังบีคอนเฟรม เมื่ออุปกรณ์ได้รับบีคอนเฟรมและ SSID มันจึงสามารถเข้าร่วมเครือข่ายนั้นได้ หากเป็นเครือข่าย ESS ซึ่งมีแอ็คเซสพอยต์หลายตัว อุปกรณ์ไร้สายจึงได้รับบีคอนเฟรมจากหลายๆแอ็คเซสพอยต์ ซึ่งในกรณีนี้อุปกรณ์ไร้สายจะพยายามเลือกเครือข่ายที่มีความแรงของสัญญาณมากที่สุด
- Active scanning ในการสแกนแบบแอ็กทิฟนั้นอุปกรณ์ไร้สายจะเป็นผู้ส่งเฟรมร้องขอออกไปยังทุกช่องทางที่พร้อมใช้งาน จากนั้นก็รอเฟรมตอบกลับจากทุกแอ็คเซสพอยต์ที่พร้อมใช้งาน ซึ่งในเฟรมตอบกลับนั้นจะมีข้อมูลที่จำเป็นสำหรับสถานีเพื่อใช้ในการเชื่อมต่อกับเครือข่ายไร้สาย

2.3.5 การเข้าร่วมเครือข่ายไร้สาย (Joining the WLAN)

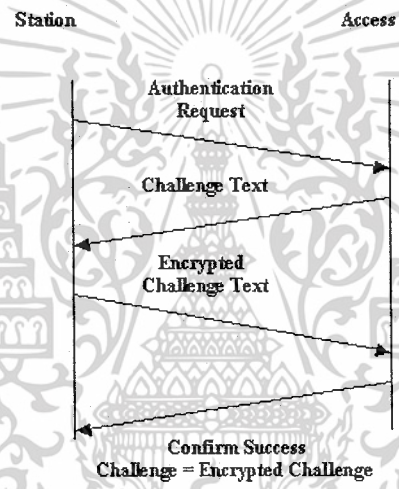
เมื่ออุปกรณ์ไร้สายค้นพบเครือข่ายไร้สายเรียบร้อยแล้ว อุปกรณ์ไร้สายต้องทำการร้องขอเพื่อเข้าร่วมเครือข่าย โดยการร้องขอเข้าร่วมเครือข่ายนี้จะมีสองส่วนคือการพิสูจน์ตัวจริง(authentication) และการเข้าใช้งาน(association)

2.3.5.1 การรับรองสิทธิ์ (Authentication)

กระบวนการในการรับรองสิทธิ์และการเข้าใช้งานในระบบเครือข่ายไร้สายตามมาตรฐานIEEE 802.11 นี้จะแบ่งออกเป็นสองรูปแบบ คือ

- ระบบเปิด(Open System Authentication) เป็นวิธีการรับรองสิทธิ์ที่พื้นฐานที่สุดโดยหลังจากที่อุปกรณ์ไร้สายค้นพบเครือข่ายไร้สายและได้รับข้อมูลที่จำเป็นแล้ว อุปกรณ์ไร้สายส่งเฟรมร้องขอเข้าร่วมเครือข่ายไปยังแอ็คเซสพอยต์ โดยเฟรมนี้ประกอบด้วยข้อมูลเกี่ยวกับดาต้าเรท(data rate) ที่อุปกรณ์นั้นสามารถสนับสนุนและ SSID ของเครือข่ายที่ต้องการจะเข้าร่วม เมื่อแอ็คเซสพอยต์ได้รับเฟรมร้องขอการเข้าร่วมเครือข่ายแล้วแอ็คเซสพอยต์จะทำการตัดสินใจโดยจะทำการเปรียบเทียบ SSID ที่ได้รับมากับ SSID ของเครือข่าย หากตรงกันก็จะทำการรับรองสิทธิ์ของอุปกรณ์ไร้สายนั้น โดยแอ็คเซสพอยต์จะทำการตอบกลับโดยการส่งเฟรมตอบกลับไปว่าจะยอบรับหรือไม่ยอมรับอุปกรณ์ไร้สายตัวนั้น

- การใช้กุญแจร่วม(Shared Key Authentication) การรับรองสิทธิ์แบบนี้จะมีการใช้งานกุญแจที่ใช้งานร่วมกันระหว่างแอ็คเซสพอยต์และอุปกรณ์ไร้สาย โดยอุปกรณ์ไร้สายจะทำการส่งเฟรมร้องขอการรับรองสิทธิ์ไปยังแอ็คเซสพอยต์ จากนั้นแอ็คเซสพอยต์จะทำการส่งเฟรมรับรองสิทธิ์ที่ประกอบด้วยข้อความที่เรียกว่า challenge text กลับมายังอุปกรณ์ไร้สาย อุปกรณ์ไร้สายจะทำการเข้ารหัสข้อความนั้นแล้วส่งกลับไปให้แอ็คเซสพอยต์ แอ็คเซสพอยต์ก็ทำการถอดรหัสข้อความที่ได้รับมาเพื่อดูว่าตรงกับข้อความต้นฉบับหรือไม่



รูปที่ 2.13 การรับรองสิทธิ์แบบใช้กุญแจร่วม

2.3.5.2 การสร้างสัมพันธ์(Association)

เมื่ออุปกรณ์ไร้สายได้รับการรับรองสิทธิ์เรียบร้อยแล้ว ขั้นตอนสุดท้ายคือการได้รับการยอมรับให้เข้าใช้เครือข่ายไร้สายโดยเรียกว่า association ในขั้นตอนของการรับรองสิทธิ์หากแอ็คเซสพอยต์ยอมรับอุปกรณ์ไร้สายนั้นแอ็คเซสพอยต์จะทำการจองพื้นที่หน่วยความจำและทำการกำหนดหมายเลขความสัมพันธ์นั้น

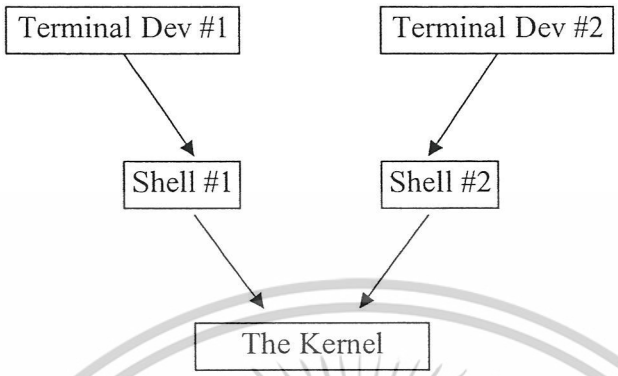
2.4 ลีนุกซ์

ในเครือของระบบปฏิบัติการตระกูลลีนุกซ์นั้นมีให้เลือกใช้งานได้อย่างหลายหลาย โดยแต่ละระบบปฏิบัติการก็มีความสามารถหรือจุดเด่นที่แตกต่างกันไป แม้ว่าจะอยู่ในตระกูลลีนุกซ์เหมือนกันก็ตาม แต่ก็มีความแตกต่างกันให้ความสามารถที่แตกต่างกันอยู่บ้าง เช่น การให้บริการเป็นเซิร์ฟเวอร์ในงานต่างๆ หรือแอพพลิเคชั่นที่หลากหลาย เป็นต้น นอกจากนี้โดยส่วนมากเครื่องมือต่างๆ ของลีนุกซ์มักไม่ได้อยู่ในรูปแบบการติดต่อแบบกราฟิก เช่น ลีนุกซ์ที่มีขนาดเท่ากับแผ่นดิสก์ขนาด 1.44 นิ้ว ก็อาจสามารถที่จะใช้เป็นเราท์เตอร์ เป็นต้น แต่อย่างไรก็ดีในระบบปฏิบัติการลีนุกซ์ส่วนมากจะมีการติดต่อกับผู้ใช้แบบกราฟิกอยู่แล้วโดยเรียกการติดต่อรูปแบบนี้ว่า “X Windows” (รายละเอียดศึกษาเพิ่มเติมได้ http://www.XFree86.org) โดยภายใน X windows จะประกอบไปด้วยส่วนของ ตัวจัดการวินโดวส์ และสิ่งแวดล้อมในการทำงานของระบบปฏิบัติการ ซึ่งชนิดหรือประเภทส่วนประกอบทั้งสองตัวนี้อาจจะแตกต่างกันได้ในลีนุกซ์ต่างชนิดกัน ซึ่งในส่วนของสิ่งแวดล้อมในการทำงาน ที่เป็นที่นิยมมีด้วยกันอยู่สองตัวคือ GNOME และ KDE โดยเมื่อลงลีนุกซ์ จะสามารถเลือกว่าจะเอาสิ่งแวดล้อมในการทำงานดังกล่าวว่าจะใช้ชนิดใด

ความแตกต่างของลีนุกซ์นี้ เพราะฉะนั้นในการใช้งานควรเลือกใช้ลีนุกซ์ให้เหมาะสมกับงานที่ต้องใช้ แต่ก็สามารถเลือกแพ็กเก็ตเพิ่มเติมเพื่อให้ความสามารถที่ไม่มีของลีนุกซ์บางตัว มีความสามารถนั้นได้เหมือนกันโดยผ่าน “ตัวจัดการแพ็กเกจ” หรือ “package manager” โดยตัวจัดการแพ็กเกจที่ได้รับความนิยมคือ “Red Hat Package- Manager: RPM” ซึ่งระบบปฏิบัติการลีนุกซ์หลายตัวก็ใช้ตัวนี้เช่นกัน นอกจากนี้ก็สามารถไปดาวน์โหลดโปรแกรมต่างๆ มาใช้ได้ โดยโปรแกรมดังกล่าวจะอยู่ในรูปแบบ “tarball format” ซึ่งเป็นรูปแบบในการบีบอัดไฟล์แบบหนึ่ง ซึ่งจะมีสคริปต์ที่ใช้ในการติดตั้งโปรแกรมปนอยู่ด้วย ซึ่งวิธีนี้ไม่ต้องผ่านตัวจัดการแพ็กเก็ต แต่ในการจัดการลบโปรแกรมเหล่านี้ออกก็จะสามารถทำได้ยากตามไปด้วยเช่นกัน

ในการใช้งานลีนุกซ์นั้นมีความหลากหลายไม่แพ้กับระบบปฏิบัติการอื่นๆ เลยทีเดียว โดยการทำงานก็มีทั้งในระดับใช้งานทั่วๆไป และการใช้งานแบบให้บริการ(server) ตัวอย่างการให้บริการเช่น Routing , FTP, Scientific station, Desktop Publishing Software เป็นต้น

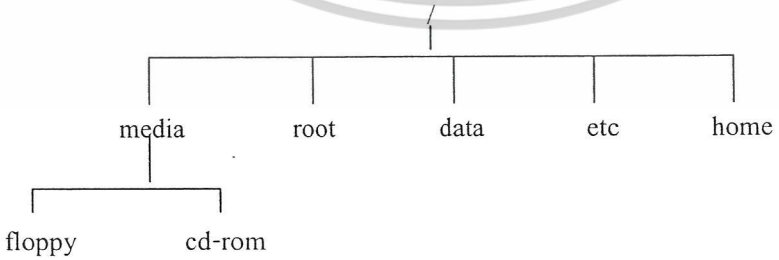
2.4.1 การใช้งาน



รูปที่ 2.14 การใช้งานบนลีนุกซ์

ในการทำงานต่างๆ จะดำเนินการผ่านซอฟท์แวร์ ซึ่งหากงานนั้นต้องอาศัยการบริการจากฮาร์ดแวร์ ตัวซอฟท์แวร์ก็ต้องทำการร้องขอให้ฮาร์ดแวร์ทำงานตามที่ต้องการได้ โดยในทุกระบบปฏิบัติการประกอบขึ้นด้วยจากการทำงานหลายๆ ส่วน โดยจะมีส่วนประกอบหลักส่วนหนึ่งที่ทำหน้าที่ในการเรียกส่วนประกอบอื่นๆ ทั้งหมด ซึ่งเรียกว่า เคอร์เนล(kernel) เมื่อผู้ใช้ตอบโต้กับเครื่องคอมพิวเตอร์นั้นก็เปรียบเสมือนเป็นการตอบโต้กับเคอร์เนลนั่นเอง เพียงแต่การโต้ตอบกับเคอร์เนลนั้นไม่สามารถโต้ตอบได้โดยตรง แต่ต้องผ่านช่องทางของผู้ใช้เท่านั้น ซึ่งช่องทางดังกล่าวเรียกว่าเทอร์มินอล(terminal) ซึ่งลีนุกซ์รองรับการทำงานได้หลายๆ เทอร์มินอลพร้อมๆ กัน ซึ่งในการเข้าใช้แต่ละครั้งจะต้องมีอินเทอร์เฟสในการติดต่อกับเคอร์เนลซึ่งจะเรียกจุดที่ที่เชื่อมต่อกันนี้ว่า เปลือกระบบ(shell) โดยเปลือกระบบจะคอยทำหน้าที่ส่งข้อมูล(อินพุท)จากผู้ใช้ไปเป็นอินพุทของเคอร์เนลสำหรับการประมวลผลโดยปริยายแล้ว เปลือกระบบที่ใช้ในลีนุกซ์คือ BASH Shell (command line)

2.4.2 โครงสร้างของแฟ้มข้อมูล



รูปที่ 2.15 ตัวอย่างโครงสร้างไฟล์บนลีนุกซ์

ในลีนุกซ์โครงสร้างของไฟล์จะแตกต่างกับในวินโดวส์ซึ่งแบ่งฮาร์ดดิสก์ไปตามพาร์ทิชั่นซึ่งมีตัวอักษรกำกับไว้อยู่ แต่ในลีนุกซ์นั้นทุกแฟ้มข้อมูลทุกไฟล์จะต้องอยู่ภายใต้แฟ้มเดียวกันทั้งหมดคืออย่างจะต้องอยู่ภายใต้ รูท, root, (มักระบุโดย '/') โดยจะปิดบังความหลากหลายของพาร์ทิชั่นกับผู้ใช้ โดยในลีนุกซ์มีไดเรกทอรี่ที่สำคัญๆ ได้แก่

/dev เก็บไฟล์ของอุปกรณ์

/proc ใช้ในการทำงานของระบบปฏิบัติการ

/etc เก็บไฟล์คอนฟิกต่างๆ

/sbin ไบนารีของระบบ

/bin ไบนารีของระบบเพิ่มเติม

/lib ไลบารีร่วมที่ใช้ในการทำงานของระบบ

/mnt เป็นที่รวมเมาท์พอยท์ของอุปกรณ์ต่อพ่วง

/usr ใช้สำหรับเก็บโปรแกรมหรือเครื่องมือต่างๆ ของผู้ใช้งาน

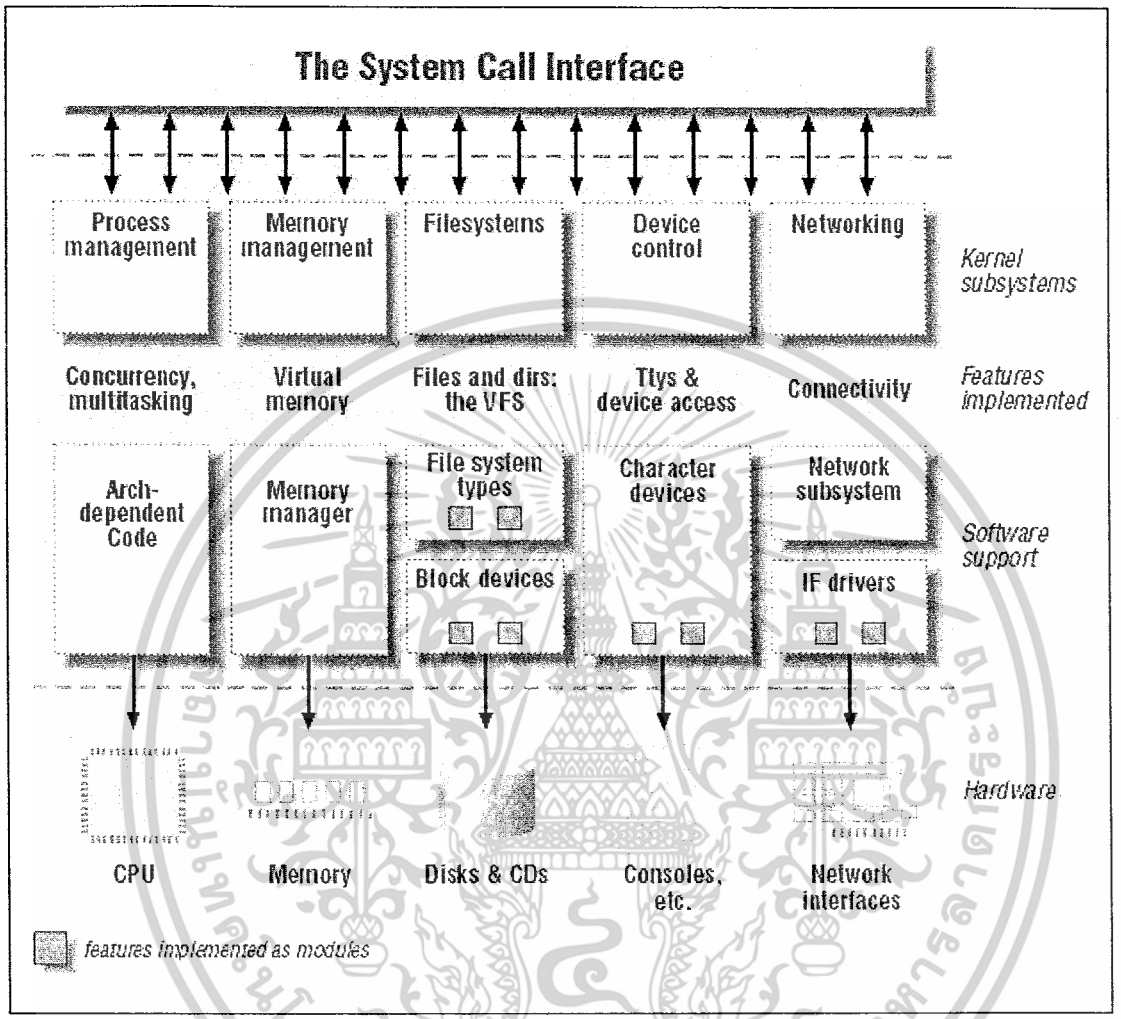
2.4.3 BASH Shell

โดยทั่วไป ในการทำงานของ BASH shell จะประกอบไปด้วยสามส่วนคือ stdin, stdout และ stderr โดยในแต่ละตัวก็คือ คำสั่ง, ผลลัพธ์จากคำสั่ง และ ความผิดพลาดที่เกิดขึ้นจากคำสั่ง

ในการใช้ BASH shell สามารถใช้ redirect stdio and stderr จากหน้าของเทอร์มินอล ให้ไปเก็บไว้ในไฟล์ๆ นึงโดยใช้สัญลักษณ์ '>' นั่นเอง ตามด้วยชื่อไฟล์หรือที่อยู่ของไฟล์นั่นเองโดยหน้าเครื่องหมาย '>' จะมีเลขคอยกำกับว่าจะเก็บ stdin(1) หรือ stderr(2) ก็ได้ เช่น "ls /etc/hosts /etc/h 1>goodoutput" ผลที่ได้คือไฟล์ goodoutput จะมีข้อความจากคำสั่ง ls ให้ แต่หากพิมพ์คำสั่ง "ls /etc/hosts /etc/h 2 > badoutput" ผลที่ได้จะพบว่าใน badoutput เป็นคำแสดงความผิดพลาดการจากการเรียกดูไฟล์หรือแฟ้มเอกสาร

นอกจากนี้สามารถที่จะส่งค่าที่ได้จากคำสั่งหนึ่งไปประมวลผลต่อกับอีกคำสั่งหนึ่งได้โดยใช้สัญลักษณ์ '|' โดยการใช้สัญลักษณ์นี้มีรูปแบบคือ command1 | command2 | command3 หมายความว่าคำสั่งที่ 1 เมื่อประมวลผลเสร็จจะนำข้อมูลที่ได้ (stdout) ไปเป็นข้อมูลเริ่มต้น (stdin) ให้กับคำสั่งที่สอง และเช่นเดียวกันคือผลที่ได้จากคำสั่งที่สองซึ่งใช้ข้อมูลจากการประมวลผลคำสั่งแรก จะถูกส่งต่อไปให้คำสั่งที่สามประมวลผลต่อไปและแสดงค่าที่ได้ออกสู่หน้าจอ ให้ผู้ใช้ได้เห็น

2.4.4 การติดต่อระหว่างอุปกรณ์กับระบบปฏิบัติการลีนุกซ์



รูปที่ 2.16 โครงสร้างการทำงานในระดับเคอร์แนล

โพรเซสทุกโพรเซสนอกจากจะต้องเกี่ยวข้องกับหน่วยประมวลผลกลางกับหน่วยความจำแล้ว บางครั้งยังต้องมีการติดต่อกับอุปกรณ์อื่นๆ ด้วย เพื่อให้การทำงานสำเร็จ โดยโพรเซสหากต้องการติดต่อกับอุปกรณ์หรือฮาร์ดแวร์ใด โพรเซสจะต้องติดต่อตามช่องทางที่ระบบปฏิบัติการได้มีไว้ให้นั่นก็คือ system call จากนั้นระบบปฏิบัติการจะไปทำการเรียกเคอร์แนลเพื่อให้มารับคำร้องขอการใช้งานของโพรเซสต่อไป ซึ่งในการร้องขอใช้ทรัพยากรแต่ละตัวก็จะมีเคอร์แนลมารองรับแตกต่างกันไป โดยแยกได้ดังนี้

- Process Management เป็นส่วนที่ใช้สำหรับจัดการโพรเซสต่างๆ มีหน้าที่ในการสร้างหรือทำลายโพรเซส รับผิดชอบเกี่ยวกับการติดต่อระหว่างโพรเซสหรือกับ

อุปกรณ์รับข้อมูล/แสดงข้อมูล(I/O device)และรับผิดชอบการจัดการตารางการใช้งานหน่วยประมวลผลกลางของแต่ละโพรเซสอีกด้วย

- Memory Management เป็นส่วนที่จัดการเกี่ยวกับ การใช้งานหน่วยความจำ รับผิดชอบการสร้างหน่วยความจำเสมือน เพื่อรองรับโพรเซสบางส่วนที่ไม่สามารถถูกจัดเก็บไว้ในหน่วยความจำหลักได้เพียงพอ
- File systems เนื่องจากระบบปฏิบัติการลีนุกซ์ (ซึ่งมีพื้นฐานจาก ระบบปฏิบัติการยูนิกซ์) มักมองทุกสิ่งอยู่ในรูปของไฟล์ทำให้ขนาดงานของส่วนนี้ จะมีขนาดใหญ่ โดยหน้าที่หลักๆ ของเคอร์เนลในส่วนนี้คือ ต้องทำการสร้างระบบไฟล์บนอุปกรณ์ที่ไม่มีโครงสร้างให้เกิดมีโครงสร้างที่ลีนุกซ์เข้าใจได้
- Device Control การทำงานทุกอย่างของระบบจะต้องเกิดขึ้นที่อุปกรณ์ต่างๆ ซึ่งอุปกรณ์เหล่านี้จะถูกควบคุมด้วย "โค้ดโปรแกรม" ซึ่งนั่นคือ ไดร์เวอร์ (device driver) ซึ่งโค้ดคำสั่งต่างๆ ก็แตกต่างกันไปขึ้นอยู่กับอุปกรณ์ แม้กระทั่งอุปกรณ์ประเภทเดียวกันและยี่ห้อเดียวกัน ก็อาจมีไดร์เวอร์ที่แตกต่างกันซึ่งอาจจะให้ฟังก์ชันการทำงานที่แตกต่างกันตามไปด้วย
- Networking รับผิดชอบการทำงานเกี่ยวกับเครือข่าย โดยมักจะถูกจัดการโดยระบบปฏิบัติการ อินเทอร์เฟสหรืออุปกรณ์เครือข่ายจะทำการส่งข้อมูลและรับข้อมูลเท่านั้นโดยตัวอุปกรณ์เองจะไม่สนใจว่า ข้อมูลแพ็กเก็ตดังกล่าวนั้น เป็นของโพรเซสใด เนื่องจากแพ็กเก็ตมักจะถูกส่งมาอย่างไม่เกี่ยวข้องกัน หรือไม่ต่อเนื่องกัน โดยอินเทอร์เฟสจะทำการเก็บแพ็กเก็ต, ระบุ และถอดข้อความออกก่อนที่จะส่งไปให้โพรเซสต่อไป โดยระบบจะคอยทำหน้าที่ในการจัดการส่งข้อมูลระหว่างโปรแกรมกับอุปกรณ์เครือข่าย นอกจากนั้นเคอร์เนลในส่วนนี้จะต้องจัดการในส่วนงานจำพวก การสร้างเส้นทางเดินข้อมูล(routing), การคำนวนหาตำแหน่ง(address resolution) ด้วย

ข้อดีของระบบปฏิบัติการลีนุกซ์ประการหนึ่งคือ ขณะระบบปฏิบัติการทำงานอยู่ ก็สามารถทำการเพิ่มฟีเจอร์ต่างๆให้กับลีนุกซ์ได้ โดยส่วนของโค้ดคำสั่งที่เพิ่มเข้าไปโดยจะเรียกว่า 'module' โดยแต่ละโมดูลจะต้องเป็น object code เท่านั้น โดยในการทำงานของโมดูลจะแยกตามชนิดของอุปกรณ์โดยในลีนุกซ์แบ่งอุปกรณ์ออกเป็น 3 ประเภทคือ

- Character Device -- > อุปกรณ์ชนิดนี้จะทำการส่งข้อมูลอยู่ได้ทีละ1 ไบท์เท่านั้น โดยคำสั่งพื้นฐานที่มักใช้คือ read, write, open และ close ในการติดต่อกับอุปกรณ์ประเภทนี้ต้องทำการติดต่อผ่านไฟล์โหนด โดยอุปกรณ์ประเภทนี้มักเป็นอุปกรณ์ที่ใช้ในการส่งรับข้อมูลเป็นลำดับ (sequential)
- Block Device -- > อุปกรณ์ชนิดนี้ทำการส่งข้อมูลทีละหลายๆ ไบท์ โดยลักษณะของอุปกรณ์ประเภทนี้คือเป็นอุปกรณ์ที่มีระบบไฟล์เป็นของตนเอง โดยการใช้งานจะต้องมี mount point จำพวกเช่น ฮาร์ดดิสก์ ซีดีรอม เป็นต้น
- Network Interface -- > การทำงานเครือข่ายจะต้องผ่านอินเทอร์เฟส ซึ่งจะมีหน้าที่ในการส่งข้อมูลตามที่เคอร์แนลสั่งการมา แต่ไม่รู้ว่างานที่ทำนั้นเป็นของ transaction ใด การทำงานจะแตกต่างกับสองชนิดแรกคือจะไม่ยุ่งเกี่ยวกับไฟล์ เคอร์แนลจะทำการเรียกไดร์เวอร์โดยตรง

โมดูลจะทำหน้าที่คอยรับ คำร้องขอ(request) จากโพรเซสต่างๆ ซึ่งอาจเกิดขึ้นเมื่อไรก็ได้ เพราะฉะนั้นโมดูลจะมีความแตกต่างกับโปรแกรมในระดับของยูสเซอร์สเปซ(user-space) ทั่วๆ ไปคือจะไม่มีฟังก์ชัน main() ที่เป็นฟังก์ชันเมื่อหลักในการทำงาน และเนื่องจากโมดูลนั้นฝังตัวอยู่ในเคอร์แนลสเปซ(kernel-space) ไลบารีต่างๆ จะต้องนำมาจากสิ่งที่มีอยู่ในระดับเคอร์แนลเท่านั้นไม่สามารถเชื่อมโยง(include) ได้เหมือนปกติที่เขียนในระดับของ user-space

ตัวอย่างในการเขียนโมดูล(hello.c)

```
#include <linux/kernel.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");  //Display license
static int init_moudule(void){
   printk(This is hello Module\n");
   return 0;  }
static void cleanup_module(void){
   printk("Goodluck\n");  }
```

โดยในการเพิ่มโมดูลลงในเคอร์ในสามารถทำได้โดยใช้คำสั่ง insmod

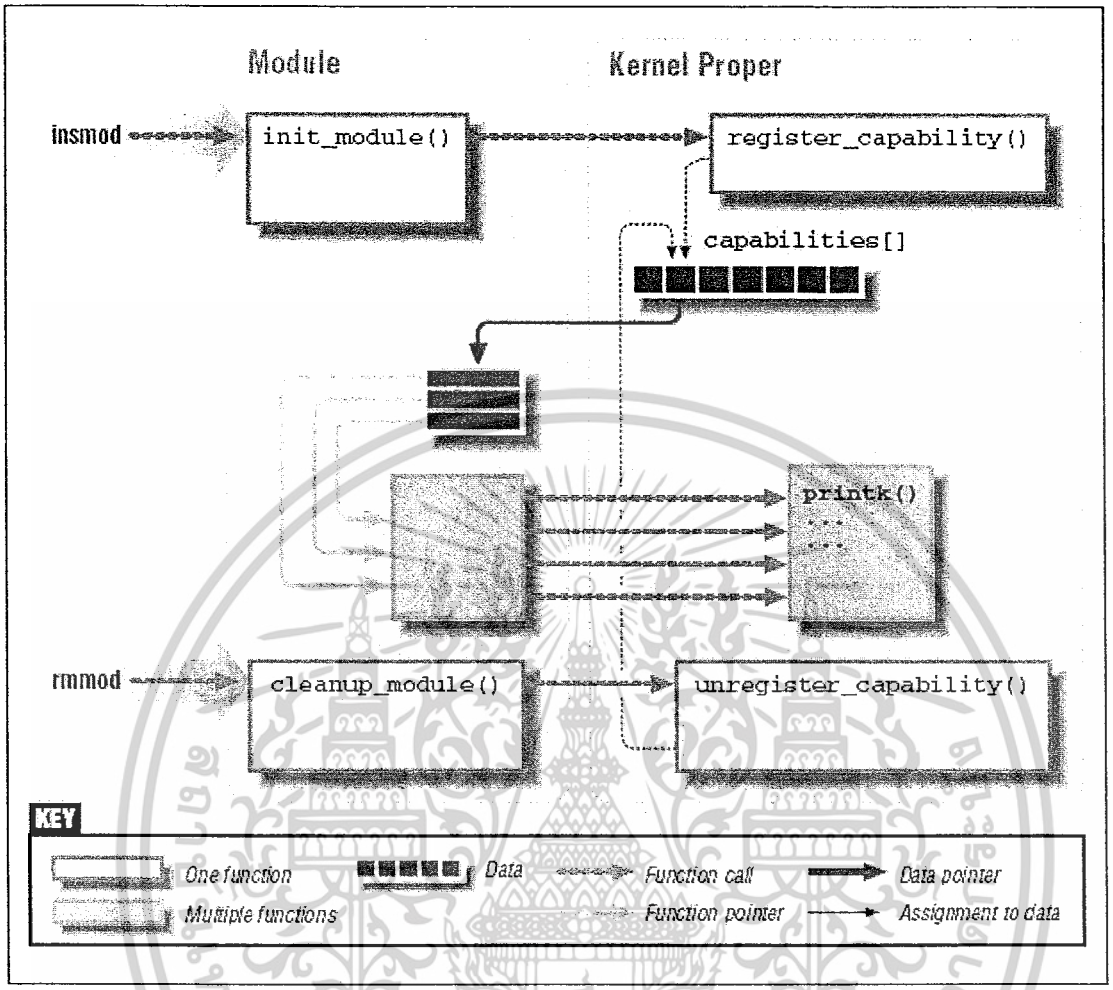
```
root# insmod ./hello.o
This is Hello Module
```

ผลลัพธ์ที่ได้จากการเพิ่มโมดูล hello (โค้ดตามด้านบน) คือมีข้อความว่า "This is Hello Module" ตามคำสั่งที่ได้เขียนไว้ในข้างต้น(หากไม่พบข้อความใดๆ ปรากฏ ให้ใช้คำสั่ง dmesg หรือดูข้อมูในไฟล์ /var/log/kern.log) โดยเมื่อโปรแกรมถูกเพิ่มเข้าไปในเคอร์แนล จะทำการเรียกฟังก์ชัน "int init_module(void)" ซึ่งภายในฟังก์ชันดังกล่าง มีการสั่งให้พิมพ์ "This is Hello Module" ดังกล่าวออกมาเช่นเดียวกันกับฟังก์ชัน "void cleanup_module(void)" ซึ่งจะถูกเรียกเมื่อทำการเอาโมดูลนี้ออกโดยใช้คำสั่ง rmmod โดยทั้ง init_module() และ cleanup_module() สามารถเปลี่ยนแปลงชื่อฟังก์ชันได้โดยต้องทำการเรียกฟังก์ชัน moudule_init(ชื่อฟังก์ชัน) , module_exit(ชื่อฟังก์ชัน)

```
root# rmmod hello
Goodluck
```

สังเกตว่าในโค้ดคำสั่งด้านบนจะไม่มีการใช้ไลบารีหรือฟังก์ชันที่พบในภาษาซี ที่ถูกเขียนในระดับของแอพพลิเคชั่นเลย เนื่องจากโค้ดดังกล่างนั้น ถูกเขียนเพื่อให้ทำงานในระดับของเคอร์แนล เพราะฉะนั้นไลบารีและคำสั่งต่างๆ ต้องเป็นของเคอร์แนลเท่านั้น แต่โดยทั่วไปลักษณะการใช้งานก็ยังคงเป็นภาษาซี อาจมีความแตกต่างกันบ้างเล็กน้อย โดยสามารถศึกษาไลบารีของเคอร์แนลในลีนุกซ์ได้ใน /usr/include/linux หรือ /usr/include/asm แต่โดยหลักๆ แล้วในการเขียนโมดูลต้องมีการใช้ไลบารี module.h

นอกจากนี้ในการสร้างโมดูลจะต้องมีการประกาศความสามารถของโมดูลด้วย เพื่อบ่งบอกว่าแอพพลิเคชั่นต่างๆ สามารถเรียกใช้การทำงานจากโมดูลนี้ มีอะไรบ้าง โดยในการประกาศจะต้องประกาศด้วยเป็นตัวแปรชนิดโครงสร้างและในการเรียกใช้งานโมดูลนั้นจะต้องถูกเรียกใช้งานผ่านไฟล์ซึ่งจะต้องสร้างโหนดขึ้นมาเพื่อรองรับการเรียกใช้งานจากระดับแอพพลิเคชั่น



รูปที่ 2.17 การเชื่อมโมดูลกับเคอร์เนล

2.5 การรับประกันคุณภาพ(Quality of Service)

ในระบบเครือข่ายทั่วไป ข้อมูลที่วิ่งอยู่ในระบบจะมีความหลากหลายของข้อมูลตามความต้องการของผู้ใช้ที่แตกต่างกัน ข้อมูลบางประเภทต้องการความรวดเร็วในการส่งที่ค่อนข้างสูงแต่ข้อมูลบางประเภทไม่ได้เน้นว่าต้องส่งรวดเร็ว เพียงแต่ต้องการให้ผู้รับได้รับข้อมูลที่ครบถ้วนสมบูรณ์

ระบบรับประกันคุณภาพ(Quality of Service) คือ ระบบที่จะช่วยเพิ่มประสิทธิภาพการรับส่งข้อมูลบนระบบเครือข่ายให้ดีขึ้น โดยเป็นไปตามเงื่อนไขที่ผู้ใช้ต้องการ ซึ่งเราอาจสามารถนิยามเงื่อนไขต่างๆได้ดังนี้

- การมีให้ใช้งานได้ (Availability) ซึ่งในทางอุดมคติต้องเป็น 100 เปอร์เซ็นต์ของเวลาการใช้งาน แต่ในความเป็นจริงไม่มีระบบเครือข่ายใดจะให้บริการได้ 100 เปอร์เซ็นต์

- ช่องสัญญาณที่ส่งได้ (Throughput) หมายถึงค่าจริงที่เราสามารถรับส่งข้อมูลจากปลายหนึ่งไปยังปลายอีกข้างหนึ่งได้ได้ด้วยอัตราเร็วเท่าไหร่ในหน่วยจำนวนบิตต่อวินาที
- การสูญหายของแพ็กเก็ต (Packet loss) หมายถึงจำนวนแพ็กเก็ตที่สูญหายระหว่างการส่ง
- ลาเทนซี่ (Latency) คือค่าเวลาในการเดินทางของแพ็กเก็ตจากต้นทางไปถึงปลายทาง
- เวลาจิตเตอร์ (Jitter) คือค่าการแปรปรวนของค่าลาเทนซี่ หมายถึงแพ็กเก็ตเคลื่อนที่จากต้นทางไปยังปลายทางหลายๆ แพ็กเก็ต ปรากฏว่าการไปถึงปลายทางใช้ระยะเวลาต่างกันทำให้ข้อมูลบางส่วนที่ไปก่อนอาจถึงทีหลัง หรือมีเวลาเหลื่อมกัน ทำให้ต้องมีการตรวจสอบลำดับของแพ็กเก็ตที่รับมาในฝั่งผู้รับด้วย

ตารางที่ 2.1 ข้อมูลบนอินเตอร์เน็ตที่มีความไวต่างๆ

ชนิดข้อมูล	ช่องสัญญาณที่ส่งได้	การสูญหายของแพ็กเก็ต	ลาเทนซี่	จิตเตอร์
เสียง	ต่ำมาก	ปานกลาง	สูง	สูง
พาณิชย์อิเล็กทรอนิกส์	ต่ำ	สูง	สูง	ต่ำ
ทรานแซกชั่น	ต่ำ	สูง	สูง	ต่ำ
อีเมล์	ต่ำ	สูง	ต่ำ	ต่ำ
telnet	ต่ำ	สูง	ปานกลาง	ต่ำ
การเรียกจากบราวเซอร์ทั่วไป	ต่ำ	ปานกลาง	ปานกลาง	ต่ำ
การโอนย้ายไฟล์	สูง	ปานกลาง	ต่ำ	ต่ำ
วีดีโอคอนเฟอเรนซ์	สูง	ปานกลาง	สูง	สูง

ดิสทริบิวชั่น โคออดิเนชั่น ฟังก์ชัน(Distribution Coordination Function) คือเทคนิคพื้นฐานในการเข้าใช้งานตัวกลางของมาตรฐาน IEEE 802.11 ซึ่งเทคนิคนี้จะมีการเข้าใช้งานแบบแย่งชิงตัวกลางกันคือใครเข้ามาก่อนก็ได้ส่งก่อน โดยดิสทริบิวชั่น โคออดิเนชั่น ฟังก์ชัน ได้ใช้อัลกอริธึมซีเอสเอ็มเอซีเอ (CSMA/CA) ในการควบคุมการใช้งานตัวกลางเพื่อป้องกันการชนกันของข้อมูล ซึ่งจากลักษณะการเข้าใช้งานตัวกลางแบบนี้ ต้องแย่งชิงตัวกลางกันใครมาก่อนได้ก่อน ทำให้ไม่เหมาะสมกับการใช้ส่งข้อมูลประเภทเรียลไทม์หรือข้อมูลประเภทมัลติมีเดีย เนื่องจากข้อมูลประเภทนี้มีขนาดของเฟรมที่สั้นแต่ต้องการความล่าช้า(delay) ในการส่งที่น้อย(ส่งให้เร็ว) ทำให้ในการใช้งานเครือข่ายไร้สายแบบไม่มีระบบรับประกันคุณภาพ หากมีข้อมูลประเภทมัลติมีเดียเข้ามาใช้งานตัวกลางต่อจากข้อมูลประเภททรานแซคชั่นหรือข้อมูลทั่วไป ซึ่งเป็นข้อมูลประเภทที่ต้องการความถูกต้องในการส่งมาก จึงทำให้ใช้

เวลาในการส่งที่นาน ส่งผลให้ข้อมูลประเภทมัลติมีเดีย อาจจะต้องรอใช้งานช่องสัญญาณเป็นระยะเวลายาวนาน ทำให้เกิดความล่าช้าในการส่งข้อมูลได้ ผู้รับก็จะได้รับข้อมูลที่ไม่ราบรื่นต่อเนื่องกัน เช่นภาพและเสียงอาจจะกระตุกมาก

การนำระบบรับประกันคุณภาพเข้ามาใช้ในระบบเครือข่ายไร้สาย จะเป็นการช่วยรับรองว่าข้อมูลแต่ละประเภทจะถูกจัดสรรการส่งอย่างเหมาะสม ตามความต้องการใช้งาน ตัวอย่างเช่น ในเครือข่ายแลนไร้สายนั้น ให้ความสำคัญกับข้อมูลประเภทมัลติมีเดียมากกว่าข้อมูลประเภทอื่น ผู้ดูแลระบบก็สามารถทำการจัดการการใช้งานเครือข่ายแลนไร้สาย ให้มีการส่งข้อมูลประเภทมัลติมีเดียมากกว่าข้อมูลประเภทอื่นๆได้ ซึ่งจะเป็นการช่วยแก้ปัญหาความล่าช้าในการส่งได้ในระดับหนึ่ง

บทที่ 3
การสร้างไวร์เลสแอ็คเซสพอยท์

3.1 การติดตั้งระบบปฏิบัติการลีนุกซ์

โครงงานนี้ใช้ดีเบียนดิสทริบิวชั่น เนื่องจากเป็นดิสทริบิวชั่นที่ให้ประสิทธิภาพในการทำงานกับอุปกรณ์เครือข่ายไร้สายในการพัฒนาโครงงานนี้ ตามที่ผู้พัฒนาไดร์เวอร์ได้แนะนำ โดยสิ่งที่ต้องการในการติดตั้งระบบคือ แผ่นซีดีระบบปฏิบัติการลีนุกซ์ดีเบียน นอกจากนี้ยังต้องการการเชื่อมต่อกับอินเทอร์เน็ทเพื่อทำการโหลดและติดตั้งแพ็กเกจเพิ่มเติม โดยขั้นตอนการติดตั้งมีดังนี้

- ทำการแก้ค่าในไบออส ให้ทำการบูทเครื่องจากแผ่นซีดีรอม
- ทำการปรับแต่งพาร์ทิชันของฮาร์ดดิสก์ เพื่อให้เพียงพอและเหมาะสมตามแต่ต้องการในการใช้งาน โดยใช้คำสั่ง cfdisk หรือ fdisk ก็ได้ แต่ในการติดตั้งลีนุกซ์จะต้องแบ่งอย่างน้อยออกเป็น 2 พาร์ทิชั่นคือ linux native partition และ linux swap โดยพาร์ทิชั่นชนิดหลัง(ลีกนุกซ์สแวป) จะต้องมีพื้นที่อย่างน้อยเป็นสองเท่าของพื้นที่ของหน่วยความจำหลัก
- เลือกประเภทที่จะติดตั้งและทำการเลือก package หลักว่าต้องการอะไรบ้างจากนั้นระบบจะถามอีกว่าต้องการเลือกแพ็กเกจย่อยๆ ด้วยตนเองอีกหรือไม่ ซึ่งหากเราต้องการให้ดีเบียนทำงานในส่วนใดก็ให้เลือกแพ็กเกจที่ต้องการได้
- ระบบจะทำการติดตั้งและโหลดจากแพ็กเกจบางส่วนจากทางอินเทอร์เน็ตหากไม่พบซีดีที่แพ็กเกจในส่วนที่เราต้องการติดตั้ง
- เซทค่าต่างๆ ของสำหรับแพ็กเกจบางตัวที่ต้องการถูกกำหนดค่าก่อนการทำงาน

3.2 การติดตั้งไดร์เวอร์ของการ์ดเครือข่ายไร้สาย

ไดร์เวอร์ที่ใช้ในโครงงานนี้ สามารถดาวน์โหลดได้ที่ www.madwifi.net โดยที่เวอร์ชั่นที่ใช้ในโครงงานนี้คือ 0.9.2 โดยไดเรกทอรี่ที่เก็บโค้ดการทำงานของไดร์เวอร์ไว้ใน /usr/src

3.2.1 ความต้องการ

- ซอร์สของเคอร์เนลที่ต้องการ(ขึ้นอยู่กับชนิดของลีนุกซ์ โดยในที่นี้เราใช้ลีนุกซ์ดีเบียน ซึ่งจะต้องทำการโหลดเคอร์เนล เฮดเดอร์เพิ่มเติม)
- ต้องมีการสนับสนุนการทำ Wireless Extension โดยใช้ชื่อออพชั่นในเคอร์เนลว่า CONFIG_NET_RADIO ซึ่งสามารถตรวจสอบได้ในไฟล์ ".config"

- สนับสนุนกรทำงาน sysctl โดยตรวจสอบได้ในไฟล์ ".config" ในบรรทัดที่ CONFIG_SYSCTL
- สนับสนุนการทำ Crypto API ตรวจสอบได้จาก CONFIG_CRYPTO ในไฟล์ ".config" ภายในเคอร์เนล
- คอมไพล์เลอร์ต้องเป็นเวอร์ชั่นเดียวกันกับที่คอมไพล์เคอร์เนล

3.2.2 การติดตั้ง

- ทำการไปไดเรกทอรี่ของโปรแกรม จากนั้นใช้คำสั่ง make เพื่อทำการรันสคริปไฟล์ Makefile

  Host-ap:~# cd /usr/src/madwifi-0.9.2

  Host-ap:/usr/src/madwifi-0.9.2# make

- จากนั้นใช้คำสั่ง make install เพื่อรันคำสั่งใน Makefile ในช่วง install

  Host-ap:/usr/src/madwifi-0.9.2# make install

- ถึงขั้นนี้จะเป็นเป็นการนำไฟล์ที่ถูกคอมไพล์ไปไว้ในไดเรกทอรี่ที่พร้อมต่อการถูกเพิ่ม โดยเราจะทำการเพิ่มโมดูลโดยใช้คำสั่ง modprobe หรือ insmod ก็ได้โดยโมดูลที่สามารถเพิ่มได้มีดังนี้
  - ath_pci ไดร์เวอร์ของการ์ดเครือข่ายไร้สาย
  - ath_hal ตัวติดต่อกับ LINUX HAL
  - wlan ใช้ในการสนับสนุนการทำงานของเครือข่ายไร้สาย
  - wlan_wep เป็นการเข้ารหัสแบบ WEP
  - wlan_tkip เป็นการเข้ารหัสแบบ TKIP
  - wlan_ccmp เป็นการเข้ารหัสแบบ AES-CCMP
  - wlan_xauth สนับสนุนการทำงานของการพิสูจน์ตัวบุคคลภายนอก
  - wlan_acl การสนับสนุนการทำงานสำหรับ แอ็คเซสพอยท์
  - wlan_scan_ap สนับสนุนการค้นหา แอ็คเซสพอยท์
  - wlan_scan_sta สนับสนุนการค้นหา เครื่องลูกข่าย

3.3 การทำแอคเซสพอยท์บนลีนุกซ์ ด้วย Madwifi

- ทำการสร้างอินเทอร์เฟสของไวร์เลสขึ้นมาก่อน โดยในที่นี้ให้ใช้การสร้างอินเทอร์เฟสแบบอัติโนมัติเพื่อให้สร้างอินเทอร์เฟสที่รองรับการทำงานในโหมด มาสเตอร์ ได้โดยใช้คำสั่ง

  Host-ap#wlanconfig ath0 create wlandev wifi0 wlanmode ap

- สิ่งที่ได้จากขั้นตอนนี้คืออินเทอร์เฟส ath0 ซึ่งจะเป็นอินเทอร์เฟสของ access point ของเรา โดยหลังจากนี้เราต้องทำการสร้าง bridge เพื่อทำการเชื่อมต่ออินเทอร์เฟสของไวร์เลสกับอีเธอร์เน็ทเข้าด้วยกัน โดยในลีนุกซ์บางตัวต้องทำการติดตั้งแพ็กเก็ตเกี่ยวกับบริดจ์เพิ่ม โดยแพ็กเก็ตที่เพิ่มคือ "bridge-utils" ทำการสร้างอินเทอร์เฟส bridge และทำการเชื่อมต่ออินเทอร์เฟสไวร์เลสและอีเธอเนทเข้าด้วยกันโดยใช้คำสั่งต่อไปนี้

  Host-ap#brctl addbr br0

  Host-ap#brctl addif br0 ath0

  Host-ap#brctl addif br0 eth0

  จากนั้นทดลองใช้คำสั่ง brctl show br0 จะได้ผลลัพท์ดังนี้

```
bridge name     bridge id               STP enabled     interfaces
br0             8000.00065b7d2136       no              eth0
                                                        ath0
```

- หลังจากนั้นให้ทำการกำหนด ip address ของทั้งไวร์เลสอินเทอร์เฟสและอีเธอร์เนทอินเทอร์เฟส เป็น "0.0.0.0" และกำหนด ip address ให้กับ bridge ตามต้องการ

  Host-ap#ifconfig ath0 0.0.0.0

  Host-ap#ifconfig eth0 0.0.0.0

  Host-ap#dhclient br0

  จากนั้นทดลองรันคำสั่ง ifconfig ดูผลลัพธ์ที่ได้คือ

```
ath0      Link encap:Ethernet  HWaddr 00:13:F7:03:08:1B
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:904287 errors:0 dropped:0 overruns:0 frame:0
          TX packets:697747 errors:0 dropped:3214 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:181031382 (172.6 MiB)  TX bytes:473287324 (451.3
MiB)
br0       Link encap:Ethernet  HWaddr 00:06:5B:7D:21:36
```

```
          inet addr:10.49.176.29  Bcast:10.49.176.255
Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:699289 errors:0 dropped:0 overruns:0 frame:0
          TX packets:69850 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:105005114 (100.1 MiB)  TX bytes:4629492 (4.4 MiB)

eth0      Link encap:Ethernet  HWaddr 00:06:5B:7D:21:36
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:752035 errors:0 dropped:0 overruns:31 frame:0
          TX packets:498901 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:553185029 (527.5 MiB)  TX bytes:160463122 (153.0
MiB)
          Interrupt:18 Base address:0xdc80

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:3522 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3522 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1786589 (1.7 MiB)  TX bytes:1786589 (1.7 MiB)

wifi0     Link encap:UNSPEC  HWaddr 00-13-F7-03-08-1B-00-00-00-00-00-
00-00-00-00-00
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1090865 errors:0 dropped:6515 overruns:0
frame:937064
          TX packets:741713 errors:9385 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:199
          RX bytes:222739512 (212.4 MiB)  TX bytes:494960913 (472.0
MiB)
          Interrupt:17 Memory:e0956000-e0966000
```

- หลังจากนั้นก็ทำการกำหนดค่าต่างๆ ให้กับแอคเซสพอยท์ที่สร้างขึ้น เช่นค่า essid , channel, WEP เป็นต้น

  Host-ap#iwconfig ath0 essid “IT_KMITL”

  Host-ap#iwconfig ath0 channel 11

หลังจากนั้นทดรองรันคำสั่ง iwconfig ath0 ผลลัพธ์ที่ได้คือ

```
ath0      IEEE 802.11g  ESSID:"IT_KMITL"
          Mode:Master  Frequency:2.462 GHz  Access Point:
00:13:F7:03:08:1B
          Bit Rate:0 kb/s   Tx-Power:9 dBm   Sensitivity=0/3
          Retry:off   RTS thr:off   Fragment thr:off
          Encryption key:off
          Power Management:off
          Link Quality=53/94  Signal level=-42 dBm  Noise level=-95 dBm
          Rx invalid nwid:12792  Rx invalid crypt:0  Rx invalid frag:0
```

```
Tx excessive retries:0  Invalid misc:0   Missed beacon:0
```

สังเกตุได้ว่าไม่มีข้อมูลในส่วนของช่องสัญญาณ แต่เราสามารทราบได้จากการใช้ย่านความถี่ของอินเทอร์เฟสไวร์เลสซึ่งในที่นี้ใช้คาวมถี่ 2.462 กิกเฮิร์ตซ ซึ่งเรานำมาคำนวนหาช่องสัญญาณได้ว่าใช้ช่องสัญญาณใด โดยหาใช้ช่องสัญญาณที่ 1 จะมีความถี่ที่ 2.412 จากนั้นช่องสัญญาณที่เพิ่มขึ้นทีละ 0.005 กิกเฮิร์ตซ ซึ่งเราจะพบช่องสัญญาณที่ใช้คือช่องสัญญาณที่ 6

บทที่ 4
การพัฒนาโครงงาน

4.1 ภาพรวมของโครงงาน

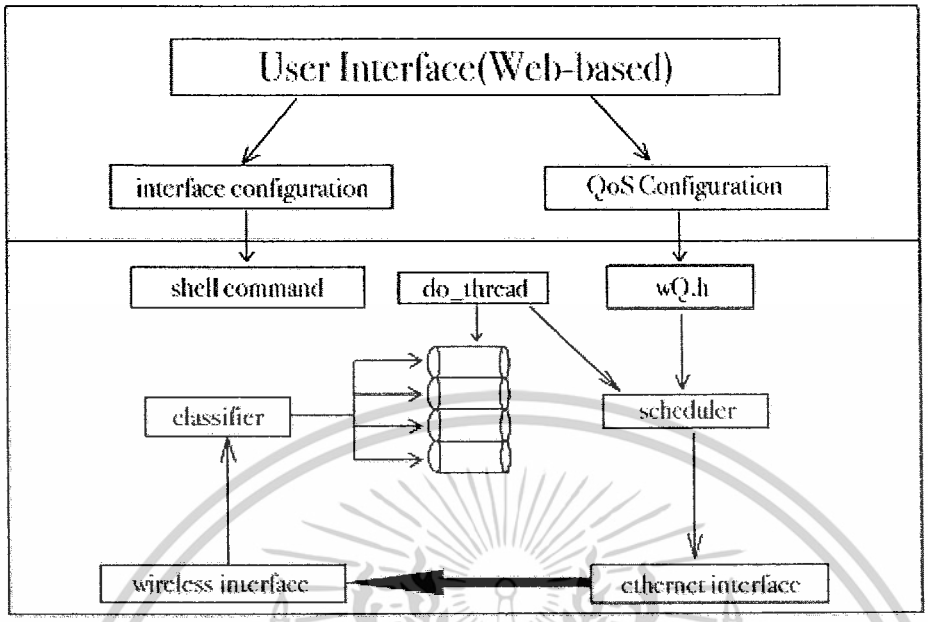
ในโครงงานนี้ได้มุ่งเน้นในการจัดการคิวของข้อมูลเป็นหลัก เพื่อให้ข้อมูลประเภทที่ผู้ใช้ให้ความสำคัญ สามารถสื่อสารออกไปจากเครือแลนข่ายไร้สายได้อย่างมีประสิทธิภาพมากกว่าข้อมูลอื่น โดยในการพัฒนาแบ่งออกเป็นสองช่วงในการพัฒนาคือ

1. ส่วนการพัฒนาเพื่อให้ได้ส่วนในการจัดการคิวข้อมูลซึ่งต้องทำใน ระดับเคอร์แนลของลีนุกซ์ ซึ่งจะทำการพัฒนาต่อจากการพัฒนาโมดูลที่ทำการเชื่อมอินเทอร์เฟสระหว่างอีเธอร์เน็ทกับเครือข่ายไร้สาย
2. การพัฒนาในเรื่องของส่วนติดต่อกับผู้ใช้ซึ่งจะมีอยู่ในรูปแบบของหน้าเวปเพจ โดยจุดประสงค์เพื่อให้ผู้ใช้สามารถปรับแต่งได้ง่าย

โดยหลักการในการจัดการคิวของโครงงานนี้ ความสำคัญของข้อมูลที่ผู้ใช้ต้องการจะถูกแบ่งไปตามพอร์ทปลายทางในระดับชั้นทรานสปอร์ทของโอเอสไอโมเดล โดยในการจัดการคิวจะทำที่ฝั่งแพ็กเก็ตที่ได้รับจากอินเทอร์เฟสไร้สายเท่านั้น โดยการจัดการคิวนั้นเราทำการแบ่งคิวออกทั้งหมดเป็น 4 คิวด้วยกันโดยแต่ละคิวจะเก็บข้อมูลที่มีพอร์ทปลายทางเดียวกันทั้งหมด ยกเว้นอยู่ 1 หนึ่งคิวที่จะทำการเก็บข้อมูลจากพอร์ทอื่นๆ ที่ไม่ตรงกับสามคิวที่มีการกำหนดพอร์ท

ส่วนการเลือกคิวในการส่งข้อมูลออกทางอุปกรณ์อีเธอร์เน็ทนั้น เราทำการตัดสินใจจากข้อมูลที่ผู้ใช้กำหนดให้มีความสำคัญสูงสุดจะถูกออกจากคิวก่อน โดยการส่งจะทำการเลือกคิวและส่งข้อมูลออกจากคิวในปริมาณที่ถูกกำหนดไว้ตามความสำคัญ จนครบตามจำนวนหรือจนกว่าคิวนั้นว่าง จากนั้นถึงจะทำการดึงข้อมูลจากคิวอื่นและทำการส่งข้อมูลออก นอกจากนี้ยังมีการกำหนดค่าปริมาณคิวสูงสุดที่แต่ละคิวจะสามารถมีข้อมูลเก็บไว้อยู่ได้ โดยหากเกินค่านี้ก็จะสามารถให้คิวนั้นสามรถส่งข้อมูลได้สูงสุดมากกว่าปกติ 2 เท่า

ในส่วนการทำงานที่ติดต่อกับผู้ใช้ ได้ทำการพัฒนาเวปเพจเพื่อให้ผู้ใช้สามารถดูค่าต่างๆ ที่กำหนดไว้อยู่ ณ ปัจจุบันของเครือข่ายรวมทั้งในเรื่องการจัดการคุณภาพด้วยเช่นกัน นอกจากนั้นผู้ใช้ยังสามารถปรับแต่งการใช้งานบางอย่างได้ด้วยเช่นกัน โดยการทำงานในส่วนนี้ทำการพัมนาด้วย PHP เพื่อทำการติดต่อกับเทอร์มินัลและในเรื่องการอ่านและเขียนไฟล์ โดยภาพรวมของโครงงานเป็นไปตามรูปที่ 4.1



รูปที่ 4.1 แสดงภาพรวมของโครงงาน

4.2. การพัฒนาโครงงานในระดับเคอร์เนล

การพัฒนาในส่วนนี้ เป็นการพัฒนาเพื่อทำให้อินเทอร์เฟสสองอินเทอร์เฟสคือ อินเทอร์เฟสที่ต่อกับเครือข่ายใช้สายกับอินเทอร์เฟสที่เชื่อมต่อกับเครือข่ายไร้สายนั้น สามารถส่งผ่านข้อมูลระหว่างกันได้เสมือนเป็นบริดจ์แต่จะมีการใส่คำสั่งในการจัดการข้อมูลลงไปด้วย ซึ่งการพัฒนาในช่วงนี้เป็นการพัฒนาในระดับของเคอร์เนลที่มีรูปแบบเป็นโมดูล ซึ่งการตอบสนองเมื่อเปรียบเทียบกับการพัฒนาในระดับยูเซอร์สเปสแล้ว การทำงานในระดับเคอร์เนลจะสามารถตอบสนองงานได้เร็วกว่า

การพัฒนาในขั้นนี้ทำการแบ่งการพัฒนาเป็น 2 ช่วง คือ ช่วงแรกทำการพัฒนาโมดูลที่ทำการเชื่อมต่อระหว่างอินเทอร์เฟส และช่วงที่สองเป็นการพัฒนาในส่วนของการรับประกันคุณภาพ โดยจะมีการสร้างคิวขึ้นมาเพื่อจัดเรียงประเภทของข้อมูลที่แบ่งตามพอร์ทในระดับของทรานสปอร์ท

4.2.1. การพัฒนาส่วนติดต่อระหว่างอินเทอร์เฟส

โดยในการพัฒนาช่วงแรกเป็นการพัฒนาให้โมดูล สามารถทำการการฟอร์เวิร์ดข้อมูลกันระหว่างอินเทอร์เฟสได้ โดยโมดูลในส่วนนี้ยังไม่มีฟังก์ชันในการทำงานเกี่ยวกับการรับประกันคุณภาพ ซึ่งเมื่อพัฒนาโมดูลในช่วงนี้เสร็จสิ้น โมดูลที่ได้จะมีการทำงานคล้ายเป็นบริดจ์ โดยหลักการของทั้งสองอินเทอร์เฟสมีหลักการคล้ายกันคือ เมื่ออินเทอร์เฟสสามารถได้รับเฟรมข้อมูลมาจากเครือข่ายของตน ข้อมูลก็จะโอนถ่ายส่งผ่านไปยังอีกอินเทอร์เฟสหนึ่งเพื่อทำการส่งเฟรมข้อมูล

4.2.1.1. ไลบารีที่สำคัญ

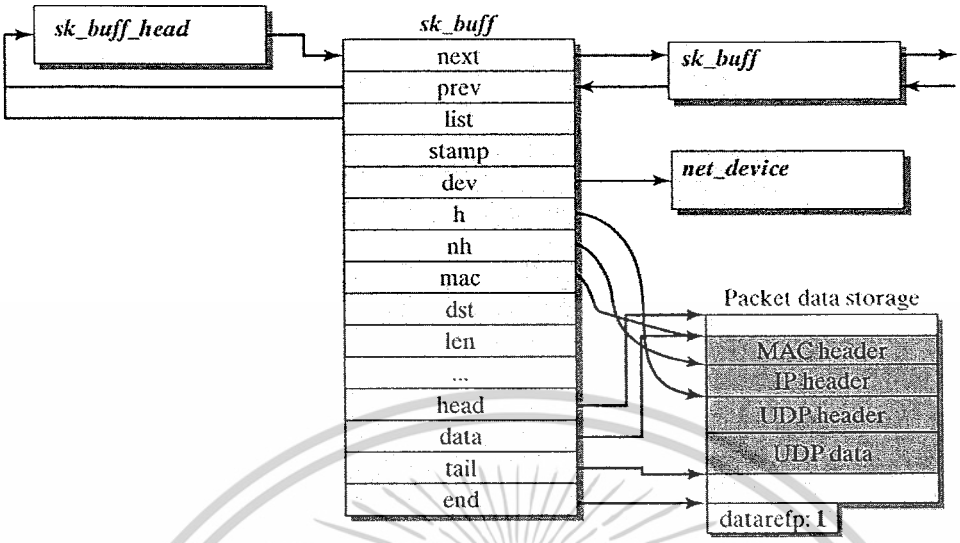
1). skbuff.h

เป็นไลบารีที่จัดการในเรื่อง ซอคเก็ตบัฟเฟอร์(socket buffer) ซึ่งมีโครงสร้างที่ใช้ในแสดงแทนข้อมูลและมีฟังก์ชันที่ใช้สำหรับจัดการข้อมูลในบัฟเฟอร์ ซึ่งเป็นรูปแบบที่ถูกใช้ในการประมวลผลในระดับของเคอร์แนล โดยตั้งแต่ในแอพพลิเคชั่นส่งข้อมูลเข้าไปในซอคเก็ต และซอคเก็ตจะทำการสร้างข้อมูลที่เป็นโครงสร้างแบบ sk_buff เข้ามาในเคอร์เนล โดยมีพารามิเตอร์ต่างๆ จากแอพพลิเคชั่นอยู่ ซึ่ง sk_buff จะถูกใช้จนกระทั่งถูกส่งข้อมูลให้อุปกรณ์เครือข่าย โดยในการใช้งานจะมีสองสถานการณ์ก็คือ รับหรือส่งแพคเก็ต โดยโครงสร้างที่ใช้ในการจัดการบัฟเฟอร์ที่สำคัญตัวหนึ่งก็คือ sk_buff โดย ซอคเก็ตบัฟเฟอร์สามารถแบ่งได้ออกเป็นสองส่วน คือ

- packet data : เป็นส่วนที่เก็บข้อมูลที่ถูกใช้ส่งในเครือข่าย โดยข้อมูลในส่วนนี้จะหมายถึง ข้อมูลในของแต่ละระดับชั้นที่รับผิดชอบโดยใน sk_buff นั้นคือส่วนที่เป็น sk_buff->data จนถึง sk_buff->tail
- management data : เป็นส่วนที่ใช้ในการจัดการข้อมูลที่จะถูกส่งออกไปจริง โดยมากใช้เพื่อระบุคุณสมบัติต่างๆ ของข้อมูลเอง(จำพวก pointer, timer etc.) ซึ่งจะเป็นรูปแบบของฟังก์ชันที่ใช้ในการแลกเปลี่ยนข้อมูลของระหว่างโปรโตคอล โครงสร้างซอคเก็ตบัฟเฟอร์มีข้อมูลภายในบางส่วนที่สำคัญดัง รูปที่ 4.2

พารามิเตอร์ที่สำคัญของ sk_buff คือ

- next และ prev ใช้เพื่อจัดการลำดับซอคเก็ตบัฟเฟอร์ในคิว(struct skb_queue_head) โดยสามารถใช้ฟังก์ชันสองตัวคือ skb_queue_head(), skb_dequeue_tail() โดยข้อมูลในส่วนนี้จะไม่ควรถูกเปลี่ยนแปลงจากโปรแกรมเมอร์โดยตรง(ควรใช้ผ่านฟังก์ชัน)



รูปที่ 4.2 แสดงโครงสร้างของข้อมูลชนิด sk_buff

- list เป็นตัวชี้ตำแหน่งที่ซอคเก็ตบัฟเฟอร์เก็บอยู่(sk_buff_head)
- sk ชี้ไปยังซอคเก็ตที่สร้างแพคเกจขึ้นมา อาจเป็นค่า null ก็ได้
- stamp ระบุเวลาที่สร้าง โครงสร้างขึ้นมาหรือเวลาที่แพคเก็ตมาถึงระบบ
- dev เป็นพอยเตอร์ที่อ้างอิงกับอุปกรณ์ที่สามารถใช้งานกับแพ็คเก็ตนี้ได้
- h, nh, mac เป็นพอยเตอร์ที่ชี้เฮดเดอร์ของข้อมูลในแต่ละระดับชั้น โดย h ชี้ไปยังตำแหน่งเริ่มต้นของเฮดเดอร์ในระดับทรานสปอร์ท, nh ชี้ไปยังตำแหน่งเริ่มต้นในระดับเนทเวิร์ค และ mac ชี้ไปยังตำแหน่งเริ่มต้นในระดับดาตาลิงค์
- dst อ้างอิงข้อมูลจาก routing cache โดยเป็นข้อมูลบอกว่าแพคเก็ตกำลังจะไปที่เครื่องใดหรืออ้างอิง MAC header ของอินเทอร์เฟสที่เก็บข้อมูลตัวนี้ไว้ ข้อมูลในส่วนนี้เป็นโครงสร้างชนิด dst_entry
- cloned เป็นส่วนที่ระบุว่าแพคเก็ตนี้เคยถูกโคลนมาก่อนหรือไม่ ถ้าเคยจะถูกกำหนดค่ามากกว่าศูนย์
- pkt_type เป็นส่วนที่ระบุประภทของแพคเก็ตซึ่งมีดังต่อไปนี้
  - PACKET_HOST
  - PACKET_BROADCAST
  - PACKET_MULTICAST
  - PACKET_OTHERHOST

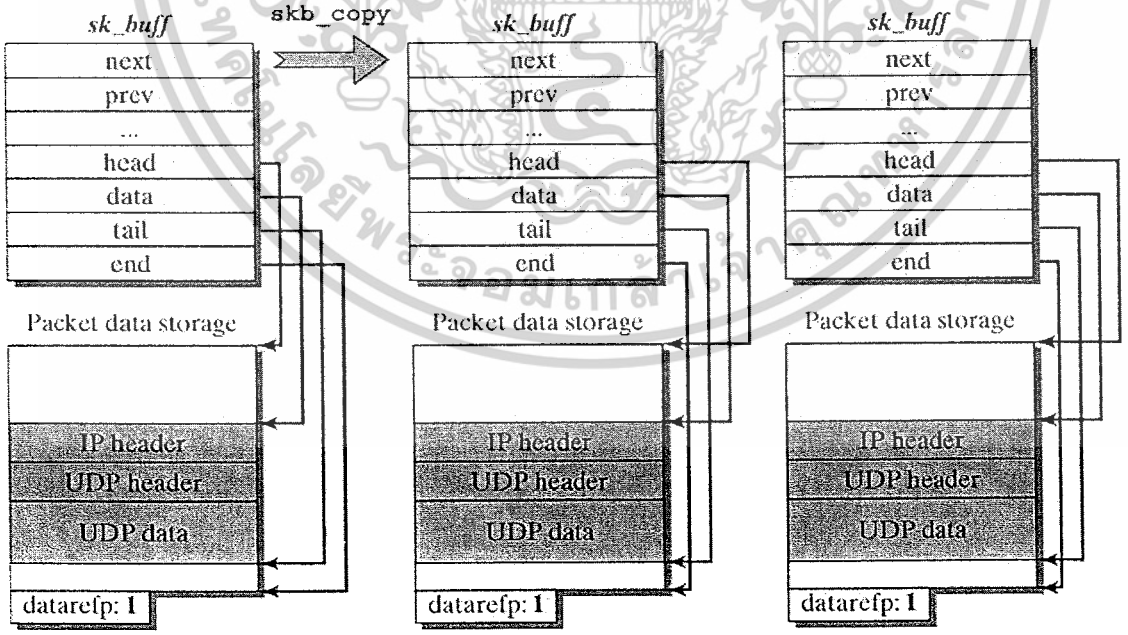
- PACKET_OUTGOING
- PACKET_LOOPBACK
- PACKET_FASTROUTE

- len เป็นข้อมูลแบบจำนวนเต็มที่บ่งบอกถึงขนาดของข้อมูล ซึ่งจะนับจากขนาดข้อมูลที่เคอร์เนลควบคุมหรือทำงานได้เท่านั้น เพราะฉะนั้นแล้วข้อมูลในส่วน preamble, padding และ checksum จะไม่ถูกรวมไว้ด้วย
- data, head, tail, end ในส่วนของ data และ tail จะชี้ตำแหน่งเริ่มต้นของข้อมูล ซึ่งชี้ก็ต่อเมื่อมีข้อมูลอยู่จริง(และใช้งานได้จริงเท่านั้น) ส่วน head และ end จะบ่งบอกถึงตำแหน่งทั้งหมดที่เราสามารถใช้ในการเก็บข้อมูลของแพคเก็ตได้

นอกจากนี้ในเคอร์แนลยังฟังก์ชันให้เราใช้งานในการจัดการเกี่ยวกับ ซอคเก็ตบัฟเฟอร์ด้วยโดยสามารถแบ่งได้ออกเป็นกลุ่มดังนี้

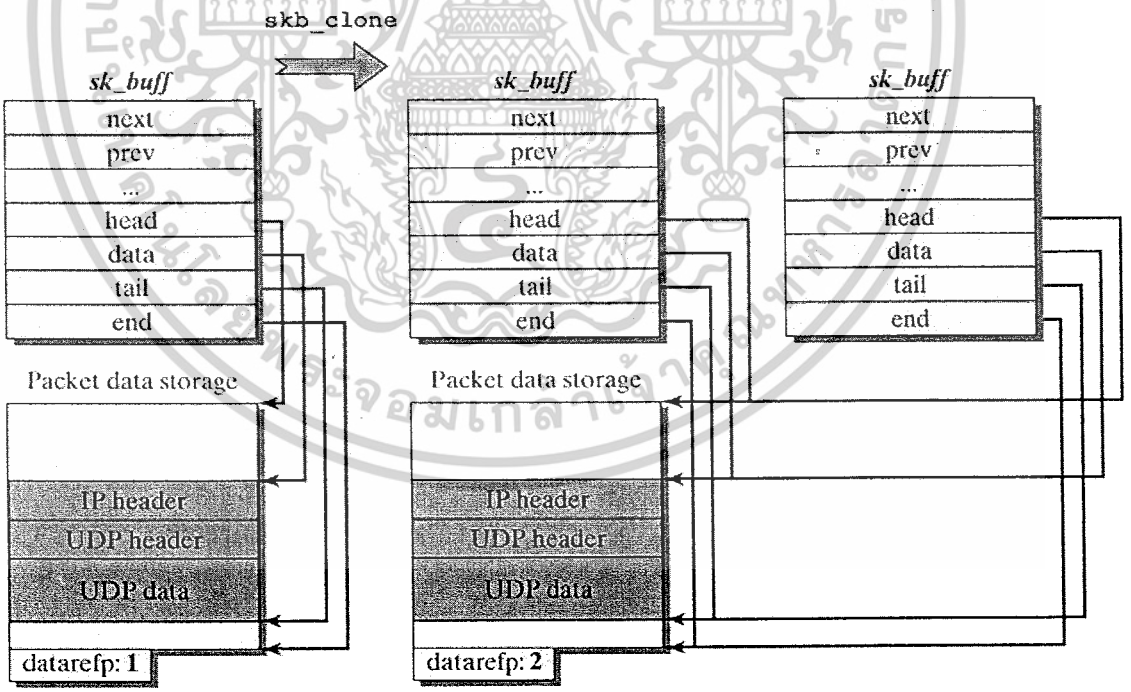
การสร้างและปล่อยซอคเก็ตบัฟเฟอร์

- alloc_skb(size, gfp_mask) เป็นฟังก์ชันที่ใช้ในการจัดสรรพื้นที่ในการเก็บโครงสร้างซอคเก็ตบัฟเฟอร์โดยในกรณีนี้ต้องทำการระบุขนาดที่ทำการเก็บด้วย ส่วน gfp_mask เป็นแฟลกที่ใช้สำหรับในการจองพื้นที่ในหน่วยความจำ



รูปที่ 4.3 แสดงการทำสำเนาโครงสร้างข้อมูลจากฟังก์ชัน skb_copy()

- skb_copy(skb, gfp_mask) ใช้ในการสร้างสำเนา skb ซึ่งเป็นโครงสร้างซึ่งเป็นชนิดซอคเก็ตบัฟเฟอร์ โดยเป็นการทำซ้ำทั้งในส่วนของโครงสร้างและข้อมูล โดยขั้นแรกจะทำการเรียกฟังก์ชัน alloc_skb() เพื่อระบุตำแหน่งของโครงสร้างใหม่ที่จะใช้เป็นโครงสร้างสำเนาใหม่ซึ่งไม่มีข้อมูลอยู่ภายในโครงสร้าง และค่อยกำหนดค่าในฟิลด์ต่างๆ เข้าไปภายหลัง โดยรูปแบบเมื่อทำสำเนาดูได้ดัง รูปที่ 4.3 โดยหน่วยความจำที่เก็บเพย์โหลดของซอคเก็ตบัฟเฟอร์ใหม่นั้นต้องทำระบุให้ด้วย kmalloc() และสำเนาข้อมูลด้วยคำสั่ง memcpy() จากนั้นพอยเตอร์ถึงจะชี้ไปยังข้อมูลใหม่ซึ่งเป็นผลที่ได้จากคำสั่ง skb_copy() คือได้โครงสร้างซอคเก็ตบัฟเฟอร์ตัวใหม่ซึ่งมีคุณสมบัติในการทำงานเป็นอิสระต่อกัน
- skb_clone(skb) เป็นการสร้าง ซอคเก็ตบัฟเฟอร์ขึ้นมาใหม่แต่เฉพาะโครงสร้างเท่านั้น แต่ในส่วนของข้อมูลจะยังคงอ้างอิงจากตำแหน่งเดิม และในส่วนของตัวแปรที่ระบุจำนวนอ้างอิงก็จะเพิ่มขึ้นตามไปด้วยเช่นกัน โดยโครงสร้างที่ได้จากการโคลนนี้จะเป็นลักษณะอ่านได้อย่างเดียวเท่านั้น รูปแบบของการโคลน ดังรูปที่ 4.4



รูปที่ 4.4 แสดงการทำงานของฟังก์ชัน skb_clone()

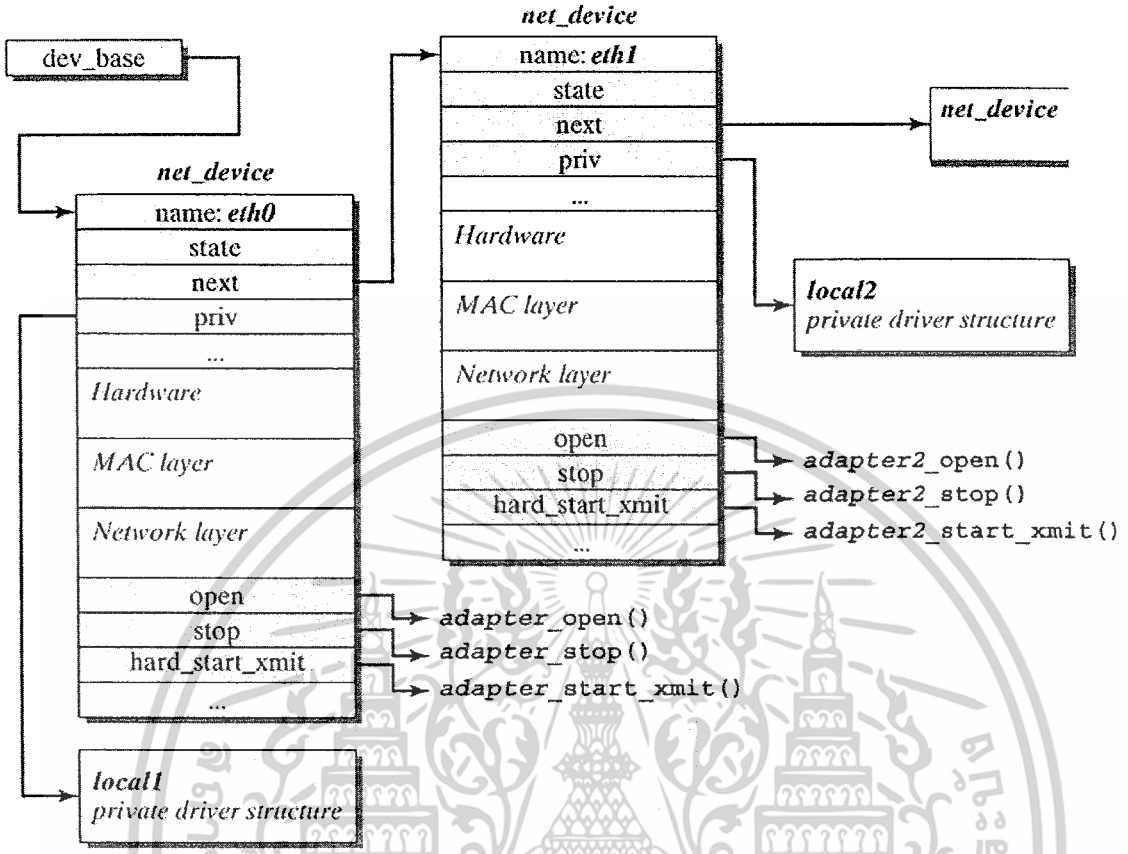
- kfree_skb() เป็นฟังก์ชันที่ใช้ในการคือทรัพยากรให้กับระบบ โดยทรัพยากรในที่นี้คือซอคเก็ตบัฟเฟอร์

การจัดการในส่วนของข้อมูล

- skb_get(sk_buff) จำนวน user ที่ทำการอ้างอิงโครงสร้างนี้อยู่
- skb_put(skb,len) เป็นการเพิ่มข้อมูลเข้าไปด้านท้ายต่อจากข้อมูลเดิม การทำลักษณะนี้เราจะต้องมั่นใจว่าพื้นที่ด้านท้าย(tailroom) มีพื้นที่เหลือเพียงพอ
- skb_push(skb,len) คล้ายกับ skb_put() เพียงแต่เป็นการเพิ่มด้านหน้าของข้อมูล ซึ่งก็เช่นกันเราควรที่จะตรวจสอบว่าพื้นที่ด้านหน้า(headroom) มีพื้นที่เหลือเพียงพอ
- skb_pull(skb,len) ทำการดึงข้อมูลขนาด len ของโครงสร้าง skb นับจากจุดเริ่มต้น
- skb_reserve(skb, len) เพิ่มขนาดพื้นที่ของข้อมูลด้านท้ายเป็นขนาด len ไบท์(ส่วนที่เพิ่มเป็นส่วนของ packet data เท่านั้น)

2). netdevice.h

เป็นไลบารีที่ใช้ในการเก็บข้อมูลและมีฟังก์ชันในการติดต่อกับอุปกรณ์เครือข่าย โดยในการพัฒนาโครงงานนี้เราต้องใช้ไลบารีนี้ในการสั่งงานอินเทอร์เฟสอีเธอร์เนทและอินเทอร์เฟสไร้สาย ซึ่งอุปกรณ์เครือข่ายจะถูกจัดให้อยู่ในรูปของ singly linked linear list โดยภายใน netdevice.h จะมีโครงสร้าง ที่ใช้ในการแทนอินเทอร์เฟสหรืออุปกรณ์คือ net_device ซึ่งมีรายละเอียดดังรูปที่ 4.5



รูปที่ 4.5 แสดงคครงสร้างข้อมูลของตัวแปรชนิด net_device

ฟิลด์ที่สำคัญๆ ได้แก่

- name เป็นชื่ออินเทอร์เฟสเครือข่าย เช่น eth0, ath0 เป็นต้น
- owner เป็นพอยเตอร์ที่ชี้โครงสร้างโมดูลที่ทำการสร้าง โครงสร้าง net_device ขึ้นมา
- state ข้อมูลแสดงสถานะของอุปกรณ์เครือข่าย โดยมี
  - LINK_STATE_START แสดงว่าอุปกรณ์กำลังทำงานอยู่ โดยเป็นแฟลกที่ควรจะสามารถอ่านได้เพียงอย่างเดียว โดยมีฟังก์ชัน netif_running(dev) เป็นฟังก์ชันที่สร้างแฟลกนี้ขึ้น
  - LINKS_STATE_XOFF เป็นการบ่งบอกว่าอินเทอร์เฟสขณะนี้สามารถที่จะส่งข้อมูลได้ โดยฟังก์ชัน netif_stop_queue() เป็นการกำหนดแฟลกนี้และมีฟังก์ชัน netif_start_queue() เป็นฟังก์ชันที่ลบแฟลกนี้ออก
- trans_start เก็บเวลาที่ใช้ในการเริ่มส่งข้อมูล
- last_rx ใช้เก็บเวลาที่ได้รับข้อมูลสุดท้าย

- priv เป็นพอย์เตอร์ที่ชี้ไปยังข้อมูลเฉพาะของอุปกรณ์เครือข่าย ซึ่งส่วนนี้จะประกอบไปด้วยตัวแปรหรือคำสั่งที่ใช้ในการจัดการอุปกรณ์เครือข่าย
- refcnt เก็บจำนวนที่อ้างอิงจากอุปกรณ์เครือข่ายนี้
- xmit_lock, xmit_lock_owner และ queue_lock ใช้เพื่อป้องการการส่งข้อมูลหรือเข้าถึงทรัพยากรพร้อมๆ กัน โดยมีรูปแบบเป็น spinlock_t
- rmem_end, rmem_start บอกตำแหน่งเริ่มและสุดท้ายของบัฟเฟอร์ที่ใช้ในการรับข้อมูล
- mem_start, mem_end บอกตำแหน่งเริ่มและสุดท้ายของบัฟเฟอร์ที่ใช้ในการส่งข้อมูล
- irq เป็นเลขที่คอยจัดการเกี่ยวกับอินเทอร์รัพ
- if_port เก็บชนิดของสื่อที่ใช้ในการสื่อสารของอินเทอร์เฟส
- hard_header_lenght เป็นขนาดของเฮดเดอร์ในระดับดาตาลิงค์
- type ระบุชนิดของอุปกรณ์เนทเวิร์ค
- dev_addr[max_ADDR_LEN] เป็นส่วนที่เก็บแอดเดรสของอุปกรณ์ ในเลเยอร์ 2

ฟังก์ชันการทำงานของอุปกรณ์เครื่อข่าย

- init() ใช้เพื่อค้นหาหรือเริ่มต้นในการติดต่ออุปกรณ์เครือข่าย ซึ่งจะทำการค้นหาและกำหนดชนิดของอินเทอร์เฟส โดยหลักแล้วก็จะต้องมี net_device ที่ถูกสร้างขึ้นมาและยังจัดการเกี่ยวกับข้อมูลต่างๆ ของอุปกรณ์ จากนั้นอุปกรณ์เน็ทเวิร์คก็จะลงทะเบียนโดย register_netdevice()
- uninit() ใช้เมื่ออุปกรณ์ต้องการออกจากระบบ(unregister_netdevice())
- open() ทำการสั่งให้อุปกรณ์เริ่มทำงาน โดยการจะสั่ง open() ได้อุปกรณ์จะต้องถูกลงทะเบียนเสียก่อน นอกจากจะใช้ฟังก์ชันนี้แล้วยังสามารถใช้คำสั่ง 'ifconfig' ในเทอร์มินัลก็ได้เช่นกัน
- stop() ยกเลิกการทำงานของอุปกรณ์และคืนทรัพยากรต่างๆ ให้กับระบบถึงแม้จะไม่ทำงานแต่อุปกรณ์ยังลงทะเบียนอยู่ในระบบ
- hard_start_xmit() โดยเป็นคำสั่งที่ใช้ในการส่งข้อมูล โดยหาส่งสำเร็จจะรีเทิร์นค่ากลับมาเป็น 0, หากล้มเหลวจะคืนค่าอื่นๆ ออกมา
- get_stats()ดูค่าสถิติและข้อมูลเกี่ยวกับอุปกรณ์เน็ทเวิร์ค โดยข้อมูลจะรีเทิร์นด้วยโครงสร้าง net_device_stats

- get_wireless_stats() เมื่อเรียกใช้ฟังก์ชันนี้จะรีเทิร์นข้อมูลสำหรับอุปกรณ์เนทเวิร์คไร้สาย เป็นข้อมูลในโครงสร้าง iw_statistics นอกจากนี้ยังสามารถตรวจดูได้จากคำสั่ง 'iwconfig'
- do_ioctl() โดยปกติฟังก์ชันนี้จะไม่ถูกเรียกใช้ในโปรโตคอลระดับบน เพราะไม่เป็นฟังก์ชันทั่วไป(ปกติจะใช้ ioctl())
- set_config() ใช้เพื่อเปลี่ยนแปลงค่าคอนฟิกูเรชั่นต่างๆ ขณะที่ระบบกำลังทำงานอยู่

4.2.1.2. การพัฒนาโมดูลเชื่อมอินเทอร์เฟสเบื้องต้น

ในขั้นตอนนี้ได้ทำการสร้างโมดูลขึ้น โดยมีการพัฒนาแบ่งออกเป็น 4 ฟังก์ชันหลักๆ คือ

1). static int bridge_init_module()

เป็นฟังก์ชันแรก ที่จะถูกเรียกเมื่อเราเริ่มทำการแทรกโมดูลที่พัฒนาขึ้นเข้าไป โดยจุดประสงค์ของฟังก์ชันนี้เพื่อ ตรวจสอบอุปกรณ์ที่ใช้ในเครือข่ายทั้งสองฝั่ง ว่ามีและพร้อมที่จะทำงาน, จัดเตรียมและรีจิสเตอร์เพื่อทำการดักแพคเก็ตข้อมูล, กำหนดตัวแปรที่จำเป็นในการทำงานของโมดูล โดยขั้นตอนการทำงานมีดังนี้

```
ath_if = dev_get_by_name("ath0");
        if(ath_if)
                /* มีอุปกรณ์ที่ชื่อ ath0 อยู่จริงในระบบ*/
        else
                return -ENODEV; /* ไม่มีอุปกรณ์ชื่อ ath0 อยู่ในระบบ*/
```

- ตรวจสอบว่าอินเทอร์เฟสนั้นมีอยู่จริงหรือไม่ โดยทำการค้นหาโดยใช้คำสั่ง dev_get_by_name(name) ซึ่งเป็นการค้นหาอุปกรณ์โดยการใส่พารามิเตอร์ที่เป็นชนิดคาแรคเตอร์ ที่เป็นชื่อของอินเทอร์เฟสนั้น(ชื่ออินเทอร์เฟสนั้นสามารถอ้างอิงได้ตามฟิลด์ในโครงสร้าง net_device->name) ซึ่งผลที่ได้จากฟังก์ชันนี้คือ พอยเตอร์ของโครงสร้าง net_device ที่ชี้ไปยังอุปกรณ์ดังกล่าว แต่หากอุปกรณ์ไม่มีอยู่จริง จะทำการส่งค่า NULL กลับมาให้ โดยในโมดูลนี้นอกจากเพื่อหาว่ามีอินเทอร์เฟสอยู่จริงหรือไม่แล้ว ยังเป็นการกำหนดเพื่อให้ได้โครงสร้างของอุปกรณ์มาเพื่อใช้ในส่วนอื่นๆ อีกด้วย
- โดยหากพบว่าค่าในโครงสร้าง net_device ที่ได้จาก ฟังก์ชัน dev_get_by_name() จะ

ส่งค่าออกจาฟังก์ชันนี้(bridge_init_module()) ด้วยค่าที่เป็นติดลบ ซึ่งส่งผลให้การแทรกโมดูลเข้าในระบบประสบความล้มเหลว

```
struct packet_type my_pkt;  /* สร้างโครงสร้าง packet_type*/
my_pkt.type = htons(ETH_P_ALL); /*กำหนดชนิดของเฟรม*/
my_pkt.dev = ath_if;  /* กำหนดอินเทอร์เฟสที่ทำการดักรอ */
my_pkt.func = ath_recv; /* ฟังก์ชันที่ถูกเรียกเมื่อมีข้อมูลเข้ามา */
my_pkt.data = NULL;  /* ระบุประเภทของข้อมูล */
my_pkt.next = NULL; /* ใช้ในการเชื่อมโยงกับ packet_type ตัวอื่น */
dev_add_pack(&my_pkt);
```

- จากนั้นทำการสร้างโครงสร้าง เพื่อลงทะเบียนในการดักรอข้อมูลที่จะเข้ามาทางอินเทอร์เฟส โดยการลงทะเบียนเพื่อทำการดักรอจะใช้คำสั่ง dev_add_pack(struct packet_type) โดยเราจะต้องสร้างโครงสร้าง pack_type เพื่อใช้ในการลงทะเบียน โดยภายในโครงสร้างมีอยู่ด้วยกันห้าส่วนคือ type, dev, func, data และ next
- โดยเมื่อเราใช้คำสั่ง dev_add_pack() เมื่อมีข้อมูลใดๆ เข้ามาในอินเทอร์เฟสที่ระบุไว้ ซึ่งฟังก์ชันใน dev.c จะมาทำการเรียกฟังก์ชันที่เราได้ลงทะเบียนไว้(packet_type.func) พร้อมทั้งส่งพารามิเตอร์มาให้สามประเภทด้วยกันคือ sk_buff, net_device และ packet_type
- สุดท้ายคือการส่งค่ากลับ โดยหากมาถึงขั้นนี้ด้วยความสมบูรณ์ค่าที่ส่งกลับคือ 0

2). static void bridge_exit_module()

เป็นฟังก์ชันที่ถูกเรียกเมื่อมีความต้องการในการยกเลิก หรือถอดถอนโมดูลนี้ออกจากระบบเช่นจากคำสั่ง rmmod โดยจุดประสงค์ของฟังก์ชันนี้คือ ทำการถอดทรัพยากรที่ทำการจองออกจากระบบ โดยมีขั้นตอนการทำงานดังนี้

- ทำการถอนฟังก์ชันที่ทำการดักรอข้อมูลจากอินเทอร์เฟส ออกจากระบบโดยการใช้คำสั่ง dev_remove_pack(struct packet_type) โดยใช้โครงสร้าง packet_type ที่มีข้อมูลเหมือนกันกับที่ทำการเพิ่มการดักฟังข้อมูลของระบบใน bridge_init_module()

3). int eth_recv() และ int ath_recv()

โดยทั้งสองฟังก์ชันมีพารามิเตอร์ที่ถูกส่งมา 3 ค่าเหมือนกัน คือ struct sk_buff, struct net_device และ struct packet_type โดยทั้งสองฟังก์ชันเป็นฟังก์ชันที่ทำการลงทะเบียนไว้เพื่อดัก

รอข้อมูลจากอุปกรณ์เครือข่าย ซึ่งการลงทะเบียนได้ทำไปใน bridge_init_module() ดังข้างต้นแล้ว โดยทั้งสองฟังก์ชัน มีจุดประสงค์เดียวกันคือ ทำการฟอร์เวิร์ดข้อมูลที่ดักจับได้ไปยังอีกอินเทอร์เฟสหนึ่ง โดยขั้นตอนการทำงานมีดังนี้

```
struct sk_buff temp;
temp = skb_copy(struct sk_buff, GFP_ATOMIC)
```

- ทำการสร้างพอยเตอร์โครงสร้างซอคเก็ตบัฟเฟอร์ขึ้นมา เพื่อนำไปชี้กับโครงสร้างสำเนาซอคเก็ตบัฟเฟอร์ที่ได้รับจากฟังก์ชัน netif_recieve_skb() โดยจะเป็นพอยเตอร์ที่ใช้ตลอดในฟังก์ชันนี้แทนโครงสร้างที่ได้รับ GFP_ATOMIC ที่เป็นอาร์กิวเมนท์นั้นเป็นรูปแบบของ atomic แบบหนึ่ง(รูปแบบของการป้องกันการแย่งชิงทรัพยากรที่ใช้ร่วมกัน) ที่ใช้ในการสำรองหน่วยความจำ
- จากนั้นทำการเพิ่มเฮดเดอร์ของฮาร์ดแวร์ โดยเพิ่มพื้นที่ส่วนเฮดรูม(headroom) เนื่องจากข้อมูลที่ได้รับจะถูกถอดเฮดเดอร์ของฮาร์ดแวร์ออก

```
spin_lock_irq(&eth_if->xmit_lock);
```

- ทำการป้องกัน อินเทอร์รัพจากโปรเซสอื่นๆ ไม่ให้เข้าถึงข้อมูลในช่วงนี้ โดยการใช้คำสั่ง spin_lock_irq(spinlock_t) โดยอาร์กิวเมนท์ที่ใส่ อยู่ในโครงสร้าง net_device->xmit_lock เพื่อไม่ให้โปรเซสอื่นทำงานในช่วงนี้

```
temp->dev = eth_if;
```

- หลังจากนั้นจะมีการเปลี่ยนความรับผิดชอบโครงสร้าง sk_buff ให้อีกอินเทอร์เฟสเป็นผู้รับผิดชอบ เพื่อใช้ในการส่ง โดยเราจะทำการเปลี่ยนที่ dev ในโครงสร้าง skb_buff

```
eth_if->hard_start_xmit(temp, eth_if);
```

- เมื่อโอนถ่ายความรับผิดชอบแล้ว จะทำการส่งข้อมูลในโครงสร้าง sk_buff นั้นออกโดยใช้ฟังก์ชันที่อยู่ในโครงสร้างของ net_device นั่นคือ hard_start_xmit(struct sk_buff, net_device) ซึ่งอาร์กิวเมนท์ที่ต้องระบุคือโครงสร้างของ sk_buff ที่ต้องการ

ส่ง และโครงสร้างของอุปกรณ์ที่เป็นผู้ทำการส่งข้อมูลออก โดยหากส่งข้อมูลได้สมบูรณ์ ฟังก์ชันดังกล่าวจะรีเทิร์นค่าออกมาเป็น 0 แต่หากส่งไม่สำเร็จจะรีเทิร์นค่าออกมาเป็นต่ำกว่า 0

```
spin_unlock_irq(&eth_if->xmit_lock);
kfree_skb(skb);
```

- หลังจากนั้นจะเป็นการคืนรีซอร์สให้กับระบบ โดยอันดับแรกทำการปลดล๊อกการกันอินเทอร์รัพออก โดยการใช้คำสั่ง spin_unlock_irq(spinlock_t) และจากนั้นทำการคืนทรัพยากรโครงสร้าง sk_buff ที่ถูกส่งออกไปด้วยคำสั่ง kfree_skb()
- จากนั้นทำการรีเทิร์นค่า 0 ออกเพื่อบ่งบอกว่าฟังก์ชันทำงานได้สมบูรณ์

4.2.2. การพัฒนาในส่วนการรับประกันคุณภาพ

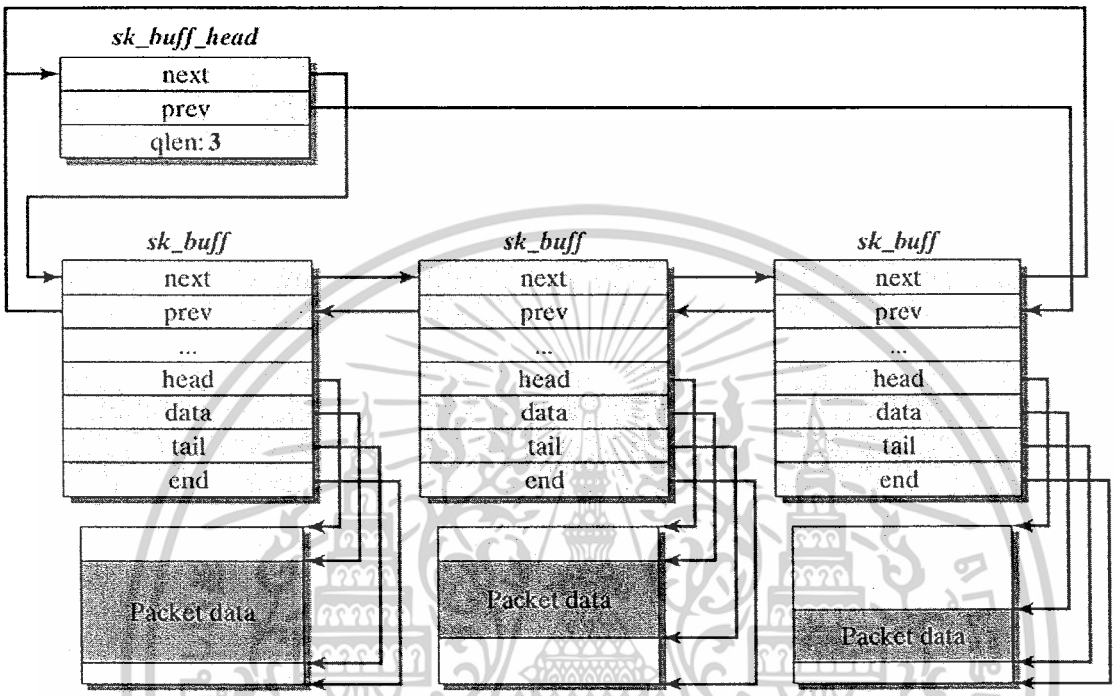
ในช่วงนี้เป็นการพัฒนาโมดูลต่อจาก ส่วนที่แล้วโดยลักษณะของการประกันคุณภาพที่สร้างขึ้นคือมีการสร้างคิวของข้อมูล(โครงสร้าง sk_buff) ซึ่งจะแบ่งคิวออกเป็นจำนวน 4 คิวด้วยกัน ซึ่งความแตกต่างของแต่ละคิวที่ข้อมูลจะไปอาศัยนั้นคือ พอร์ทปลายทางของข้อมูล ซึ่งเมื่อได้รับข้อมูลเข้ามาจากอินเทอร์เฟสไร้สาย เราจะทำการจัดเข้าคิวที่ถูกต้อง คือตามพอร์ทที่ถูกกำหนดไว้ จากนั้นในการส่งข้อมูลออก จะมีอัลกอริทึมในการตัดสินใจว่าคิวใดควรจะถูกเรียกขึ้นมาเพื่อทำการส่งข้อมูลออก โดยในการเรียกคิวขึ้นมาเพื่อจะทำการส่งข้อมูลออก เราได้ทำการใช้ เธร็ด เข้ามาช่วย โดยจะมีการตรวจสอบคิวเป็นระยะว่าคิวแต่ละคิวว่างอยู่หรือไม่ ถ้าไม่ว่างก็ให้ทำการส่งข้อมูลออก ตามอัลกอริทึมที่กำหนดไว้

4.2.2.1. การพัฒนาในส่วนการจัดการคิว

ในการจัดการคิวสามารถแบ่งได้ออกเป็น 2 ส่วน คือ เมื่อได้รับข้อมูลเข้ามาแล้วทำการจัดการให้ข้อมูลเข้าคิวตามที่เรากำหนด และมีการจัดการอีกทีเมื่อต้องการส่งข้อมูลออกจากคิว โดยอัลกอริทึมในการเลือกข้อมูลเพื่อออกจากคิว ในการพัฒนาขั้นตอนนี้เราทำการสร้างโครงสร้างเพื่อใช้ในการจัดการคิวขึ้นโดยโครงสร้างดังกล่าวมีข้อมูลดังนี้

```
struct APQueue{
    int         port[NUMQ]; /* หมายเลขพอร์ทของแต่ละคิว */
    int         percent[NUMQ]; /* น้ำหนักในการส่งข้อมูลจากแต่ละคิว */
    int         wait[NUMQ]; /* เวลาที่ใช้ในการรอการส่งข้อมูล */
    spinlock_t  acc_que; /* ใช้เพื่อป้องกันอินเทอร์รัพในการเข้าใช้คิว */
```

```
        struct sk_buff_headheadq[NUMQ] /* เป็นคิวของโครงสร้าง sk_buff */
}
```



รูปที่ 4.6 แสดงถึงโครงสร้างข้อมูลของตัวแปรชนิด sk_buff_head

โครงสร้างของ sk_buff_head เป็นรูปแบบโครงสร้างในการจัดการคิวแบบง่ายๆ มีการจัดการด้วยลิงค์ลิสต์คู่(Doubly Linked List) โดยมีรูปแบบ ดังรูปที่ 4.6 โดยโครงสร้างนี้มีส่วนประกอบอยู่ 4 ส่วน คือ next เป็นพอยเตอร์ที่ชี้ไปยังโครงสร้าง sk_buff อันแรกของคิว, prev ชี้ไปยังตำแหน่งสุดท้ายของคิว, qlen บอกจำนวนลิงค์ที่อยู่ในคิว และ lock เป็นตัวแปรชนิด spinlock_t ซึ่งเป็นตัวแปรที่ใช้ในการควบคุมการเข้าถึงทรัพยากรพร้อมๆ กันแบบหนึ่ง

โดยเมื่อเริ่มโมดูล(เมื่อมีการแทรกโมดูลลงในระบบ) เราจะทำการสร้างคิวขึ้นมาโดยการใช้ฟังก์ชัน skb_queue_head_init(sk_buff_head) ซึ่งจะทำการสร้างคิวขึ้นมาทั้งหมด 4 คิว ตามจำนวนที่เรากำหนดไว้สำหรับโครงงานนี้ นอกจากสร้างคิวแล้ว ยังทำการใส่ค่าเริ่มต้นให้กับคิวด้วย ตามโค้ดด้านล่างนี้

```
        for(i = 0; i<4 ; i++){
                skb_queue_head_init(apQ.headq[i]);
```

```
}
```

นอกจากนั้นยังต้องกำหนดค่าเริ่มต้นให้กับคิว เพื่อสร้างคุณสมบัติของทั้ง 4 คิวขึ้นมา ได้แก่ หมายเลขพอร์ทของแต่ละคิว, ค่าน้ำหนักของแต่ละคิว และระยะเวลาที่รอได้นานสุดของแต่ละคิว

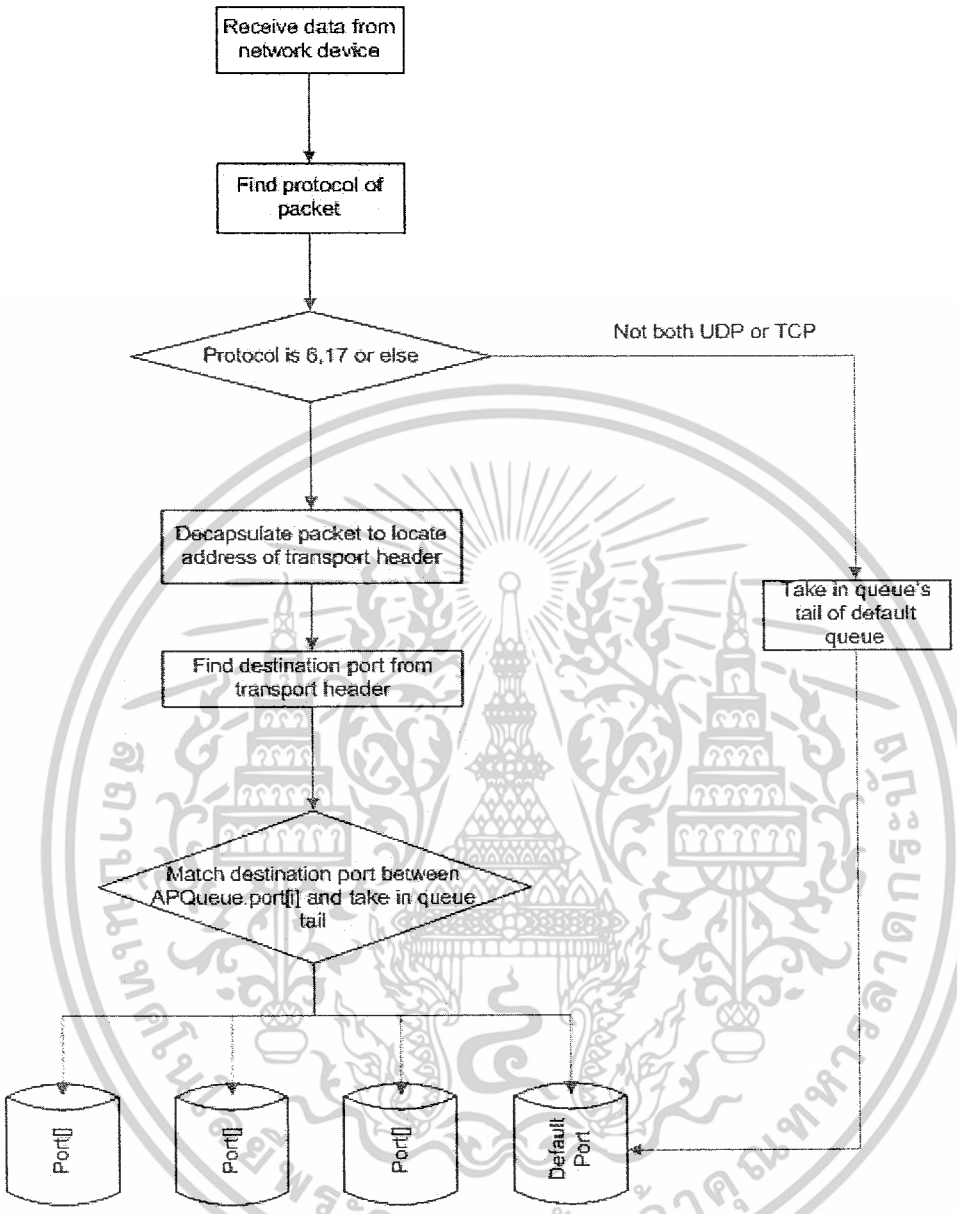
การจัดการคิวเมื่อได้รับข้อมูล

การทำงานในช่วงนี้มีจุดประสงค์เพื่อให้ข้อมูลที่แทนด้วยโครงสร้าง sk_buff จัดเข้าอยู่ในคิวที่แบ่งไปตามพอร์ทปลายทางในระดับทรานสปอร์ทได้สามารถดูรูปประกอบได้ใน รูปที่ 4.7 โดยรายละเอียดขั้นตอนมีดังนี้

- ทำการสำเนาโครงสร้าง sk_buff ที่ได้รับมาและทำการสร้างส่วนเฮดเดอร์ตามขั้นตอนเหมือนโมดูลบริดจ์ปกติ
- จากนั้นจะทำการส่งไปยังฟังก์ชัน classifier(temp) เพื่อให้ทำการจัดเข้าคิวให้ โดยส่งอาร์กิวเมนท์ที่มีพอยเตอร์ชี้ไปยังตำแหน่ง sk_buff ที่สำเนาในฟังก์ชันนี้

```
sktmp->nh.iph=(struct iphdr *)((sktmp->data)+sizeof(struct ethhdr));
port_ip = (int *)sktmp->nh.ph->protocol;
```

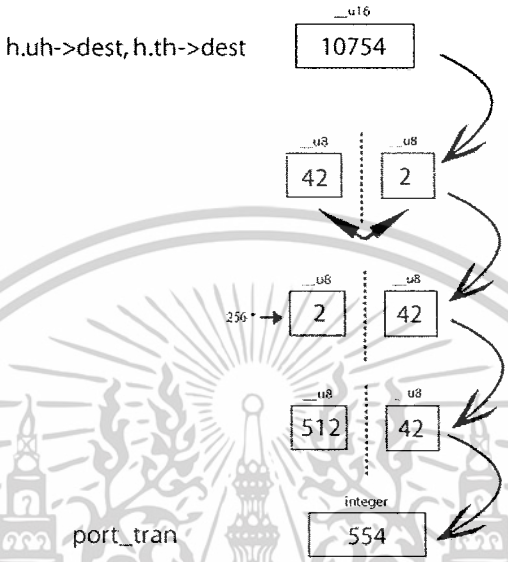
- ใน classified(struct sk_buff *) จะทำการตรวจสอบว่าข้อมูลชุดนี้เป็นข้อมูลประเภท UDP หรือ TCP โดยการตรวจสอบเฮดเดอร์ในระดับเนทเวิร์ค ซึ่งจะมีฟิลด์ที่เรียกว่า protocol โดยมีขนาด 8 บิท(ดูรายละเอียดได้ใน ip.h) โดยหากเป็นทีซีพีโปรโตคอลจะมีค่าเท่ากับ 6 และหากเป็นยูดีพีจะมีค่าเท่ากับ 17 โดยการหาโปรโตคอลในส่วนนี้ก็เพื่อเป็นการหาพอร์ทจะได้ถูกต้อง แต่ก่อนอื่นเราต้องทำการกำหนดตำแหน่งเริ่มต้นของเนทเวิร์คเฮดเดอร์ก่อนว่าอยู่ตำแหน่งใด โดยหาจาก ตำแหน่งของฟิลด์ data ใน sk_buff บวกกับขนาดของอีเธอร์เน็ทเฮดเดอร์



รูปที่ 4.7 แสดงขั้นตอนการใส่ข้อมูลลงคิว

```
int portNum(__u16 *port){
        int port_num;
        port_num = (int)(*((__u8 *)port+1));
        port_num = (((int)(*((__u8 *)port)))*256)+portnum;
}
…
```

```
sktmp->h.th=(struct tcphdr *)((sktmp->data)+sizeof(struct ethhdr)+sizeof(struct iphdr);
port_tran = ((__u16 *)&(skbtmp->h.th->dest));
```



รูปที่ 4.8 แสดงวิธีในการเปลี่ยนพอร์ทปลายทาง

- จากนั้นจะทำการหาพอร์ทที่ใช้ในระดับทรานสปอร์ท โดยทั้งทีซีพีโปรโตคอลและยูดีพีโปรโตคอลจะต้องอ้างอิงจากฟิลด์ที่ชื่อเดียวกันคือ dest โดยก่อนจะอ้างอิงต้องกำหนดตำแหน่งเริ่มต้นของเฮดเดอร์ในระดับทรานสปอร์ทเสียก่อน หลังจากนั้นจะส่งไปยังเมธอด portNum() เพื่อให้ทำการแปลงและส่งค่าพอร์ทกลับมาให้ แต่เนื่องจากหากทำการแปลงเป็นตัวแปรชนิดจำนวนเต็มตรงๆ จะทำให้ค่าผิดเพี้ยนได้ เพราะฉะนั้นจะทำการการแบ่งตัวแปร unsigned word ออกเป็น 8 บิทก่อนหลังจากนั้นทำการแปลงเป็นจำนวนเต็มแล้วเอาเลขที่มีบิทค่าตั้งแต่ 256 ขึ้นไปคูณเข้าด้วย 256 แล้วนำมารวมกันจะได้ ค่าที่เป็นหมายเลขพอร์ทที่ถูกต้อง โดยสามารถดูขั้นตอนได้ตามรูปที่ 4.8

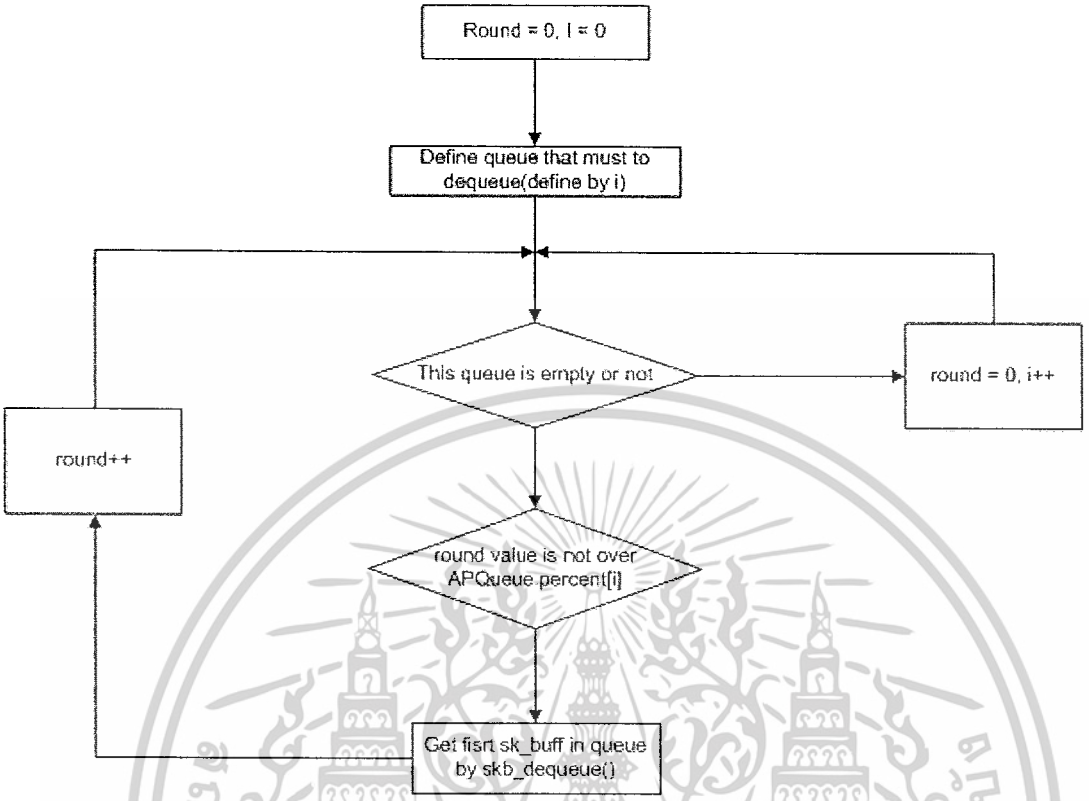
```
if(port_tran==apQ.port[0]){
        spin_lock_irq(&apQ.acc_que);
        skb_queue_tail(&apQ.headq[0], sktmp);
        spin_unlock_irq(&apQ.acc_que);
}
```

- จากนั้นเป็นการจัดคิวโดยนำตัวแปร port_tran มาเทียบกับ apQ.port[] ว่าตรงกับคิวใด หากไม่ตรงกับคิวใดเลยจะเข้าคิวที่มี apQ.port[] ที่เท่ากับศูนย์ และคำสั่งในการเข้าคิวนั้นในที่นี้ใช้คำสั่งต่อท้ายคิวคือ skb_queue_tail(apQ.headq[],sktmp); ซึ่งจะเป็นการนำโครงสร้างไปต่อท้ายสุดของคิว(ใน sk_buff_head->tail จะชี้มายัง sktmp ที่ได้ทำการต่อท้ายเข้าไป) โดยก่อนและหลังทำการเพิ่มข้อมูลลงในคิวเราต้องป้องกันการถูกขัดจังหวะขณะที่ทำการเพิ่มคิวก่อนโดยการใช้ spin_lock_irq(&apQ.acc_que); และยกเลิกการป้องด้วย spin_unlock_irq(&apQ.acc_que);

การจัดการดึงข้อมูลออกจากคิว

ในส่วนของการดึงข้อมูลออกจากคิวนั้น ทำการดึงคิวข้อมูลเมื่อมีการตรวจสอบพบว่าคิวข้อมูลนั้นมีข้อมูลเข้ามาเข้าคิวแล้ว ฟังก์ชันในการดึงข้อมูลก็จะถูกให้ใช้ทำงาน โดยการดึงข้อมูลจะทำการวนรอบทุกๆ คิวในแต่ละรอบ โดยทำการดึงข้อมูลในแต่ละคิวได้จำนวนสูงสุดที่ดึงออกจะคิวได้ เป็นไปตามค่าน้ำหนักที่กำหนดไว้ตั้งแต่เริ่มต้นของโมดูล(bridge_init_module()) แต่หากคิวใดมีจำนวนข้อมูลน้อยกว่าค่าน้ำหนักที่จะดึงคิวออก ก็จะข้ามคิวนั้นไปทันทีหรือหากคิวนั้นไม่มีข้อมูลอยู่ก็จะข้ามการดึงข้อมูลของคิวนั้นไป โดยเมื่อดึงข้อมูลได้ก็จะทำการส่งข้อมูลไปให้อินเทอร์เฟสทันที สามารถดูรูปประกอบได้ รูปที่ 4.9 โดยขั้นตอนการทำงานของการดึงข้อมูลออกจากคิวมีดังนี้

- วนลูปเข้าส่วนการดึงข้อมูลออกจากคิว สร้างตัวแปรสามตัวคือ i เป็นเลขที่ระบุคิวของแต่ละรอบ โดย i เป็นลำดับของคิวที่จะจัดการในรอบนั้น นอกจากนั้นยังเป็นค่าที่ใช้เป็นเงื่อนไขในลูป for() ถัดไปด้วย, round เป็นจำนวนที่รอบที่ทำการวนในพอร์ทนั้นแต่ละครั้ง, check ใช้เพื่อเก็บจำนวนชุดข้อมูลที่คงเหลืออยู่ในคิว
- เข้าเงื่อนไข for(i=0;i<4;i++) เพื่อเป็นการไล่เช็คไปทีละคิว ทั้งหมด 4 คิว คือ 0-3 หลังกจากนั้นทำการกำหนดค่า round ให้แต่ละรอบของการดึงคิวออก โดยค่าเริ่มต้นให้เป็น 0



รูปที่ 4.9 แสดงการถึงข้อมูลออกจากคิว

- จากนั้นจะเข้าเงื่อนไขใน while ลูป โดยเงื่อนไขคือตรวจสอบว่าข้อมูลในคิวที่ i นั้นมีหรือไม่ และจำนวนรอบในการดึงคิวต้องน้อยกว่าค่าน้ำหนักที่ได้กำหนดไว้ โดยทั้งสองเงื่อนไขจะต้องเป็นจริงถึงจะทำการเข้าใน while ลูป
- หากเงื่อนไขจาก while ลูปเป็นจริง จะทำการดึงข้อมูลออกจากคิว โดยก่อนจะทำการถึงข้อมูลออกจากคิวจะต้องทำการใช้ spin_lock_irq() ก่อนเพื่อป้องกันการใช้ทรัพยากรที่ทับซ้อนกัน จากนั้นใช้ฟังก์ชัน skb_dequeue(sk_buff_head) เพื่อทำการดึงข้อมูลออก โดยสิ่งที่จะรีเทิร์นจากฟังก์ชันนี้คือพอยเตอร์ที่ชี้ไปยังตำแหน่งของข้อมูลแรกของคิวนั้น ซึ่งค่า sk_buff_head.next ก็จะเปลี่ยนไป เป็นข้อมูลชุดถัดไปของข้อมูลที่ถูกดึงออกมาจากคิว หลังจากดึงข้อมูลเสร็จทำการปลดล็อกการป้องกันอินเทอร์รัพ
- จากนั้นเข้าสู่ช่วงของการส่งข้อมูลไปยังอุปกรณ์เครือข่ายและคืนพื้นที่ที่ข้อมูลจองอยู่ในตอนแรกให้กับระบบหลังจากนั้นก็จะเข้า while ลูปอีกครั้งหนึ่ง
- หากหลุดออกจาก while ลูปออกมาได้ จะทำการตรวจสอบ ค่า i ว่าเป็นค่าสุดท้ายใช่หรือไม่ หรือพูดอีกนัยหนึ่งคือทำการส่งข้อมูลจากคิวลำดับสุดท้ายรึยัง ถ้าใช่จะเข้าไป

เพื่อทำการตรวจสอบว่ายังมีข้อมูลเหลืออยู่ในคิวเหรือไม่ถ้าเหลือ ก็จะแก้ไขค่า i ให้เป็น -1 เพื่อทำการดึงข้อมูลออกจากคิวอีกรอบหนึ่ง จากนั้นจะเข้า for ลูปอีกครั้งหนึ่งเพื่อตรวจสอบเงื่อนไข

- หากหลุดจากเงื่อนไขใน for() ได้ก็จะทำการกำหนดค่า go_send ให้เป็น 0 อีกครั้ง

โค้ดในการจัดการดึงคิวออกเพื่อส่งข้อมูลต่อไปให้อุปกรณ์เครือข่าย

```
int i;
int round = 0;
int check = 0;
for(i=0;i<4;i++){
        round = 0;
        while(!(skb_queue_empty(&apQ.headq[i]))&&(round<apQ.percent[i])){
                spin_lock_irq(&apQ.acc_que);
                send_skb = skb_dequeue(&(apQ.headq[seq_port]));
                spin_unlock_irq(&apQ.acc_que);
                tmp->dev = eth_if;
                spin_lock_irq(&eth_if->xmit_lock);
                eth_if->hard_start_xmit(tmp, eth_if);
                spin_unlock_irq(&eth_if->xmit_lock);
                kfree_skb(tmp);
                round++;
        }
        if(i==3){
                check = apQ.headq[0].qlen + apQ.headq[1].qlen;
                check = check + apQ.headq[2].qlen + apQ.headq[3].qlen;
                if(check>0){
                        i = -1;
                }
        }
}
```

```
        go_send = 0;
}
```

4.2.2.2 เธร็ด

รูปแบบของเธร็ด นำมาใช้คือมีการวนลูปตลอดเวลาที่เธร็ดทำงานอยู่และจะหน่วงด้วยระยะเวลาหนึ่งเพื่อมาทำการตรวจสอบสถานะของคิว เพื่อให้การทำงานในส่วนของการดึงข้อมูลออกเกิดขึ้น โดยการสร้างเธร็ดขึ้นมาอยู่ในไฟล์ kthread.c, kthread.h (http://www.scs.ch/~frey/linux/kernelthreads.html) โดยฟังก์ชันการทำงานหลักๆ มี 4 ฟังก์ชันคือ

- start_kthread() เป็นฟังก์ชันที่ที่ใช้ในการสรางเธร็ดใหม่โดยเรียกจากฟังก์ชันที่ต้องการสร้างเธรด โดยจะมีการทำ เซมาฟอร์(semaphore) ซึ่งเทคนิคหนึ่งในการป้องกันการเข้าใช้ทรัพยากรร่วมกัน โดยจะป้องกันจนกระทั่ง โครงสร้างเธร็ดใหม่นั้นรันได้
- stop_kthread() เป็นฟังก์ชันที่ทำการกำหนดค่าต่างๆ เพื่อให้เธร็ดเตรียมพร้อมที่จะจบการทำงาน โดยจะส่งสัญญาณ SIGKILL ไปให้เธร็ดซึ่งเมื่อตรวจสอบพบว่าการยกเลิกเป็นการใช้งานเธร็ดเป็นจริงก็จะเรียก stop_thread() อีกทีหนึ่ง
- init_kthread() เป็นเมธอดที่ถูกเรียกเมื่อมีการสร้างเธร็ดใหม่เกิดขึ้น ปันการกำหนดค่าต่างๆ ให้กับเธร็ดเพื่อให้พร้อมกับการทำงาน
- exit_thread() เป็นเมธอดที่ถูกเรียกเมื่อมีการส่งสัญญาณพร้อมที่จะจบการทำงานของเธร็ด โดยถูกเรียกจาก stop_kthread() ด้วยฟังก์ชัน up()

โดยโค้ดของเธร็ดประกอบไปด้วยสองไฟล์ คือ kthread.c และ kthread.h ซึ่งมีรายละเอียดดังนี้

```
/**********************kthread.h************************/
#ifndef _KTHREAD_H
#define _KTHREAD_H
#include <linux/config.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/tqueue.h>
#include <linux/wait.h>
```

```
#include <asm/unistd.h>
#include <asm/semaphore.h>

/*a structure to store all information we need for our thread*/
typedef struct kthread_struct
{
        /*private data*/
/*Linux task structure of thread*/
      struct task_struct *thread;
/* Task queue need to launch thread*/
      struct tq_struct tq;
      /* function to be started as thread*/
      void (*function) (struct kthread_struct *kthread);
      /* semaphoere needed on start and creation of thread. */
      struct semaphore startstop_sem;

      /* public data */
      /* queue thread is waiting on. Gets initialized by init_kthread, can be used by thread itself.   */
      wait_queue_head_t queue;
      /* flag to tell thread whether to die or not. When the thread receives a signal, it must check the
value of terminate and call exit_kthread and terminate if set.   */
      int terminate;
      /* additional data to pass to kernel thread */
      void *arg;
} kthread_t;

/* prototypes */
/* start new kthread (called by creator) */
void start_kthread(void (*func)(kthread_t *), kthread_t *kthread);
```

```
/* stop a running thread (called by "killer") */
void stop_kthread(kthread_t *kthread);
/* setup thread environment (called by new thread) */
void init_kthread(kthread_t *kthread, char *name);

/* cleanup thread environment (called by thread upon receiving termination signal) */
void exit_kthread(kthread_t *kthread);
#endif

/************************kthread.c************************/
#include <linux/config.h>
#include <linux/version.h>
#if defined(MODVERSIONS)
#include <linux/modversions.h>
#endif
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/tqueue.h>
#include <linux/wait.h>
#include <linux/signal.h>
#include <asm/semaphore.h>
#include <asm/smplock.h>
#include "kthread.h"

/* private functions */
static void kthread_launcher(void *data)
{
        kthread_t *kthread = data;
        kernel_thread((int (*)(void *))kthread->function, (void *)kthread, 0);
```

```
}
/* public functions */
/* create a new kernel thread. Called by the creator. */
void start_kthread(void (*func)(kthread_t *), kthread_t *kthread)
{
      /* initialize the semaphore:
        we start with the semaphore locked. The new kernel
        thread will setup its stuff and unlock it. This
        control flow (the one that creates the thread) blocks
        in the down operation below until the thread has reached
        the up() operation.
       */
      init_MUTEX_LOCKED(&kthread->startstop_sem);
      /* store the function to be executed in the data passed to
        the launcher */
      kthread->function=func;
      /* create the new thread my running a task through keventd */
      /* initialize the task queue structure */
      kthread->tq.sync = 0;
      INIT_LIST_HEAD(&kthread->tq.list);
      kthread->tq.routine =  kthread_launcher;
      kthread->tq.data = kthread;
      /* and schedule it for execution */
      schedule_task(&kthread->tq);
      /* wait till it has reached the setup_thread routine */
      down(&kthread->startstop_sem);
}

/* stop a kernel thread. Called by the removing instance */
```

```
void stop_kthread(kthread_t *kthread)
{
    if (kthread->thread == NULL)
    {
        printk("stop_kthread: killing non existing thread!\n");
        return;
    }
    /* this function needs to be protected with the big
       kernel lock (lock_kernel()). The lock must be
      grabbed before changing the terminate
       flag and released after the down() call. */
    lock_kernel();

    /* initialize the semaphore. We lock it here, the
      leave_thread call of the thread to be terminated
      will unlock it. As soon as we see the semaphore
      unlocked, we know that the thread has exited.
      */
    init_MUTEX_LOCKED(&kthread->startstop_sem);

    /* We need to do a memory barrier here to be sure that
      the flags are visible on all CPUs.
    */
    mb();

    /* set flag to request thread termination */
    kthread->terminate = 1;

    /* We need to do a memory barrier here to be sure that
```

```
        the flags are visible on all CPUs.
      */
      mb();
      kill_proc(kthread->thread->pid, SIGKILL, 1);

      /* block till thread terminated */
      down(&kthread->startstop_sem);
      /* release the big kernel lock */
      unlock_kernel();
      /* now we are sure the thread is in zombie state. We
        notify keventd to clean the process up.
      */
      kill_proc(2, SIGCHLD, 1);
}
/* initialize new created thread. Called by the new thread. */
void init_kthread(kthread_t *kthread, char *name)
{
      /* lock the kernel. A new kernel thread starts without
        the big kernel lock, regardless of the lock state
        of the creator (the lock level is *not* inheritated)
      */
      lock_kernel();

      /* fill in thread structure */
      kthread->thread = current;

      /* set signal mask to what we want to respond */
      siginitsetinv(&current->blocked, sigmask(SIGKILL)|sigmask(SIGINT)|sigmask(SIGTERM));
```

```
    /* initialise wait queue */
    init_waitqueue_head(&kthread->queue);

    /* initialise termination flag */
    kthread->terminate = 0;

    /* set name of this process (max 15 chars + 0 !) */
    sprintf(current->comm, name);

    /* let others run */
    unlock_kernel();

    /* tell the creator that we are ready and let him continue */
    up(&kthread->startstop_sem);
}
/* cleanup of thread. Called by the exiting thread. */
void exit_kthread(kthread_t *kthread)
{
    /* we are terminating */
      /* lock the kernel, the exit will unlock it */
    lock_kernel();
    kthread->thread = NULL;
    mb();
    /* notify the stop_kthread() routine that we are terminating. */
      up(&kthread->startstop_sem);
      /* the kernel_thread that called clone() does a do_exit here. */
      /* there is no race here between execution of the "killer" and real termination
        of the thread (race window between up and do_exit), since both the
        thread and the "killer" function are running with the kernel lock held.
```

```
        The kernel lock will be freed after the thread exited, so the code
        is really not executed anymore as soon as the unload functions gets
        the kernel lock back.
        The init process may not have made the cleanup of the process here,
        but the cleanup can be done safely with the module unloaded.
    */
}
```

ในการใช้เธร็ดของโมดูลเรามีจุดประสงค์เพื่อให้ตรวจสอบคิวอยู่สม่ำเสมอ ว่าคิวมีข้อมูลอยู่หรือไม่เพื่อจะได้ทำการส่งข้อมูลออกจากคิวต่อไป โดยในฟังก์ชันเริ่มต้นโมดูลเราทำการสร้างเธร็ดขึ้นมา เพื่อให้เริ่มต้นทำงานเลยโดยใช้คำสั่ง start_kthread() โดยการเรียกฟังก์ชันนี้เราจะส่งอาร์กิวเมนท์ไปพร้อมด้วยกัน 2 ตัวคือ ฟังก์ชันที่จะใช้ในการทำเธร็ด(ในโครงงานของเรา คือฟังก์ชันที่ทำการตรวจสอบคิว) โดยฟังก์ชันนั้นจะต้องมีพารามิเตอร์พอยเตอร์ของโครงสร้าง kthread_t ด้วย ส่วนอาร์กิวเมนต์อีกหนึ่งตัวคือโครงสร้าง kthread_t โดยเราเรียกใช้ตามโค้ดด้านล่างนี้

```
struct kthread_t thread;
start_kthread(do_thread, &thread);
```

โดยในโมดูลของโครงงานนี้มีฟังก์ชันที่ทำงานเป็นคอยตรวจสอบปริมาณของข้อมูลในคิว คือฟังก์ชัน do_thread(struct kthread *) ซึ่งภายในจะมีการเรียก init_kthread() เพื่อกำหนดค่าการทำงานของโมดูล และเมื่อจบฟังก์ชันนี้จะมีการเรียก exit_kthread() เพื่อเป็นการยกเลิกเธร็ด โดยรูปแบบการทำงานของฟังก์ชันจะมีลักษณะ วนลูปไม่รู้จบ ด้วย for(;;) เพื่อคอยตรวจสอบปริมาณของคิวอยู่ทุกช่วงเวลาหนึ่ง โดยในแต่ละรอบของการทำงานจะมีการหน่วงเวลาให้เธร็ดนั้นหลับไปเป็นเวลาหนึ่งและถึงจะตรวจสอบปริมาณคิวของข้อมูล ซึ่งการตรวจสอบคิวทำโดยใช้คำสั่ง skb_queue_empty() โดยหากคิวว่างอยู่จะทำการรีเทิร์นค่าออกมาเป็นจริง(true) แต่หากคิวมีข้อมูลอยู่จะรีเทิร์นค่ากลับออกมาเป็นเท็จ(false) ซึ่งเราตรวจสอบโดย

```
if(!skb_queue_empty(apQ.headq[0]){
        /* first queue is not empty */
```

```
            go_send = 1; // กำหนดให้ทำการส่งข้อมูลออกจากคิว
        }
```

โดยจะลูปในฟังก์ชัน do_thread() จะหลุดออกจากลูปได้ก็ต่อเมื่อมีการส่งสัญญาณให้ทำการยกเลิกเธร็ดที่สร้างขึ้น ทำให้เงื่อนไขด้านล่างต่อไปนี้เป็นจริง จึงหลุดออกจากลูป for(;;)

```
    if(kthread->terminate){
        /* เข้าลูปนี้ก็ต่อเมื่อมีการส่งสัญญาณจาก exit_module() */
        break;
    }
```

ซึ่งเราจะส่งสัญญาณยกเลิกก็ต่อเมื่อเรียกฟังก์ชัน stop_kthread() ซึ่งในโมดูลที่ทำการพัฒนาจะเรียก stop_kthread() เมื่อต้องการยกเลิกโมดูลหรือในฟังก์ชัน birdge_exit_module()

4.3 โปรแกรมในส่วนของการปรับแต่งการทำงานแอ็คเซสพอยต์

4.3.1 ส่วนที่ทำงานอยู่ในระดับยูเซอร์ (User Space)

โปรแกรมที่ทำงานอยู่ในระดับของยูเซอร์จะทำการพัฒนาโดยใช้ภาษา PHP และ HTML โดยให้แสดงผลออกมาติดต่อกับผู้ใช้ในรูปของเว็บเพจ โดยผู้ใช้สามารถเรียกดูสถานะและปรับแต่งค่าต่างๆ ของแอ็คเซสพอยต์ได้ เช่น แม็คแอ็ดเดรส, SSID, ช่องสัญญาณ เป็นต้น

4.3.2 แนะนำภาษา PHP

ในช่วงแรกภาษาที่นิยมใช้ในการทำงานของระบบเครือข่ายคือ HTML (Hypertext Markup Language) แต่ภาษา HTML เป็นภาษาประเภทสแตติก (คือภาษาที่ใช้สร้างข้อมูลประเภทตัวอักษร รูปภาพ โดยข้อมูลนั้นจะไม่สามารถเปลี่ยนแปลงได้) ต่อมาจึงได้มีการพัฒนาภาษาที่เป็นไดนามิก (คือภาษาที่ข้อมูลจะถูกเปลี่ยนแปลงอัตโนมัติตามเงื่อนไขที่ผู้เขียนโปรแกรมกำหนด) หนึ่งในนั้นคือภาษาสคริปต์ PHP

4.3.3 การใช้งาน PHP ร่วมกับ HTML

เราสามารถทำงานแทรกโค๊ดของ PHP ลงใน HTML ได้เลย แต่ต้องอยู่ภายในแท็กของ PHP โค๊ดของภาษา PHP จะอยู่ภายใต้เครื่องหมาย <? และจบด้วย ?> ซึ่งเรียกว่า Short style จะแตกต่างจากโค๊ดของ HTML ที่จะใช้เครื่องหมาย < และจบด้วย > แท็กของ PHP เป็นตัวบอกเว็ปเซิร์ฟเวอร์ของ PHP ว่าโค๊ดของภาษาPHP เริ่มต้นและสิ้นสุดตรงไหน ตัวอย่างของ PHP Tag แบบ Short Style

```
<? Echo "My program PHP. <BR>" ; ?>
```

4.3.4 ตัวอย่างฟังก์ชันของ PHP ที่ใช้ในการสร้างโปรแกรมติดต่อกับผู้ใช้

String Shell_exec(String cmd)

- เป็นฟังก์ชันที่ใช้ในการ execute คำสั่งผ่านทาง shell แล้วส่งผลลัพธ์ที่ได้จากการ execute กลับมาเป็นสตริง เช่น

$wire = shell_exec('ifconfig');

- เป็นสคริปต์ที่ให้ไป execute คำสั่ง ifconfig บน shell แล้วนำผลลัพธ์ที่ได้จากคำสั่ง ifconfig ไปเก็บไว้ในตัวแปร $wire เพื่อนำไปใช้ต่อไป

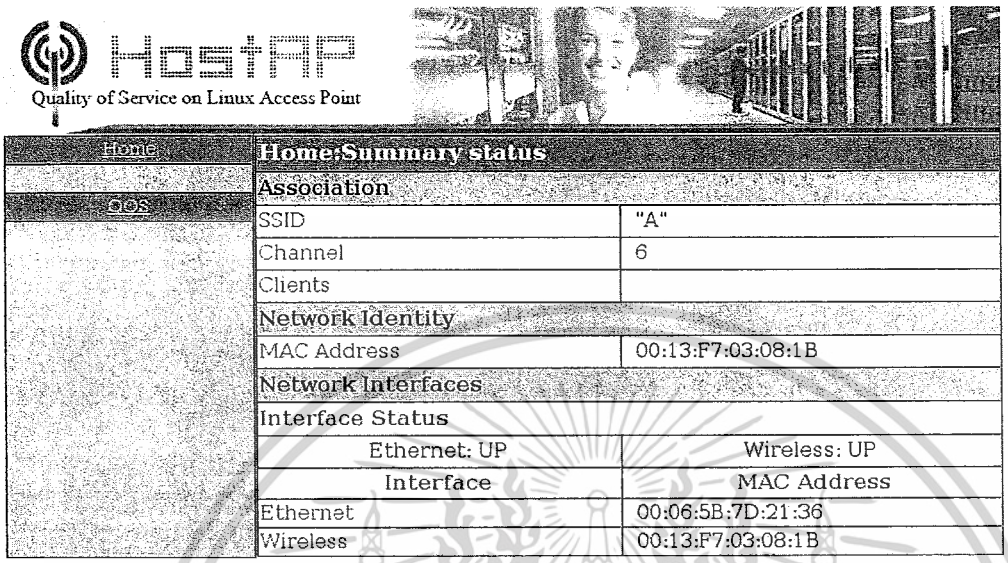
int ereg (string pattern, string string [, array regs[])

- เป็นฟังก์ชันที่ใช้สำหรับตรวจสอบข้อความและตัวอักษร ฟังก์ชันนี้จะตีความตัวอักษรพิมพ์ใหญ่และพิมพ์เล็กแตกต่างกัน เช่น

```
if(ereg("UP",$wire,$tmp))
{
        ..............
}
```

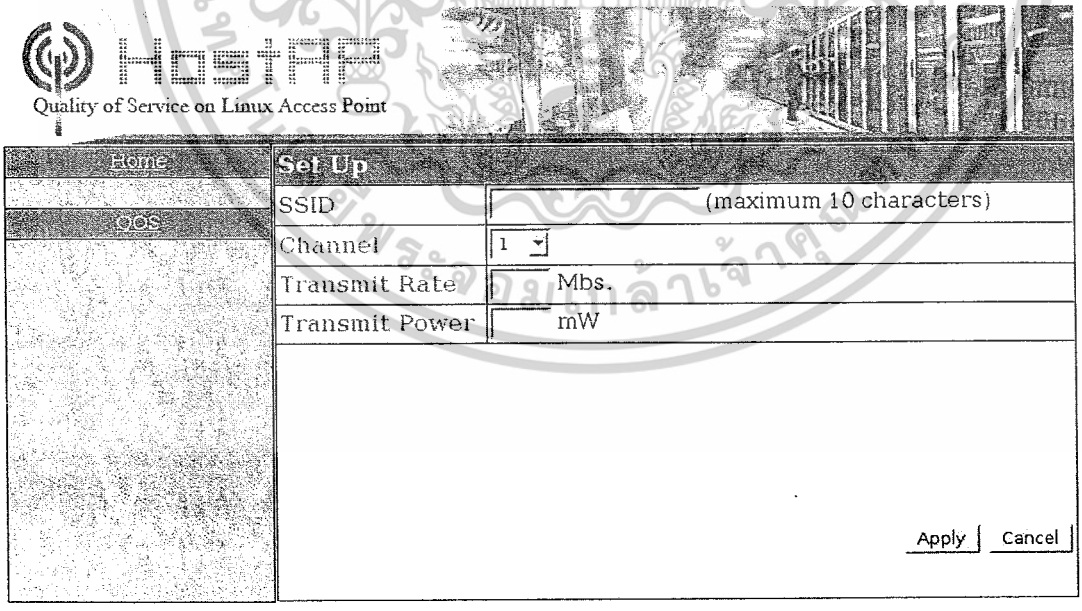
จากตัวอย่างด้นบนการใช้ฟังก์ชัน ereg() เป็นการไปค้นหาคำว่า "UP" ในตัวแปร $wire (จากตัวอย่างที่แล้ว $wire เก็บผลลัพธ์ที่ได้จากการใช้คำสั่ง ifconfig) หากในตัวแปร $wire มีคำว่า "UP" อยู่ โปรแกรมก็จะเข้าไปทำงานตามเงื่อนไขต่อไป

4.3.5 ตัวอย่างหน้าจอโปรแกรม



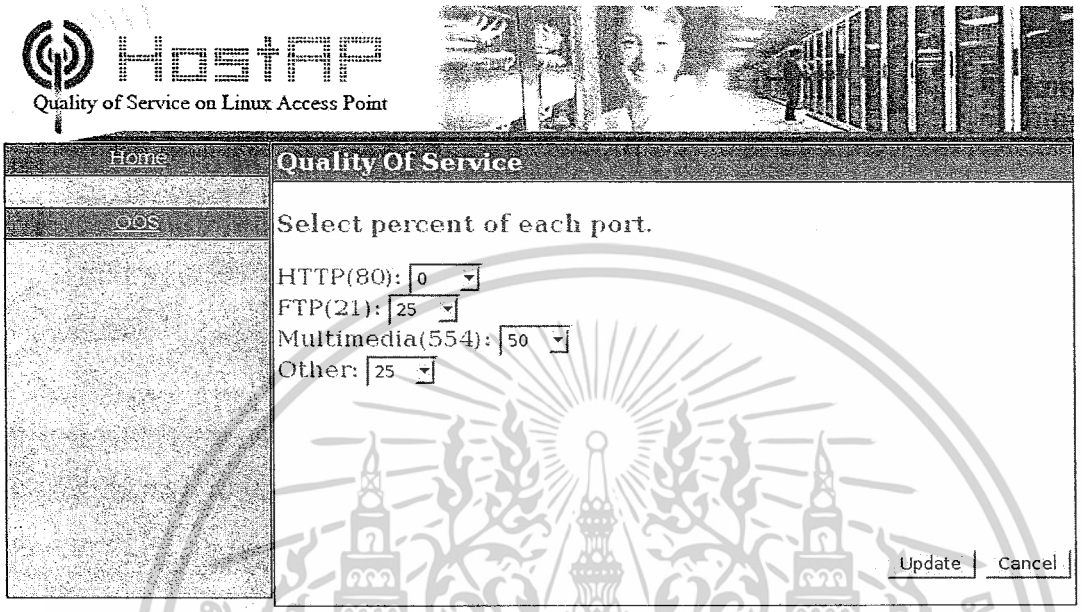
รูปที่ 4.10 หน้าจอดูสถานะต่างๆของแอ็กเซสพอยต์

รูปที่ 4.10 เป็นหน้าจอของโปรแกรมที่เอาไว้สำหรับดูค่าสถานะที่สำคัญต่างๆ ของแอ็กเซสพอยต์ อาทิเช่น SSID, ช่องสัญญาณ, แมคแอดเดรสของแอ็กเซสพอยต์, แมคแอดเดรสของเน็ตเวิร์กอินเตอร์เฟสการ์ด



รูปที่ 4.11 หน้าจอที่ใช้ในการปรับแต่งค่าของแอ็กเซสพอยต์

รูปที่ 4.11 เป็นหน้าจอในส่วนที่เอาไว้สำหรับปรับแต่งค่าการทำงานของแอ็กเซสพอยต์ เช่น SSID, ช่องสัญญาณที่ใช้ในการส่ง, ความเร็วในการส่งข้อมูล, กำลังที่ใช้ในการส่ง



รูปที่ 4.12 หน้าจอในส่วนของการรับประกันคุณภาพ

รูปที่ 4.12 เป็นส่วนของโปรแกรมที่ให้ผู้ใช้เลือกทำการปรับแต่งค่าการส่งข้อมูลของแต่ละพอร์ต

บทที่ 5
ผลการทดลอง

เนื้อหาในบทนี้จะกล่าวถึง การทดสอบใช้งานไวร์เลสแอ็คเซสพอยต์และระบบรับประกันคุณภาพที่สร้างขึ้น รายละเอียดการทดสอบใช้งานนำเสนอดังนี้

5.1 อุปกรณ์ที่ใช้ในการทดลอง

- เครื่องคอมพิวเตอร์ที่ทำหน้าที่เป็นแอ็คเซสพอยต์จำนวน 1 เครื่อง
    - รายละเอียดทางด้านเทคนิค
        - ซีพียู Intel® Pentium® 4 CPU 1.60 GHz.
        - หน่วยความจำ 512 MHz. Bus 400 MHz.
        - ฮาร์ดดิสก์ 40 GB.
- เครื่องคอมพิวเตอร์ที่ติดตั้งไวร์เลสการ์ด จำนวน 3 เครื่อง
- เครื่องคอมพิวเตอร์ที่ติดตั้งอีเทอร์เน็ต จำนวน 1 เครื่อง
- สวิซต์ 1 ตัว
- สายแลน จำนวน 2 เส้น

5.2 การจัดเตรียมด้านฮาร์ดแวร์

- ติดตั้งการ์ดไวร์เลสและอีเทอร์เน็ตการ์ดเข้ากับเครื่องคอมพิวเตอร์ที่จะทำงานเป็นแอ็คเซสพอยต์
- ทำการเชื่อมต่อเครือข่ายดังรูปการทดลอง

5.3 การจัดเตรียมด้านซอฟต์แวร์

- ติดตั้งระบบปฏิบัติการ Linux Debian ให้กับเครื่องคอมพิวเตอร์ที่ทำงานเป็นแอ็คเซสพอยต์
- ติดตั้งโปรแกรม madwifi พร้อมกับทำการปรับแต่งให้กับเครื่องคอมพิวเตอร์ที่ทำงานเป็นแอ็คเซสพอยต์

- ติดตั้งโปรแกรม LanTraffic V2 ให้กับเครื่องคอมพิวเตอร์ที่เป็นไคลเอนท์ทั้งฝั่งเครือข่ายอีเทอร์เน็ตและเครือข่ายไร้สาย

5.4 วิธีการทดลอง

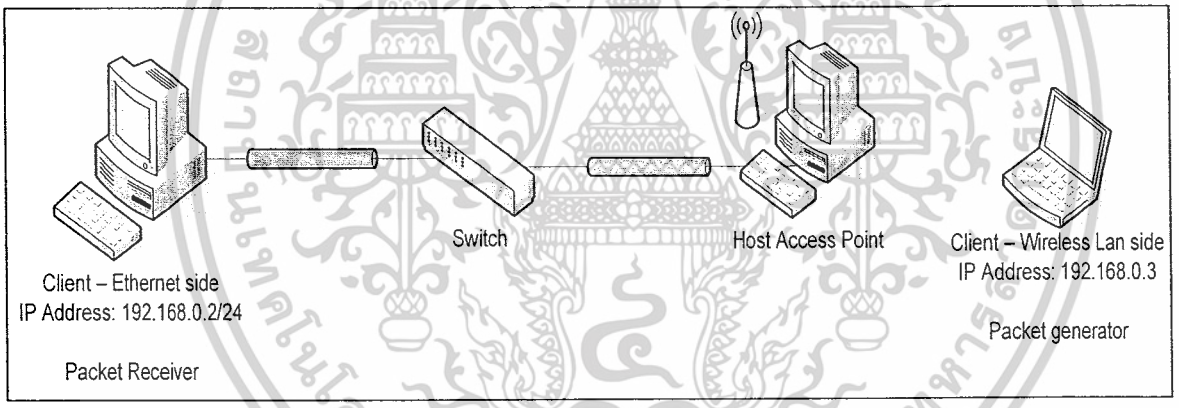
5.4.1 การทดลองที่ 1

5.4.1.1 กำหนดค่าอัตราส่วนน้ำหนักของข้อมูลในแต่ละพอร์ตที่จะถูกส่งผ่านแอ็คเซสพอยต์ตามตารางที่ 5.1

5.4.1.2 เปิดโปรแกรม LanTraffic V2 บนเครื่องเซิร์ฟเวอร์และไคลเอนท์ แล้วทำการตั้งค่าเพื่อเตรียมทำการรับ-ส่งข้อมูลตามพอร์ตที่กำหนด

5.4.1.3 ทำการรับ-ส่งข้อมูลระหว่างเครื่องไคลเอนท์และเซิร์ฟเวอร์โดยผ่านแอ็คเซสพอยต์

5.4.1.4 ทำการบันทึกผลที่ได้



รูปที่ 5.1 รูปการทดลองที่ 1

ตารางที่ 5.1 อัตราส่วนน้ำหนักของข้อมูลของแต่ละพอร์ตในแต่การทดลอง

หมายเลขพอร์ต(UDP)	ค่าน้ำหนัก (%)		
	การทดลองที่ 1.1	การทดลองที่ 1.2	การทดลองที่ 1.3
80	9	55	25
23	18	22	25
42	27	11	25
53 และ 71	45	11	25

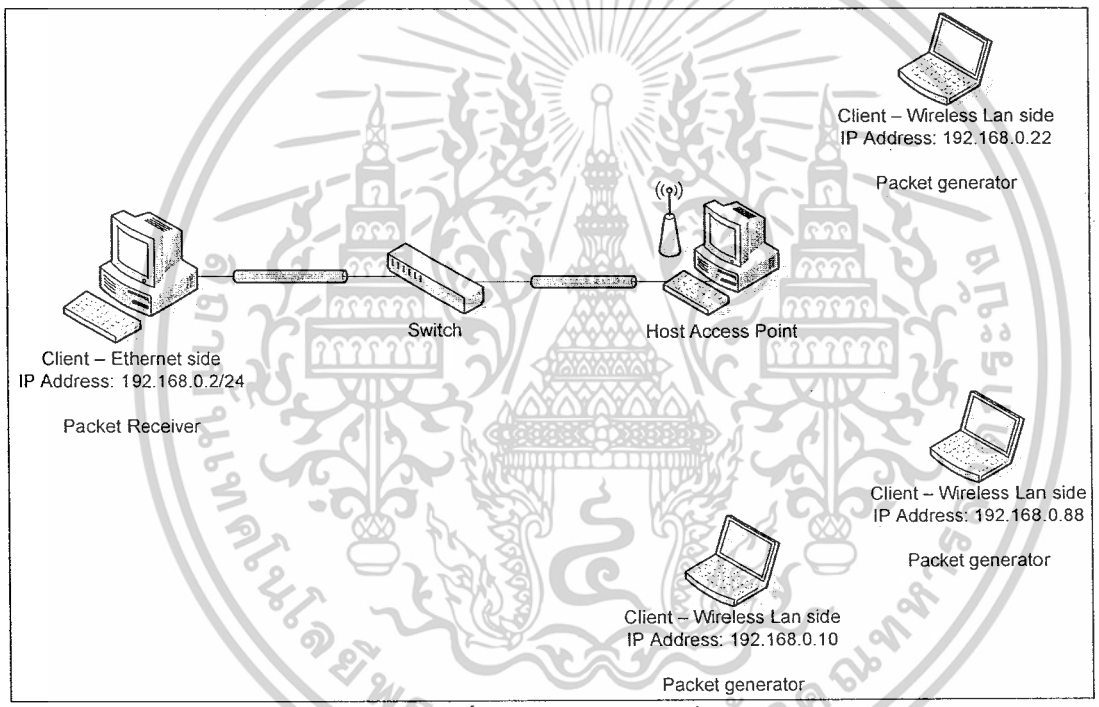
5.4.2 การทดลองที่ 2

5.4.2.1 กำหนดค่าอัตราส่วนน้ำหนักของข้อมูลในแต่ละพอร์ตที่จะถูกส่งผ่านแอ็คเซสพอยต์ตามตารางที่ 5.2

5.4.2.2 เปิดโปรแกรม TrafficLan V2 ที่เครื่องไคลเอนท์ฝั่งเครือข่ายไร้สายแล้วทำการสร้างข้อมูลที่จะส่งให้มีอัตราทรูพุธตามตารางที่ 5.3

5.4.2.3 ทำการรับ-ส่งข้อมูล ซึ่งในแต่ละการทดลองแบ่งออกเป็น 2 ส่วน คือ แอคเซสพอยท์มีระบบรับประกันคุณภาพและไม่มีระบบรับประกันคุณภาพ

5.4.2.4 บันทึกและวิเคราะห์ผล



รูปที่ 5.2 รูปการทดลองที่ 2

ตารางที่ 5.2 ค่าน้ำหนักที่กำหนดให้แต่ละพอร์ทตลอดทั้งการทดลอง

พอร์ท	ค่าน้ำหนักของแต่ละคิว
554	37 %
20	18.5%
60	18.5%
other	26%

ตารางที่ 5.3 อัตราทรูพุธที่ใช้ในการส่งของแต่ละพอร์ทในแต่ละการทดลอง

ทดลองครั้งที่	Throughput ที่ทำการกำหนดใน trafficLan V2 (Mb/s)				
	UDP:554	TCP:20	TCP:80	UDP:90	UDP:100
1	4	4	1	4	3
2	3	3	0.75	3	2.25
3	2	2	0.5	2	1.5
4	1	1	0.25	1	0.75

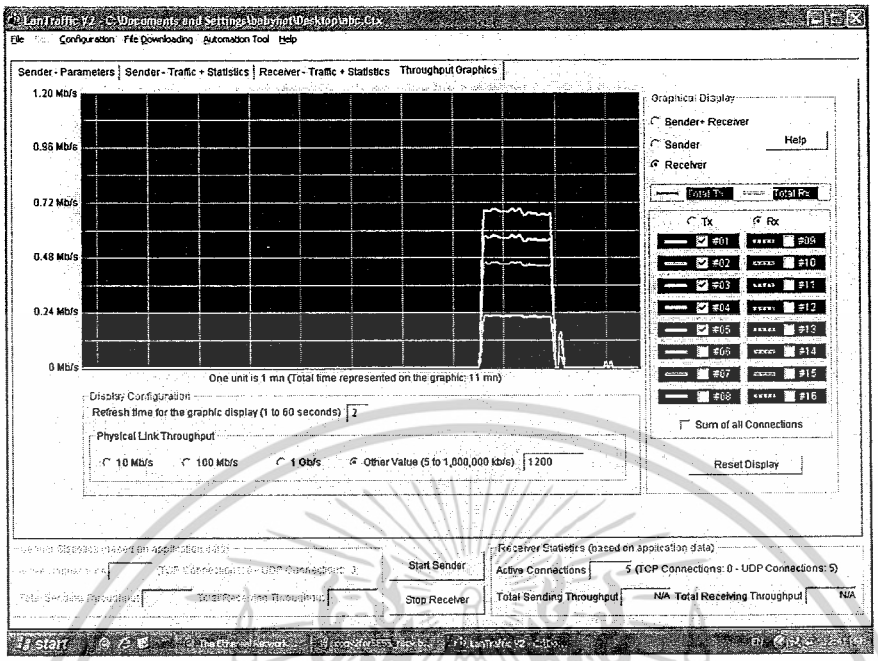
5.5 ผลการทดลอง

5.5.1 ผลการทดลองที่ 1.1

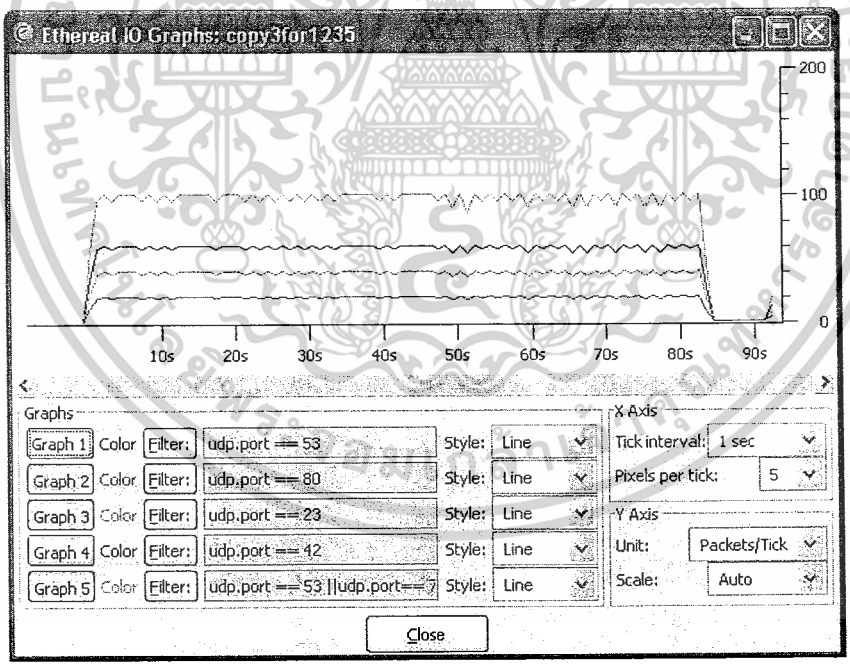
ตัวอย่างแพ็กเก็ตที่ผ่านการจัดการจากแอ็คเซสพอยต์ในการทดลองที่ 1.1

No.	Time	Source	Destination	Protocol	Info
8	1.453307	192.168.1.10	192.168.1.2	UDP	Source port: 1727 Destination port: 23
9	1.453437	192.168.1.10	192.168.1.2	UDP	Source port: 1728 Destination port: nameserver
10	1.453551	192.168.1.10	192.168.1.2	UDP	Source port: 1728 Destination port: nameserver
11	1.453672	192.168.1.10	192.168.1.2	UDP	Source port: 1728 Destination port: nameserver
12	1.453794	192.168.1.10	192.168.1.2	UDP	Source port: 1730 Destination port: 71
13	1.453916	192.168.1.10	192.168.1.2	DNS	
14	1.454039	192.168.1.10	192.168.1.2	UDP	Source port: 1730 Destination port: 71
15	1.454161	192.168.1.10	192.168.1.2	DNS	
16	1.454282	192.168.1.10	192.168.1.2	UDP	Source port: 1730 Destination port: 71
17	1.506264	192.168.1.10	192.168.1.2	UDP	Source port: 1726 Destination port: 80

18 1.506367	192.168.1.10	192.168.1.2	UDP	Source port: 1727 Destination port: 23
19 1.506487	192.168.1.10	192.168.1.2	UDP	Source port: 1727 Destination port: 23
20 1.506611	192.168.1.10	192.168.1.2	UDP	Source port: 1728 Destination port: nameserver
21 1.506732	192.168.1.10	192.168.1.2	UDP	Source port: 1728 Destination port: nameserver
22 1.506853	192.168.1.10	192.168.1.2	UDP	Source port: 1728 Destination port: nameserver
23 1.506976	192.168.1.10	192.168.1.2	DNS	
24 1.507098	192.168.1.10	192.168.1.2	DNS	
25 1.507220	192.168.1.10	192.168.1.2	UDP	Source port: 1730 Destination port: 71
26 1.507343	192.168.1.10	192.168.1.2	UDP	Source port: 1730 Destination port: 71
27 1.507464	192.168.1.10	192.168.1.2	DNS	
28 1.553055	192.168.1.10	192.168.1.2	UDP	Source port: 1726 Destination port: 80
29 1.553148	192.168.1.10	192.168.1.2	UDP	Source port: 1727 Destination port: 23
30 1.553267	192.168.1.10	192.168.1.2	UDP	Source port: 1727 Destination port: 23
31 1.553391	192.168.1.10	192.168.1.2	UDP	Source port: 1728 Destination port: nameserver
32 1.553513	192.168.1.10	192.168.1.2	UDP	Source port: 1728 Destination port: nameserver
33 1.553634	192.168.1.10	192.168.1.2	UDP	Source port: 1728 Destination port: nameserver
34 1.553756	192.168.1.10	192.168.1.2	UDP	Source port: 1730 Destination port: 71
35 1.553879	192.168.1.10	192.168.1.2	DNS	
36 1.554009	192.168.1.10	192.168.1.2	UDP	Source port: 1730 Destination port: 71
37 1.554123	192.168.1.10	192.168.1.2	DNS	
38 1.554243	192.168.1.10	192.168.1.2	UDP	Source port: 1730 Destination port: 71
39 1.602473	192.168.1.10	192.168.1.2	UDP	Source port: 1726 Destination port: 80
40 1.602578	192.168.1.10	192.168.1.2	UDP	Source port: 1727 Destination port: 23



รูปที่ 5.3 กราฟแสดงทรูพุธในแต่ละพอร์ตที่ได้จากโปรแกรม LanTraffic V2 ในการทดลองที่ 1.1



รูปที่ 5.4 กราฟแสดงทรูพุธในแต่ละพอร์ตที่ได้จากโปรแกรม Ethereal ในการทดลองที่ 1.1

ตารางที่ 5.4 ตารางแสดงค่าทรูพุธของแต่ละพอร์ตในการทดลองที่ 1.1

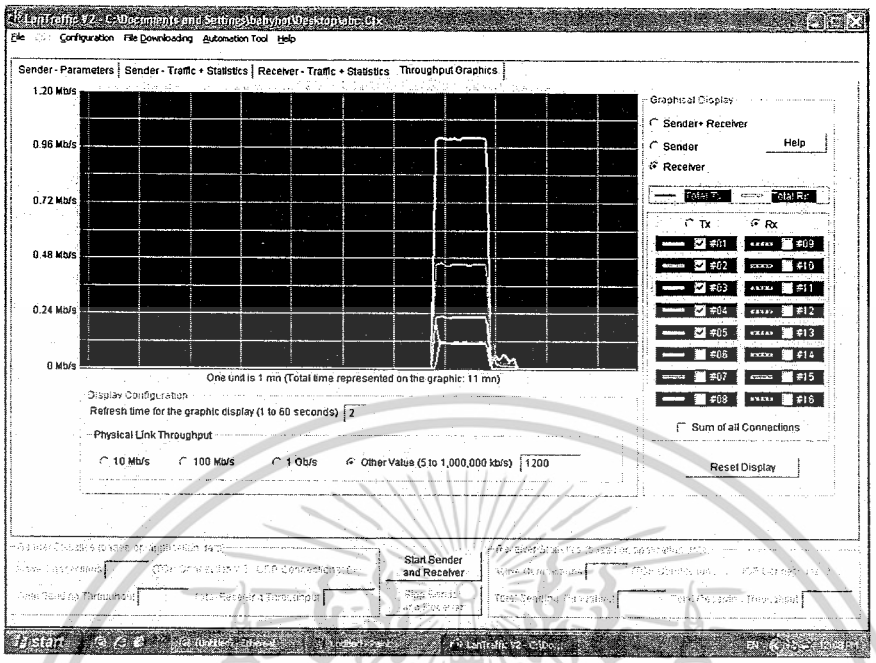
หมายเลขพอร์ต(UDP)	ค่าน้ำหนัก (%)	ทรูพุธ (Throughput) (Mbps)
80	9	0.211
23	18	0.422
42	27	0.634
53 และ 71	45	1.056

5.5.2 ผลการทดลองที่ 1.2

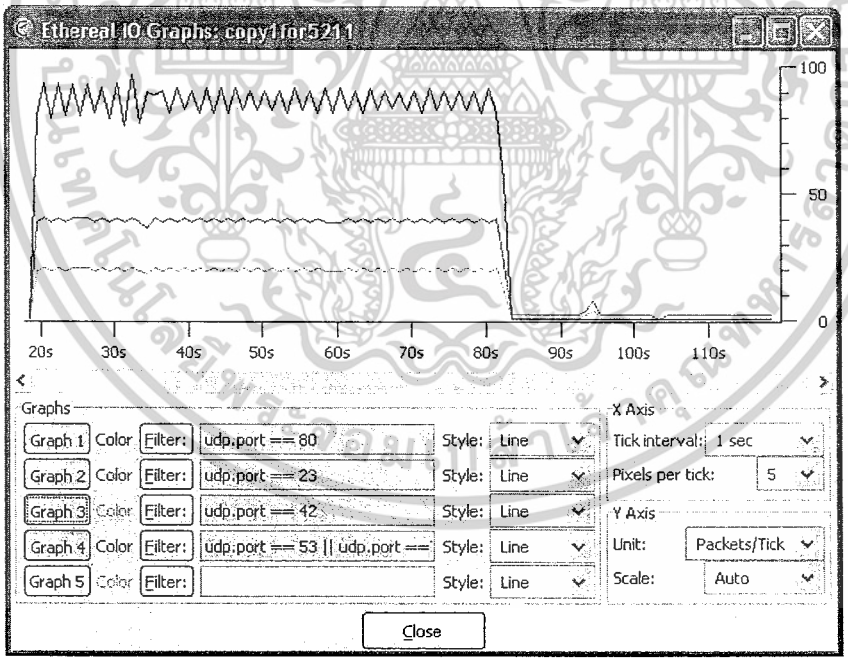
ตัวอย่างแพ็กเก็ตที่ผ่านการจัดการจากแอ็คเซสพอยต์ในการทดลองที่ 1.2

No.	Time	Source	Destination	Protocol	Info
16	18.649758	192.168.1.10	192.168.1.2	UDP	Source port: 1679 Destination port: 80
17	18.649853	192.168.1.10	192.168.1.2	DNS	
18	18.697193	192.168.1.10	192.168.1.2	UDP	Source port: 1679 Destination port: 80
19	18.697312	192.168.1.10	192.168.1.2	UDP	Source port: 1679 Destination port: 80
20	18.697433	192.168.1.10	192.168.1.2	UDP	Source port: 1679 Destination port: 80
21	18.697554	192.168.1.10	192.168.1.2	UDP	Source port: 1679 Destination port: 80
22	18.697677	192.168.1.10	192.168.1.2	UDP	Source port: 1680 Destination port: 23
23	18.697798	192.168.1.10	192.168.1.2	UDP	Source port: 1680 Destination port: 23
24	18.697921	192.168.1.10	192.168.1.2	UDP	Source port: 1681 Destination port: nameserver
25	18.698043	192.168.1.10	192.168.1.2	DNS	
26	18.753414	192.168.1.10	192.168.1.2	UDP	Source port: 1679 Destination port: 80
27	18.753528	192.168.1.10	192.168.1.2	UDP	Source port: 1679 Destination port: 80
28	18.753651	192.168.1.10	192.168.1.2	UDP	Source port: 1679 Destination port: 80
29	18.753774	192.168.1.10	192.168.1.2	UDP	Source port: 1679 Destination port: 80
30	18.753895	192.168.1.10	192.168.1.2	UDP	Source port: 1679 Destination port: 80
31	18.754016	192.168.1.10	192.168.1.2	UDP	Source port: 1680 Destination port: 23
32	18.754138	192.168.1.10	192.168.1.2	UDP	Source port: 1680 Destination port: 23
33	18.754261	192.168.1.10	192.168.1.2	UDP	Source port: 1681 Destination port: nameserver
34	18.754387	192.168.1.10	192.168.1.2	DNS	
35	18.799191	192.168.1.10	192.168.1.2	UDP	Source port: 1679 Destination port: 80

36	18.799290	192.168.1.10	192.168.1.2	UDP	Source port: 1679 Destination port: 80
37	18.799412	192.168.1.10	192.168.1.2	UDP	Source port: 1679 Destination port: 80
38	18.799533	192.168.1.10	192.168.1.2	UDP	Source port: 1679 Destination port: 80
39	18.799655	192.168.1.10	192.168.1.2	UDP	Source port: 1680 Destination port: 23
40	18.799776	192.168.1.10	192.168.1.2	UDP	Source port: 1680 Destination port: 23
41	18.799899	192.168.1.10	192.168.1.2	UDP	Source port: 1681 Destination port: nameserver
42	18.800023	192.168.1.10	192.168.1.2	DNS	
43	18.847041	192.168.1.10	192.168.1.2	UDP	Source port: 1679 Destination port: 80
44	18.847143	192.168.1.10	192.168.1.2	UDP	Source port: 1679 Destination port: 80
45	18.847267	192.168.1.10	192.168.1.2	UDP	Source port: 1679 Destination port: 80
46	18.847389	192.168.1.10	192.168.1.2	UDP	Source port: 1679 Destination port: 80
47	18.847510	192.168.1.10	192.168.1.2	UDP	Source port: 1680 Destination port: 23
48	18.847630	192.168.1.10	192.168.1.2	UDP	Source port: 1680 Destination port: 23
49	18.847754	192.168.1.10	192.168.1.2	UDP	Source port: 1681 Destination port: nameserver
50	18.847875	192.168.1.10	192.168.1.2	DNS	
51	18.943410	192.168.1.10	192.168.1.2	UDP	Source port: 1680 Destination port: 23



รูปที่ 5.5 กราฟแสดงทรูพุธในแต่ละพอร์ตที่ได้จากโปรแกรม LanTraffic V2 ในการทดลองที่ 1.2



รูปที่ 5.6 กราฟแสดงทรูพุธในแต่ละพอร์ตที่ได้จากโปรแกรม Ethereal ในการทดลองที่ 1.2

ตารางที่ 5.5 ตารางแสดงค่าทรูพุธของแต่ละพอร์ตในการทดลองที่ 1.2

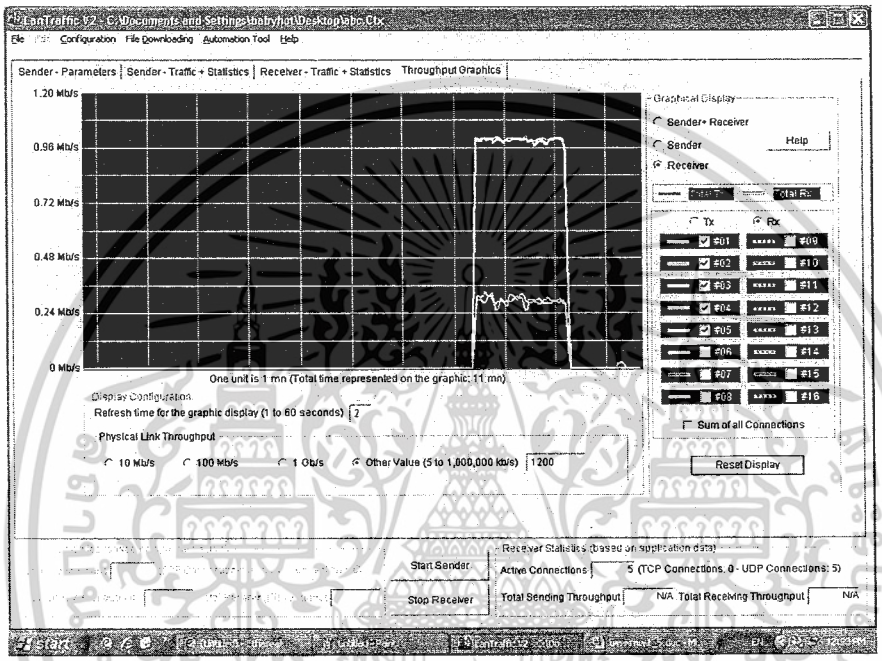
หมายเลขพอร์ต(UDP)	ค่าน้ำหนัก (%)	ทรูพุธ (Throughput) (Mbps)
80	55	1.033
23	22	0.311
42	11	0.156
53 และ 71	11	0.156

5.5.3 ผลการทดลองที่ 1.3

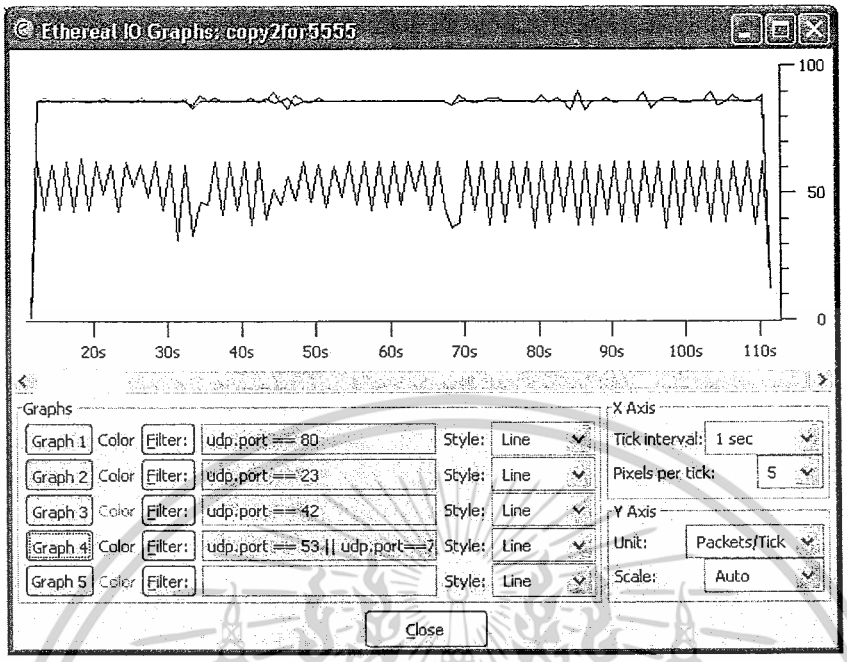
ตัวอย่างแพ็กเก็ตที่ผ่านการจัดการจากแอ็คเซสพอยต์ในการทดลองที่ 1.3

No.	Time	Source	Destination	Protocol	Info
10	7.779057	192.168.1.10	192.168.1.2	UDP	Source port: 1695 Destination port: 80
11	7.779179	192.168.1.10	192.168.1.2	UDP	Source port: 1695 Destination port: 80
12	7.779302	192.168.1.10	192.168.1.2	UDP	Source port: 1696 Destination port: 23
13	7.779422	192.168.1.10	192.168.1.2	UDP	Source port: 1696 Destination port: 23
14	7.779546	192.168.1.10	192.168.1.2	UDP	Source port: 1697 Destination port: nameserver
15	7.779668	192.168.1.10	192.168.1.2	UDP	Source port: 1699 Destination port: 71
16	7.779789	192.168.1.10	192.168.1.2	DNS	
17	7.829582	192.168.1.10	192.168.1.2	UDP	Source port: 1695 Destination port: 80
18	7.829679	192.168.1.10	192.168.1.2	UDP	Source port: 1695 Destination port: 80
19	7.829798	192.168.1.10	192.168.1.2	UDP	Source port: 1695 Destination port: 80
20	7.829923	192.168.1.10	192.168.1.2	UDP	Source port: 1695 Destination port: 80
21	7.830044	192.168.1.10	192.168.1.2	UDP	Source port: 1696 Destination port: 23
22	7.830165	192.168.1.10	192.168.1.2	UDP	Source port: 1696 Destination port: 23
23	7.830288	192.168.1.10	192.168.1.2	UDP	Source port: 1696 Destination port: 23
24	7.830410	192.168.1.10	192.168.1.2	UDP	Source port: 1696 Destination port: 23
25	7.830531	192.168.1.10	192.168.1.2	UDP	Source port: 1696 Destination port: 23
26	7.830654	192.168.1.10	192.168.1.2	UDP	Source port: 1697 Destination port: nameserver
27	7.830776	192.168.1.10	192.168.1.2	UDP	Source port: 1697 Destination port: nameserver
28	7.830912	192.168.1.10	192.168.1.2	UDP	Source port: 1697 Destination port: nameserver
29	7.831023	192.168.1.10	192.168.1.2	UDP	Source port: 1697 Destination port: nameserver
30	7.831145	192.168.1.10	192.168.1.2	UDP	Source port: 1697 Destination port: nameserver
31	7.831266	192.168.1.10	192.168.1.2	UDP	Source port: 1699 Destination port: 71

32 7.831387	192.168.1.10	192.168.1.2	DNS	
33 7.884693	192.168.1.10	192.168.1.2	UDP	Source port: 1695 Destination port: 80
34 7.884793	192.168.1.10	192.168.1.2	UDP	Source port: 1695 Destination port: 80
35 7.884917	192.168.1.10	192.168.1.2	UDP	Source port: 1695 Destination port: 80



รูปที่ 5.7 กราฟแสดงทรูพุธในแต่ละพอร์ตที่ได้จากโปรแกรม LanTraffic V2 ในการทดลองที่ 1.3



รูปที่ 5.8 กราฟแสดงทรูพุธในแต่ละพอร์ตที่ได้จากโปรแกรม Ethereal ในการทดลองที่ 1.3

ตารางที่ 5.4 ตารางแสดงค่าทรูพุธของแต่ละพอร์ตในการทดลองที่ 1.3

หมายเลขพอร์ต(UDP)	ค่าน้ำหนัก (%)	ทรูพุธ (Throughput) (Mbps)
80	25	1.032
23	25	1.033
42	25	1.033
53 และ 71	25	0.601

5.3.4 ผลการทดลองที่ 2

ตารางที่ 5.5 ตารางแสดงค่าทรูพุธของฝั่งรับที่ได้จากแต่ละการทดลองในการทดลองที่ 2

TEST NO.	Receive throughput divide by por(Mb/s)									
	UDP 554		TCP 20		TCP 80		UDP 90		UDP 100	
	send	recv	send	recv	send	recv	send	recv	send	recv
1	4	3.138	4	4.199	1	1.141	4	4.063	3	3.089
		3.541		2.674		1.139		4.095		3.082
2	3	3.025	3	3.166	0.75	0.855	3	3.079	2.25	2.319
		3.074		2.672		0.856		3.073		2.316
3	2	2.058	2	2.114	0.5	0.570	2	2.066	1.5	1.547
		2.060		2.093		0.570		2.064		1.544
4	1	1.032	1	1.061	0.25	0.285	1	1.032	0.75	0.755
		1.033		1.059		0.285		1.033		0.775

5.6 สรุปผลการทดลอง

จากผลการทดลองที่ 1 จะเห็นได้ว่าข้อมูลที่ถูกส่งผ่านแอ็คเซสพอย์ได้ถูกจัดการด้วยระบบการจัดการคิวที่สร้างขึ้น เพราะฉะนั้นข้อมูลในแต่ละพอร์ตจะสามารถส่งได้มากหรือน้อยนั้นก็ขึ้นอยู่กับค่าน้ำหนักที่เรากำหนดให้ ดังนั้นค่าทรูพุธที่ได้ในแต่ละพอร์ตก็จะสอดคล้องกับเงื่อนไขของพอร์ตและค่าน้ำหนักที่เราได้กำหนดไว้ นอกจากนี้ยังสังเกตุได้จากเฟรมที่วิ่งเข้ามาในเครื่อง พบว่าข้อมูลจะออกมาอย่างมีลำดับตามที่กำหนดไว้ ส่วนในผลการทดลองที่ 2 จะเห็นว่าอัตราทรูพุธที่รับได้เมื่อแอคเซสพอยท์มีการการใช้ระบบรับประกันคุณภาพนั้นจะให้ค่าอัตราทรูพุธที่สูงกว่าเมื่อไม่มีการใช้ระบบรับประกันคุณภาพ โดยเฉพาะอย่างยิ่งข้อมูลประเภทมัลติมีเดีย (UDP 554) จะเห็นความแตกต่างได้อย่างชัดเจน

นอกจากนี้การทดลองยังพบปัญหา ข้อมูลนั้นค้างอยู่ในคิวของแอคเซสพอยท์เป็นจำนวนหนึ่งซึ่งจะยังไม่ถูกส่งออกไป เพราะฉะนั้นทำให้เกิดดีเลย์ของข้อมูลในกรณีของข้อมูลที่อยู่ในพอร์ทที่ถูกจัดสรรให้น้อย ที่เป็นเหตุนี้เนื่องจากการเรียกคิวให้ส่งอีกรอบนั้นต้องรอการเข้ามาของเฟรมข้อมูลก่อนและการส่งข้อมูลออกจากคิวในแต่ละรอบทำได้อย่างจำกัด เพราะฉะนั้นถ้าหากไม่มีข้อมูลเข้ามาการส่งข้อมูลออกก็จะไม่เกิดขึ้นเช่นกัน

บทที่ 6
บทสรุป

6.1 บทสรุป

โครงงานได้ทำการสร้างไวร์เลสแอคเซสพอยท์ด้วยคอมพิวเตอร์ซึ่งมีอุปกรณ์เครือข่ายที่ติดต่อกับอุปกรณ์ใช้สายจำพวกอีเธอร์เน็ทและอุปกรณ์ไร้สายและใช้ระบบปฏิบัติการลีกนุกซ์ โดยแอคเซสพอยท์ที่สร้างขึ้นมีฟังก์ชันในการทำงานเกี่ยวในการจัดคิวการทำงาน โดยจุดที่ใช้ในการแบ่งคิวคือพอร์ทปลายทางของผู้รับ(เครือข่ายใช้สาย) นอกจากนี้ยังได้ทำการสร้างเว็บเพจเพื่อให้ผู้ใช้สามารถคอนฟิกค่าต่างๆ ของอุปกรณ์เครือข่ายไร้สายได้สะดวกเนื่องมีกราฟฟิกที่ผู้ใช้สามารถใช้งานได้ง่าย

โดยข้อจำกัดของโมดุลเราคือ

1. ไม่สามารถกำหนดไอพีแอดเดรสให้กับขาอินเทอร์เฟสที่ถูกเชื่อมต่อด้วยโมดูล
2. การแจกจ่ายไอพีแอดเดรสหรือการทำ dhcp server ยังต้องอาศัยโปรแกรมอื่นช่วย
3. การจัดการในส่วนของ access list หรือการเข้ารหัสต่างๆ สามารถทำได้โดยคำสั่งที่มาพร้อมกับไดร์เวอร์
4. ระยะเวลาในการรอเพื่อตรวจสอบคิว จะต้องใช้เวลาอย่างน้อย 10 มิลลิวินาที
5. ยังพบข้อมูลสูยหายหรือ lost ในการสื่อสารที่ผ่านโมดูลของเราอยู่บ้าง

6.2 ปัญหาและอุปสรรคในการพัฒนา

ความรู้ในเรื่องเกี่ยวกับการพัฒนาเป็นความรู้ที่ไม่เคยศึกษามาก่อนและทำให้ต้องเสียเวลาในการเรียนรู้ค่อนข้างมาก และในระหว่างการพัฒนา เนื่องจากการทำงานอยู่ในระดับเคอร์เนล เรื่องการจัดการรีซอร์สเป็นสิ่งสำคัญ เพราะฉะนั้นในระหว่างการพัฒนาจึงเกิดปัญหาเครื่องแฮงก์อย่างสม่ำเสมอ ทำให้ต้องรีสตาร์ทเครื่องอาจส่งผลกระทบต่ออุปกรณ์ต่างๆ ได้ง่าย ในการทดสอบการทำงานเป็นไปด้วยความยากลำบากอีกทั้งการวัดประสิทธิภาพสามารถทำได้ยาก ทราบเพียงแต่ลำดับการทำงานว่าทำถูกต้องหรือไม่

6.3 แนวทางในการพัฒนาต่อ

เนื่องจากโมดูลที่เราสร้างขึ้นลำดับในการส่งข้อมูลอาจจะเป็นไปตามที่ได้กำหนดไว้ แต่ในแง่ของประสิทธิภาพพบว่าค่าดีเลย์ของข้อมูลยังคงสูงอยู่ เนื่องจากการทำงานยังมีข้อจำกัดบางประการ ซึ่งในการพัฒนาต่อ อาจจะมีการทำที่ตัวไดร์เวอร์ของอุปกรณ์โดยตรง โดยดักการทำงานบางกรณีให้ส่งต่อมายังโมดูลของเรา เพื่อประมวลผลในเรื่องของการพัฒนาคุณภาพให้มีความสามารถเพิ่มมากขึ้น

บรรณานุกรม

สันติ ศรีลาศักดิ์, เกศมณี เที่ยงธรรม. **เปิดโลกลีนุกซ์ไดรเวอร์ kernel series**. นนทบุรี : บริษัท ออฟเซ็ท เพรส จำกัด. 2543.

สันติ ศรีลาศักดิ์, วรวุฒิ เที่ยงธรรม. **เจาะประเด็นงานเขียนโปรแกรมบนลีนุกซ์**. นนทบุรี : บริษัท ออฟเซ็ท เพรส จำกัด. 2543.

สุวัฒน์ ปุณณชัยยะ, ต้น ตัณฑ์สุทธิวงศ์, สุพจน์ ปุณณชัยยะ. **เปิดโลกของ TCP/IP และโปรโตคอลอินเตอร์เน็ต**. กรุงเทพมหานคร : บริษัท โปรวิชั่น จำกัด. 2543.

อรพิน ประวัติบริสุทธิ์. **คู่มือเรียน ภาษาซี**. กรุงเทพมหานคร : บริษัท โปรวิชั่น จำกัด. 2547.

Ashfaq A. Kean. **Practical Linux Programming : Device Drivers, Embedded System, and the Internet**. United States of America : CHARLES RIVER MEDIA, INC. 2002.

Claudia Salzberd Rodriguez, Gordon Fischer, **Steven Smolski. The Linux Kernel Primer**. United States : Peason Education, Inc. 2006.

Herbert Schidt. **C: The Complete Reference**. United States of America : McGraw-Hill. 1995.

Klaus Wehrle, Frank Pählke, Hartmut Ritter, Daniel Müller, Marc Bech. The Linux® Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel. NJ : Pearson Prentice Hall. 2005.

Mark Ciampa. **CWNA Guide to Wireless LANs**. Canada : Thomson Course Technology Inc. 2006.

Michal K. Glass, Yann Le Scouarnec, Elizabeth Naramore, Gary Mailer, Jeremy Stolz, Jason Gerner. **Beginnig PHP, Apache, MySQL Web Development**. Canada : Wiley Publish Inc. 2004.

ภาคผนวก

ภาคผนวก ก
การติดตั้งโปรแกรม(โครงงาน)

เปิด เทอร์มินัลขึ้นมาแล้วไปยังไดเรกทอรี่ของซีดีรอม (โดยมาก จะอยู่ใน "/cdrom")

```
debian:/# cd /cdrom
```

จากนั้นทำการันคำสั่ง "install" เพื่อทำการสำเนาข้อมูลต่างๆ ไปไว้ในไดเรกทอรี่ที่ถูกต้องโดยในที่นี้ หากเป็นข้อมูลที่เกี่ยวโมดุลจะไปไว้ใน "/root/toomv1_1" แต่หากเป็นไฟล์ที่เกี่ยวข้องกับเวปเพจจะไปไว้ใน "/var/www/realtest"

```
debian:/# ./install
```

จากนั้นเข้าไปในไดเรกทอรี่ "/root/toomv1_1"

```
debian:/# cd /root/toomv1_1
```

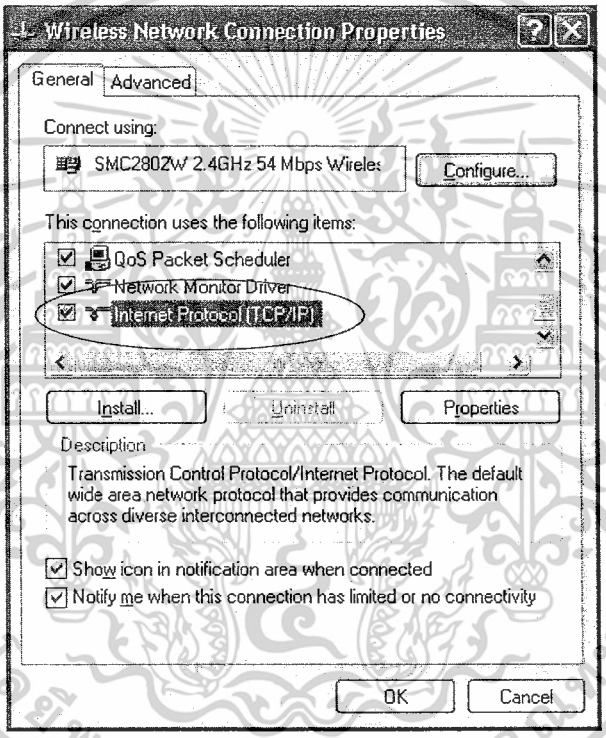
จากนั้นทำการันไฟล์ "netconfig" เพื่อทำการกำหนดค่าที่ต้องใช้ให้กับอินเทอร์เฟสทั้งไวร์เลสและอีเธอร์เน็ท

```
debian:/root/toomv1_1# ./netconfig
```

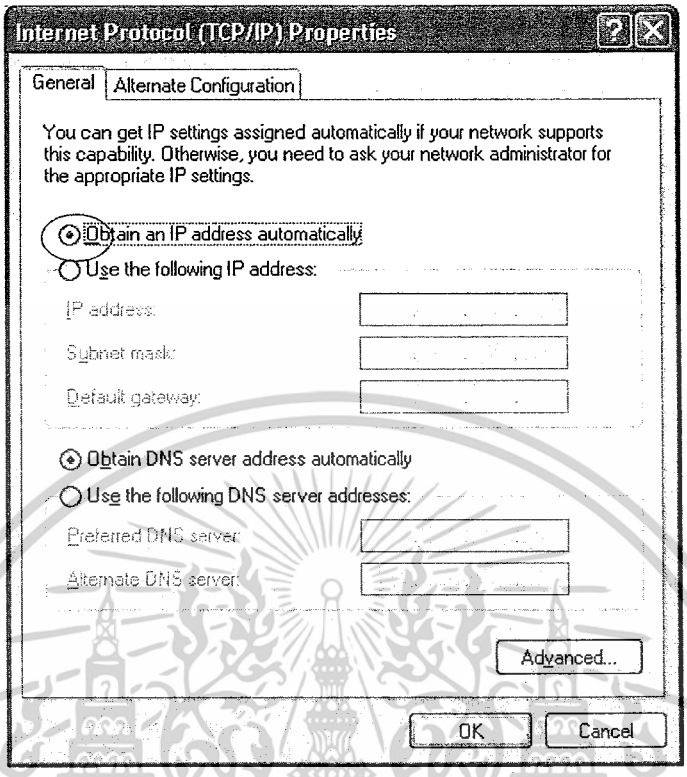
จากนั้นทำการคอมไพล์และเพิ่มโมดูลที่ทำการสร้างขึ้นลงในเคอร์แนล ซึ่งหลังจากขั้นตอนนี้แล้ว เราจะได้เครื่องที่มีการทำงานเป็นแอคเซสพอยท์ได้

```
debian:/root/toomv1_1# ./setupModule
```

จากนั้นที่เครื่องลูกข่าย ให้ลองทำการเชื่อมต่ออินเทอร์เน็ตหรือระบบเครือข่ายภายนอก โดยในที่นี้ระบบเครือข่ายภายนอกมีการเชื่อมโยงกับ dhcp server อยู่แล้ว เพราะเพราะฉะนั้นให้ทำการร้องขอไอพีแอดเดรส โดยในลีนุกซีสามารถทำได้โดยใช้คำสั่ง “dhclient interface_name” หรือในกรณีที่เป็นวินโดวส์สามารถกำหนดได้ในหัวข้อของ คุณสมบัติ->internet protocol จากนั้นให้เลือกในช่อง “obtain an IP address automatically” เพื่อให้ระบบทำการร้องขอไอพีแอดเดรสเอง หลังจากนั้นทำการเชื่อมต่อกับเครือข่ายโดยการเข้าอินเทอร์เน็ต โดยขั้นตอนในการทำการกำหนดค่าในวินโดวส์ที่ใช้การแจกจ่ายไอพีจากเซอร์เวอร์ เป็นไปตาม รูปที่ ก.1-ก.3



รูปที่ ก.1 แสดงเมื่อเข้าหน้า properties ของอินเทอร์เฟสไร้สาย



รูปที่ ก.2 แสดงหน้า internet protocol โดยต้องเลือกในช่อง “obtain an IP address automatically”

รูปที่ ก.3 แสดงเมื่อกำหนดค่าต่างๆ ได้ถูกต้องก็สามารถเข้าใช้เครือข่ายปกติได้

ภาคผนวก ข
โค้ดโมดูลที่ทำการพัฒนา

apQ.h

```
#define DEV0 "eth0"
#define DEV1 "ath0"
#define NUMQ 4
#define MAXLEN 30
struct APQueue{
        int                     port[NUMQ];
        int                     percent[NUMQ];
        int                     maxlenght;
        struct sk_buff_head     headq[NUMQ];
        spinlock_t              acc_que;
};
```

wQ.h

```
#define PORT0 2
#define PORT1 7
#define PORT2 2
#define PORT3 2
#define PORTNUM0 32
#define PORTNUM1 88
#define PORTNUM2 100
```

apQ.c

```
#define __NO_VERSION__
#include <linux/kernel.h>
#include <linux/module.h>
```

```
#include <linux/netdevice.h>
#include <linux/skbuff.h>
#include <net/datalink.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/udp.h>
#include <asm/param.h>
#include <asm/errno.h>
#include <linux/if_ether.h>
#include <linux/spinlock.h>
#include "apQ.h"
#include "wQ.h"
#include "kthread.h"

kthread_t thread;
struct APQueue apQ;
int go_send, s_thread, sum_percent;
struct net_device *eth_if,*ath_if;
struct _scheduler_data *scheduler_data;
static void do_thread(kthread_t *kthread);
struct packet_type eth_packet,ath_packet; //packet type for use in dev_add_pack()
int seq[4]; //sequence of thrasmit port, what port is should go first
int aaaa = 0;  // use for count round of thread

/* find number of port of each segments(use both UDP and TCP) */
int portNum(__u16 *port){
        int port_num;
        /* this is first part of destination port.
         * size if 1 byte (first bit is 1 and 8th bit is 128)
```

```
        * */
        port_num = (int)(*((__u8 *)port+1));

        /* this is seccond part of destination port.
        * size is 1 byte (first bit is 255 and 8th bit is 32768)
        * */
        port_num = (((int)(*((__u8 *)port)))*256)+port_num;

        /* sum of each part, result is destination port 16 bits(u16)*/
        return port_num;
}

/*this function for schedule what queue that sk_buff should be queue
 * this is just get in queue not about send data
 *
 * */
static void classifier(struct sk_buff *sktmp){
        int port_ip = 0;
        int port_tran = 0;

        /* locate network layer header by sum of ethernet header and address at data of socket buffer
*/
        sktmp->nh.iph = (struct iphdr *)((sktmp->data)+sizeof(struct ethhdr));
        port_ip = (int *) sktmp->nh.iph->protocol;

        /* if protocol in network header is 6, is TCP, 17 is UDP */
        printk("protocol is %d\n",port_ip);

        if(port_ip == 6 ){
```

```
                /* this is tcp datagram
                 * set by address at data of socket buffer plus with ethernet header and ip header
                 * will get start address of tcp header*/
                sktmp->h.th = (struct tcphdr *)((sktmp->data)+sizeof(struct ethhdr)+sizeof(struct
iphdr));
        //      port_tran = (int *)((__u8 *)sktmp->h.th->dest);
                port_tran = portNum((__u16 *)&(sktmp->h.th->dest));
                printk("this is tcp, port = %d \n",port_tran);
        }else if(port_ip==17){

                /* this is udp datagram
                 * set by address at data of socket buffer plus with ethernet header and ip header
                 * will get start address of udp header*/
                sktmp->h.uh = (struct udphdr *)((sktmp->data)+sizeof(struct ethhdr)+sizeof(struct
iphdr));
        //      port_tran = (int *)((sktmp->h.uh->dest)+1);
                port_tran = portNum((__u16 *)&(sktmp->h.uh->dest));
        //      port_tran = (*(int *)(sktmp->h.uh + 2));
                printk("this is udp, port = %d\n",port_tran);
        }else{
                /* if in this loop that mean this
                 * data is neither udp and tcp */
                printk("neither tcp and udp\n");
        }

        /*bring sk_buff in to queue classify by port if not match
         * will go in default queue in headq[3]
         * */
        spin_lock(&apQ.acc_que);
```

```
        if(port_tran == apQ.port[0]){
                skb_queue_tail(&(apQ.headq[0]),sktmp);
        }else if(port_tran == apQ.port[1]){
                skb_queue_tail(&(apQ.headq[1]),sktmp);
        }else if(port_tran == apQ.port[2]){
                skb_queue_tail(&(apQ.headq[2]),sktmp);
        }else{
                /* not match any three port above */
                skb_queue_tail(&(apQ.headq[3]),sktmp);
        }
        spin_unlock(&apQ.acc_que);
}

static void transmit_to_eth(){
        int seq_port = 0,i;
        struct sk_buff *tmp;
        if(go_send == 1){

                /* show lenght of each queue */
                printk("q0 lenght: %d w: %d\n",apQ.headq[0].qlen,apQ.percent[0]);
                printk("q1 lenght: %d w: %d\n",apQ.headq[1].qlen,apQ.percent[1]);
                printk("q2 lenght: %d w: %d\n",apQ.headq[2].qlen,apQ.percent[2]);
                printk("q3 lenght: %d w: %d\n",apQ.headq[3].qlen,apQ.percent[3]);
                int check = 0;
                int j=0;
                int send = 0;
                /* can define loop per transmit round by define limit of j */
                for(j=0;j<1;j++){
                        go_send = 0;
```

```
int i =0;
int round = 0;
int check = 0 ;
int w = 0;

/* have 4 queue so i < 4 */
for(i=0;i<4;i++){
        round = 0;
        w = apQ.percent[(seq[i])];
        printk("w: %d\n",w);

        /* this condition check for queue is over max lenght or not? */
        if(apQ.maxlenght<apQ.headq[(seq[i])].qlen){
                w = w+w;
                printk("over max lenght\n");
        }

        /* proccess to transmit to condition for leave this loop
          1). this queue is empty.
          2). number of round is reach.
          */
        while(!(skb_queue_empty(&apQ.headq[(seq[i])]))&&(round<w)){
                spin_lock_irq(&apQ.acc_que);
                tmp = skb_dequeue(&(apQ.headq[(seq[i])]));
                spin_lock_irq(&apQ.acc_que);
                spin_lock_irq(&eth_if->xmit_lock);
                tmp->dev = eth_if;
                send = eth_if->hard_start_xmit(tmp,eth_if);
                if(send!=NET_XMIT_SUCCESS){
```

```
                                        printk(KERN_INFO "sending not success
at %s:%d\n",__FILE__, __LINE__);
                                    }
                                    spin_lock_irq(&eth_if->xmit_lock);
                                    round++;
                                    check++;
                                }
                            }
                            if(check==0)
                                break;
                            check = 0;
                        }
                        go_send = 0;
                }
                return 0;

}

int eth_recv(struct sk_buff *sk,struct net_device *net,struct packet_type *pack){
        sk->data = skb_push(sk, net->hard_header_len);
//      memcpy(sk->mac.ethernet->h_source,sk->mac.ethernet->h_source,ETH_ALEN);

        spin_lock_irq(&ath_if->xmit_lock);
        sk->dev=ath_if;
        int ret = ath_if->hard_start_xmit(sk, ath_if);
        spin_unlock_irq(&ath_if->xmit_lock);

        if(ret!=NET_XMIT_SUCCESS){
                printk("Something error to send to ath");
```

```
        }
//      kfree_skb(sk);
        return 0;
}

int ath_recv(struct sk_buff *sk,struct net_device *net, struct packet_type *pack){
        struct sk_buff *tmp ;
//      tmp = skb_copy(sk,GFP_ATOMIC);

        sk->data = skb_push(sk,net->hard_header_len);

        classifier(sk);

//      kfree_skb(sk);

        return 0;
}

/* find sequence to send packet */
void find_seq(){
        int a,b,c,d;
        a = apQ.percent[0];
        b = apQ.percent[1];
        c = apQ.percent[2];
        d = apQ.percent[3];
        int round;
        int number = 0;
        int val[5] = {0,2,5,7,10};
        for(round=4;round>=0;round--){
```

```
                if(a==val[round]){
                        seq[number]=0;
                        number++;
                }
                if(b==val[round]){
                        seq[number]=1;
                        number++;
                }
                if(c==val[round]){
                        seq[number]=2;
                        number++;
                }
                if(d==val[round]){
                        seq[number]=3;
                        number++;
                }
        }
}

static void do_thread(kthread_t *kthread){
        init_kthread(kthread,"do_thread");
        for(;;){
                interruptible_sleep_on_timeout(&kthread->queue,1);
                mb();
                aaaa++;
                if(kthread->terminate){
                        printk("Terminate %d",aaaa);
                        break;
                }
```

```
                printk("thread woke up %d \n",aaaa);


                /* check for each queue that have some data in own queue or not,
                 * if it have go_send will be 1, if not it won't be assign */
                if((!skb_queue_empty(&apQ.headq[0]))||(!skb_queue_empty(&apQ.headq[1]))){
                        go_send = 1;
                }else
if((!skb_queue_empty(&apQ.headq[2]))||(!skb_queue_empty(&apQ.headq[3]))){
                        go_send = 1;
                }


                /* if some data in queue(any queue) it will be go to transmit_to_eth()
                 * for transmit data in queue to ethernet interface */
                if(go_send==1){
                        spin_lock(&apQ.acc_que);
                        transmit_to_eth();
                        spin_unlock(&ap.acc_que);
                }

        }
        exit_kthread(kthread);
}


/*insert module
 * this module is called when use 'insmod' command.
 * mainly, it's use for initiate important value for
 * this module.*/
static int __init apQ_init_module(void){
        int ret=0;
```

```
        printk(KERN_INFO __FUNCTION__": load bridge module
(%s:%i)\n",__FILE__,__LINE__);

        /* initiate network device by interface name 'ath0' and 'eth0'*/
        eth_if = dev_get_by_name(DEV0);
        ath_if = dev_get_by_name(DEV1);


        /* check that network interface have already to in system
         * if it don't have some interface or both.
         * Insert module will not complete by return -1 */
        if(eth_if){
        }else{
                printk(KERN_INFO ": check your ethernet device");
                /* return value is -19, means no such device */
                ret = -ENODEV;
        }

        if(ath_if){
        }else{
                printk(KERN_INFO ": check your wireless device");
                /* return value is -19, means no such device */
                ret = -ENODEV;
        }

        /* initiate spinlock variable to kernel.
         * or may be use spinlock_t = SPIN_LOCK_UNLOCKED
         * it's same. */
```

```
spin_lock_init(apQ.acc_que);



/* initiate packet type for use in dev_add_pack() */
eth_packet.type = __constant_htons(ETH_P_ALL); // type of ehternet frame

/* pointer to function that will be call when have any data recieve */
eth_packet.func = eth_recv;
eth_packet.dev = dev_get_by_name(DEV0);

/* use dev add pack for thrown any data that interface
 * can caught to the func that define */
dev_add_pack(&eth_packet);

/* initiate packet type for use in dev_add_pack() */
ath_packet.func=ath_recv;
ath_packet.type = __constant_htons(ETH_P_ALL);
ath_packet.dev = dev_get_by_name(DEV1);
dev_add_pack(&ath_packet);

/* initiate socket buffer queue in kernel */
int i;
for(i=0;i<NUMQ;i++){
        printk("init queue at apQ.headq[%d]\n",i);
        skb_queue_head_init(&(apQ.headq[i]));
}

apQ.port[0] = PORTNUM0;
apQ.port[1] = PORTNUM1;
```

```
        apQ.port[2] = PORTNUM2;



        /* assign weight of each port that define in wQ.h */
        apQ.percent[0] = PORT0;
        apQ.percent[1] = PORT1;
        apQ.percent[2] = PORT2;
        apQ.percent[3] = PORT3;



        printk("percent : %d:%d:%d:%d
\n",apQ.percent[0],apQ.percent[1],apQ.percent[2],apQ.percent[3]);
        apQ.maxlenght = MAXLEN;



        /* start kernel thread do_thread is call*/
        start_kthread(do_thread,&thread);

        find_seq();
        printk("seq %d:%d:%d:%d \n",seq[0],seq[1],seq[2],seq[3]);

        return ret;
}
module_init(apQ_init_module);



/*remove module
 * this function is called when user want to remove module*/
```

```
static int __exit apQ_cleanup_module(void){
        kfree(scheduler_data);

        u8 i;
        struct sk_buff *tmp;

        /* flush queue from socket buffer head */
        for(i=0;i<4;i++){
                while(!skb_queue_empty(&(apQ.headq[i]))){
                        tmp = skb_dequeue(&(apQ.headq[i]));
                        tmp->dev = eth_if;
                        eth_if->hard_start_xmit(tmp,eth_if);
                }
        }

        printk(KERN_INFO __FUNCTION__ ": remove bridge module (%s:%i)\n", __FILE__,
__LINE__);

        /* remove stack out */
        dev_remove_pack(&eth_packet);

        /* remove stack out */
        dev_remove_pack(&ath_packet);

        /* stop kernel thread
         * to break for(;;) in do_thread*/
        stop_kthread(&thread);

        return 0;
```

```
}
module_exit(apQ_cleanup_module);

MODULE_AUTHOR("BaByHoT");
MODULE_LICENSE("GPL");
```

netconfig ใช้เพื่อกำหนดค่าให้กับอินเทอร์เฟสทั้งสองอินเทอร์เฟส

```
#!/bin/sh
rmmod toom
wlanconfig ath0 destroy
wlanconfig ath0 create wlandev wifi0 wlanmode master
iwconfig ath0 essid "A"
iwconfig ath0 channel 6
ifconfig ath0 0.0.0.0 promisc down up
ifconfig eth0 0.0.0.0 promisc down up
insmod toom.o
```

setupModule ใช้ในการแทรกโมดูลเข้าในระบบโดยต้องทำการเอาโมดูลเก่าออกก่อน

```
#!/bin/sh
rmmod toom
make clean
make
insmod toom.o
```

install ใช้ในการทำสำเนาข้อมูลลงฮาร์ดดิสก์ของผู้ใช้

```
#!/bin/sh
mkdir /root/apQ
cp ./* /root/apQ
```