การพัฒนา ไคลเอนต์/เซอร์เวอร์ บนยูนิกซ์

Client/Server Development Under Unix Environment Implementation

โดย

นาย วรการ เชิดชูวุฒิกุล 38013287

นาย อัตชัย ชินราช 38013304

ปริญญานิพนธ์นี้ เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญา วิศวกรรมศาสตร์บัณฑิต

สาขาวิชา วิศวกรรมคอมพิวเตอร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหาร ลาดกระบัง

ประจำปีการศึกษา 2540

การพัฒนา ไคลเอนต์/เซอร์เวอร์ บนยูนิกซ์

Client/Server Development Under Unix Environment Implementation

โดย

นาย วรการ เชิดชูวุฒิกุล 38013287

นาย อัตชัย ชินราช 38013304

อาจารย์ที่ปรึกษา

อาจารย์ บรรจง ปิยธำรง

ปริญญานิพนธ์นี้ เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญา วิศวกรรมศาสตร์บัณฑิต

สาขาวิชา วิศวกรรมคอมพิวเตอร์

ปริญญานิพนธ์ปีการศึกษา 2540

ภาควิชา วิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหาร ลาดกระบัง

เรื่อง Client/Server Development Under Unix Environment Implementation

ผู้จัดทำ

1. นาย วรการ เชิดชูวุฒิกุล 38013287
2. นาย อัตชัย ชินราช 38013304

.....อาจารย์ที่ปรึกษา

(.....)

การนำเสนอการพัฒนา ไคลเอนต์/เซอร์เวอร์ บนยูนิกซ์

นาย วรการ เชิดชูวุฒิกุล
นาย อัตชัย ชินราช

อาจารย์ บรรจง ปิยธำรง อาจารย์ที่ปรึกษา
ปีการศึกษา 2540

บทคัดย่อ

วิทยานิพนธ์ฉบับนี้ ได้เสนอเกี่ยวกับการพัฒนาระบบไคลเอนต์/เซอร์เวอร์ (Client/Server) บนยูนิกซ์ (Unix) ซึ่งมีลักษณะและรูปแบบในการสร้าง Client/Server อยู่หลายรูปแบบด้วยกัน เช่น ซอคเก็ตซิสเต็มคอล (Socket System Call) , ทรานสปอร์ตเลเยอร์อินเทอร์เฟส (Transport Layer Interface หรือ TLI) , รีโมทโพซีเจอร์คอล (Remote Procedure Call หรือ RPC) แต่ละชนิดที่มีรูปแบบและคุณสมบัติแตกต่างกันไป เนื้อหาการนำเสนอ เป็นการทดสอบการติดต่อแบบ Client/Server จะทดสอบในลักษณะการติดต่อที่แตกต่างกันในสภาพแวดล้อม UNIX ที่มี HP-UX เป็นระบบปฏิบัติ และ คอมพิวเตอร์ส่วนบุคคล (Personal Computer หรือ PC) ภายในสภาวะแวดล้อมที่เป็นWindows การทดสอบการเชื่อมต่อของ Client/Server ได้ทดสอบบน TCP/IP ทั้งลักษณะการติดต่อแบบของ TCP และ UDP ในการเชื่อมต่อระหว่าง Unix และ คอมพิวเตอร์ส่วนบุคคล (Personal Computer หรือ PC) โดยบนสภาพแวดล้อม PC มีการใช้ฟังก์ชัน (Function) ของวินซอค (WinSock) ที่ทำงานบนสภาพแวดล้อม Windows ในลักษณะของแอปปริเคชั่นโปรแกรมมิ่งอินเทอร์เฟส (Application Programming Interface หรือ API)

หวังว่าจะเป็นประโยชน์สำหรับผู้ที่ต้องการศึกษา หรือพัฒนาเกี่ยวกับ Client/Server บน Unix รวมถึงการทำงานของ WinSock และสามารถนำไปประยุกต์ใช้งานด้านระบบเครือข่ายต่างๆ ได้

Client/Server Development Under Unix Environment Implementation

Voragarn Chirdchuvuttikun

Autachai Chinarash

Bunjong Piyathumrong Advisor

1997

Abstract

This thesis is a Client/Server development under Unix environment implementation which has many type of connection such as Socket System Call, Transport Layer Interface (TLI), Remote Procedure Call (RPC). Each type has different property. For is to test different type of Client/Server. UNIX environment on HP-UX operating system and Personal Computer (PC) on Windows environment has tested operation of TCP/IP both of TCP and UDP protocal. Connection on establishment between UNIX and PC use winsock function Which work on Windows environment with feature of Application Programming Interface (API) specification.

Hope will has useful for want study people or development about Client/Server on Unix and WinSock's working with can apply in varied network systems.

กิตติกรรมประกาศ

การทำปริญญานิพนธ์ในครั้งนี้ ผู้จัดทำขอขอบคุณ

1. อาจารย์ บรรจง ปิยธำรง อาจารย์ที่ปรึกษา
2. คุณ เชิงชาย แซ่เตียว ที่ปรึกษาด้าน Programming
3. คุณ ศราวุธ กวีวงศ์ ที่ปรึกษาด้าน Network

สารบัญ

สารบัญตาราง

ตารางที่ **หน้าที่**

สารบัญภาพ

รูปที่ **หน้าที่**

บทที่1
บทนำ

1.1 ความสำคัญและที่มา

ในโลกปัจจุบันคอมพิวเตอร์ได้เข้ามามีบทบาทเกี่ยวข้องกับชีวิตประจำวันเรามากขึ้นทุกวันเราจึงควรมีความรู้เกี่ยวกับคอมพิวเตอร์ไว้บ้าง เพื่อความสะดวกและสามารถใช้ชีวิตอยู่ในโลกปัจจุบันได้ดียิ่งขึ้น คอมพิวเตอร์ในปัจจุบันมีคอมพิวเตอร์หลายรูปแบบมากมาย และ Client/Server ก็เป็นอีกรูปแบบหนึ่งที่เราจะนำเสนอในวิทยานิพนธ์เล่มนี้ เพื่อหวังว่าผู้ที่ได้อ่านจะนำความรู้ไปพัฒนา และใช้ในชีวิตประจำวันได้บ้าง

Client/Server เป็นแบบการจัดการคอมพิวเตอร์ ที่ได้รับความนิยมอีกรูปแบบหนึ่งในปัจจุบัน เป็นรูปแบบการจัดการคอมพิวเตอร์แบบกระจายการทำงาน โดยเราจะแบ่งหน้าที่ (Process) ให้กับคอมพิวเตอร์ช่วยกันทำงานโดย Server จะเป็นผู้ให้บริการการร้องขอจาก Client ส่วน Client เป็นผู้ใช้บริการเมื่อต้องการความช่วยเหลือจาก Server ในการออกแบบระบบการทำงานแบบ Client/Server นี้จะต้องมีการแบ่งการทำงานอย่างน้อยออกเป็นสองส่วน โดยส่วนของโปรแกรมหลักจะทำงานอยู่บน Client และโปรแกรมที่ให้บริการจะอยู่บน Server ซึ่งอาจจะมีมากกว่าหนึ่ง Server ก็ได้หรือในหนึ่ง Server อาจมีมากกว่าหนึ่ง Process ก็ได้ซึ่งขึ้นอยู่การออกแบบและการทำงานของระบบ ถ้าในระบบมี Client หรือ Server ทำงานมากกว่าหนึ่งเครื่อง เราจะต้องมีการจัดระเบียบในการทำงานให้แก่ระบบ (Concurrency Control) ปริญญานิพนธ์เล่มนี้จะเป็นการพัฒนา Client/Server โมโมเดล (Model) บนสภาพแวดล้อม (Environment) บนเฮลเล็ทท์แพ็คการ์ดยูนิกซ์ (Hewlett Packard Unix หรือ HP-UX) และการติดต่อระหว่าง Client กับ Server จะอยู่ในลักษณะของ รีโมทโพซีเจอร์คอล (Remote Procedure Call หรือ RPC), ทรานสปอร์ตเลเยอร์อินเทอร์เฟส (Transport Layer Interface หรือ TLI), ซอคเก็ต (Socket) ที่อยู่บนสภาพแวดล้อม UNIX และ วินซอค (WinSock) ที่อยู่บนสภาพแวดล้อมที่เป็นวินโดว์ (Windows) บนเครื่องเพอร์ซันแน็ลคอมพิวเตอร์ (Personal Computer หรือ PC) และใช้รูปแบบการติดต่อที่เป็น ทรานสมิชชั่นคอนโทรลโปรโตคอล/อินเทอร์เน็ตโปรโตคอล (Transmission Control Protocol/Internet Protocol หรือ TCP/IP)

1.2 วัตถุประสงค์

ปัจจุบันการติดต่อสื่อสารกันระหว่าง Computer เป็นเรื่องที่จำเป็นและสำคัญไปแล้ว การติดต่อสื่อสารในรูปแบบของ Client/Server ก็เป็นรูปแบบหนึ่งที่ใช้กันอย่างแพร่หลาย และมีการนำไปใช้และประยุกต์ในลักษณะและรูปแบบที่แตกต่างกันไปตามความต้องการในการนำไปใช้งาน เช่น ใช้ในการจัดการฐานข้อมูลในลักษณะของ Client/Server (Database Server) ตัวอย่างเช่น Sql Server, Oracle Server นอกจากการจัดการฐานข้อมูลแล้ว ยังมีการนำไปประยุกต์เกี่ยวกับการจัดการในระบบเครือข่าย

(Network Management) เพื่อใช้ในการบริหารหรือตรวจสอบระบบเครือข่าย ให้มีประสิทธิภาพ เหล่านี้ล้วนอาศัยทฤษฎี และหลักการพื้นฐานของการทำงานในรูปแบบของ Client/Server ดังนั้นจึงมีความจำเป็นและถึงเวลาแล้ว ที่ควรจะศึกษาและทำความเข้าใจถึงพื้นฐานการติดต่อสื่อสารในรูปแบบของ Client/Server เพื่อก่อให้เกิดความเข้าใจถึง Application ที่อยู่ในรูปแบบของ Client/Server ต่างๆที่มีอยู่มากมายในปัจจุบัน และสามารถนำไปประยุกต์ในการสร้าง Application ต่างๆได้ ตัวอย่างเช่น File Transfer Protocol (FTP), Ping, Telnet, Network Management เป็นต้น ดังนั้นวิทยานิพนธ์ฉบับนี้จึงเป็นพื้นฐานที่ดีและเหมาะสม สำหรับการทำความเข้าใจ และสามารถนำไปประยุกต์ใช้งานในรูปแบบของ Client/Server ได้ตามความต้องการ

1.3 ขอบเขต

รายละเอียดของเนื้อหาได้นำเสนอเกี่ยวกับ Client/Server ภายใต้สภาพแวดล้อม Unix รวมไปถึงการทำงานของ WinSock ภายใต้สภาพแวดล้อม Windows โดยส่วนใหญ่ได้บรรยายเพื่อให้ได้เข้าใจถึงรูปแบบและ คุณสมบัติต่างๆ ของ Client/Server ซึ่งเป็นรูปแบบมาตรฐานที่ใช้กันอยู่ทั่วไป เพื่อก่อให้เกิดความเข้าใจ และเกิดแนวคิดในการสร้างหรือพัฒนาไปใช้ ในการพัฒนาระบบเครือข่ายได้ตามความต้องการ

1.4 วิธีการดำเนินงาน

ในการดำเนินงานมีการรวบรวมข้อมูล และรายละเอียดที่เกี่ยวกับการทำงาน และโครงสร้างของ Client/Server ในรูปแบบต่างๆ รวมไปถึงศึกษาการทำงานของ Function ต่างๆที่ใช้ในการเขียนโปรแกรมเพื่อใช้ในการทดสอบการทำงานของ Client/Server ในรูปแบบของ Socket System Call, Windows Socket และ RPC โดยทำการรับส่งข้อมูลกันระหว่าง Client กับ Server โดยมีการจัดรูปแบบของการทดสอบ ดังนี้

- Socket System Call Server บน Unix (TCP) ---> Socket System Call Client บน Unix (TCP)
- Socket System Call Server บน Unix (UDP) ---> Socket System Call Client บน Unix (UDP)
- Socket System Call Server บน Unix (TCP) ---> Windows Socket Client บน PC (TCP)
- Socket System Call Server บน Unix (UDP) ---> Windows Socket Client บน PC (UDP)
- Remote Procedure Call บน Unix

บทที่สอง อธิบายถึงทฤษฎีและหลักการในการสื่อสารระหว่าง Client กับ Server ว่าสามารถติดต่อกันได้อย่างไร และมีวิธีไหนบ้างได้ในบ้างส่วน

บทที่สาม อธิบายถึงการออกแบบโปรแกรมในลักษณะ Client/Server ในรูปแบบต่างๆ โดยแยกลักษณะการออกแบบเป็น การออกแบบโปรแกรมบน Client, การออกแบบโปรแกรมบน Server และ การออกแบบโปรแกรมโดยใช้ RPC

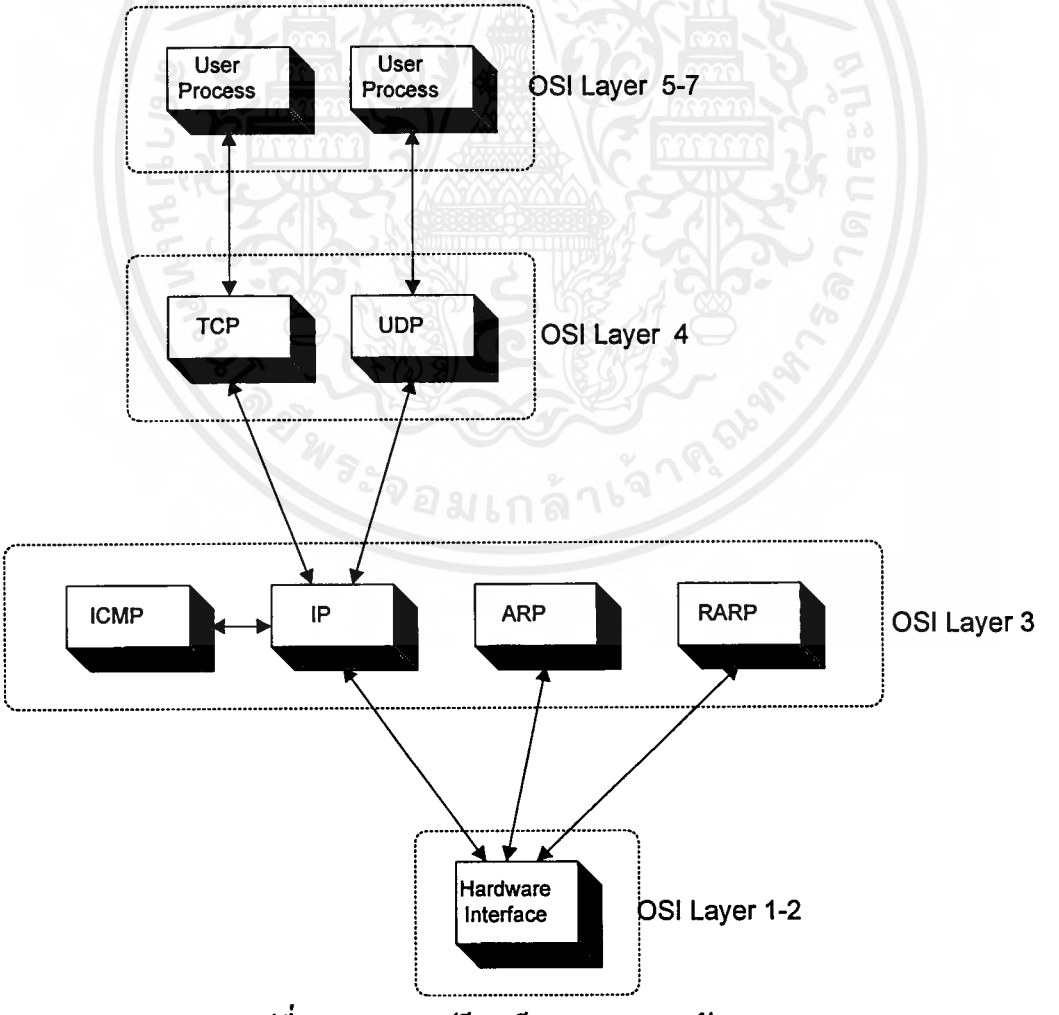
บทที่สี่ เป็นเนื้อหาของโปรแกรมที่ใช้ในการทดสอบการทำงานของ Client/Server ในรูปแบบของ Socket System Call, Windows Socket ทั้ง TCP และ UDP และการทำงานของ RPC โดยมีการแสดงลักษณะการเขียน Code ของ Programs, แสดงการทำงานของโปรแกรมด้วย Flowchart และผลลัพธ์ของแต่ละรูปแบบไว้พอสมควร

บทที่ห้า บทสรุปและวิจารณ์

บทที่ 2
ทฤษฎีและหลักการการสื่อสารระหว่าง Client และ Server

2.1 การสื่อสารระดับ Protocol

ในระบบไวด์แอเรียเน็ตเวอร์ค (Wide Area Network หรือ WAN) จะประกอบด้วย Network หลายวงเข้าด้วยกันซึ่งเรารวมเรียกว่าอินเทอร์เน็ต (Internet) ในปี คศ. 1960 Advanced Research Project Agency (ARPA) Of The U.S. Department of Defense (DoD) ได้รวบรวม มหาวิทยาลัย, องค์กรทางด้าน Computer และสถาบันวิจัยทางทหาร รวมเรียกว่า อาปาเน็ต (ARPANET) ได้รวมกันคิดค้น TCP/IP Protocol เพื่อใช้ในระบบติดต่อการสื่อสารระหว่างคอมพิวเตอร์ที่เป็นประเภทเดียวกันหรือต่างชนิดกันได้ โดยที่เครื่องคอมพิวเตอร์จะอยู่ห่างไกลกันในลักษณะของ WAN หรืออยู่ภายในบริเวณเดียวกันในลักษณะของโลคอลแอเรียเน็ตเวิร์ค (Local Area Network หรือ LAN) ก็สามารถติดต่อสื่อสารกันได้ ต่อมาได้นิยมมาใช้ใน อินเทอร์เน็ตเน็ตเวิร์คคิ่ง (Internet Networking) และในการสื่อสารที่ต้องการความเชื่อถือในการส่งข้อมูล



รูปที่2.1 แสดงการเปรียบเทียบ OSI Model กับ TCP/IP

ใน TCP/IP Protocol สามารถรับและส่งข้อมูลได้ 2 ลักษณะคือ

1. คอนเน็คชั่น-ออเรียลเต็ด (Connection-Oriented) หรือ TCP

2. คอนเน็คชั่นเลส (Connectionless) หรือ ยูสเซอร์ดาต้าแกรม โปรโตคอล (User Datagram Protocol หรือ UDP)

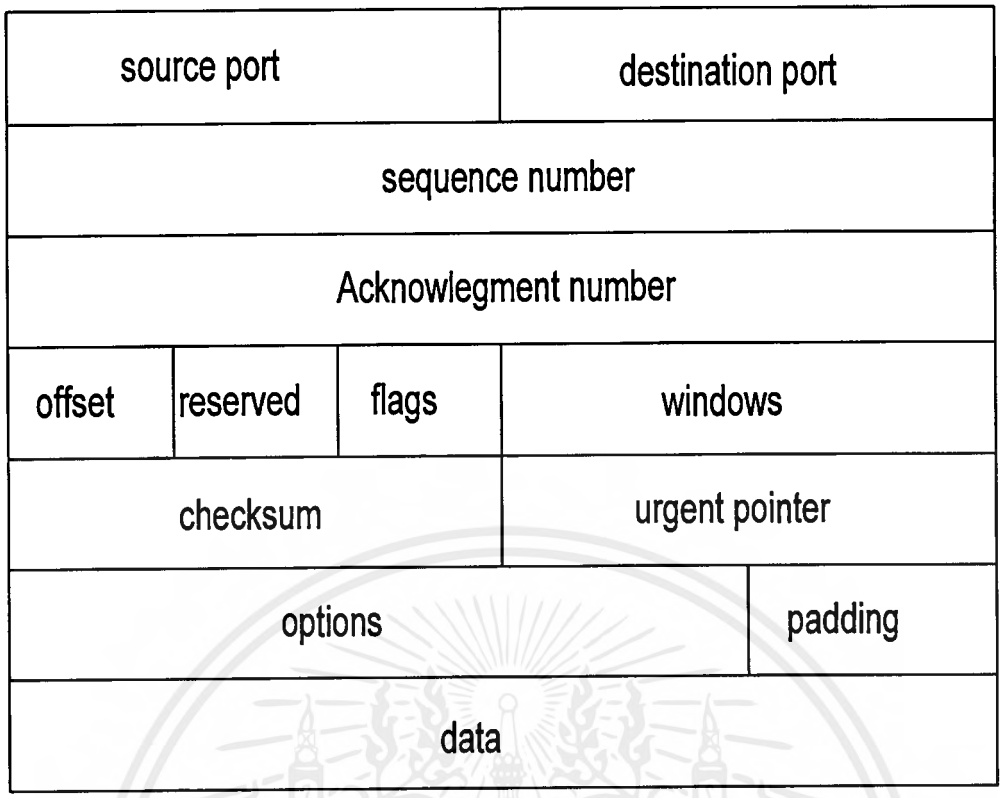
โดยที่ Protocol ทั้งสองต้องอ้างอิงผ่าน IP ซึ่งอยู่ใน Layer3 ของโอเพ็นซิสเต็มอินเทอร์คอนเน็ค (Open Systems Interconnect หรือ OSI) Model

2.1.1 Connection-Oriented

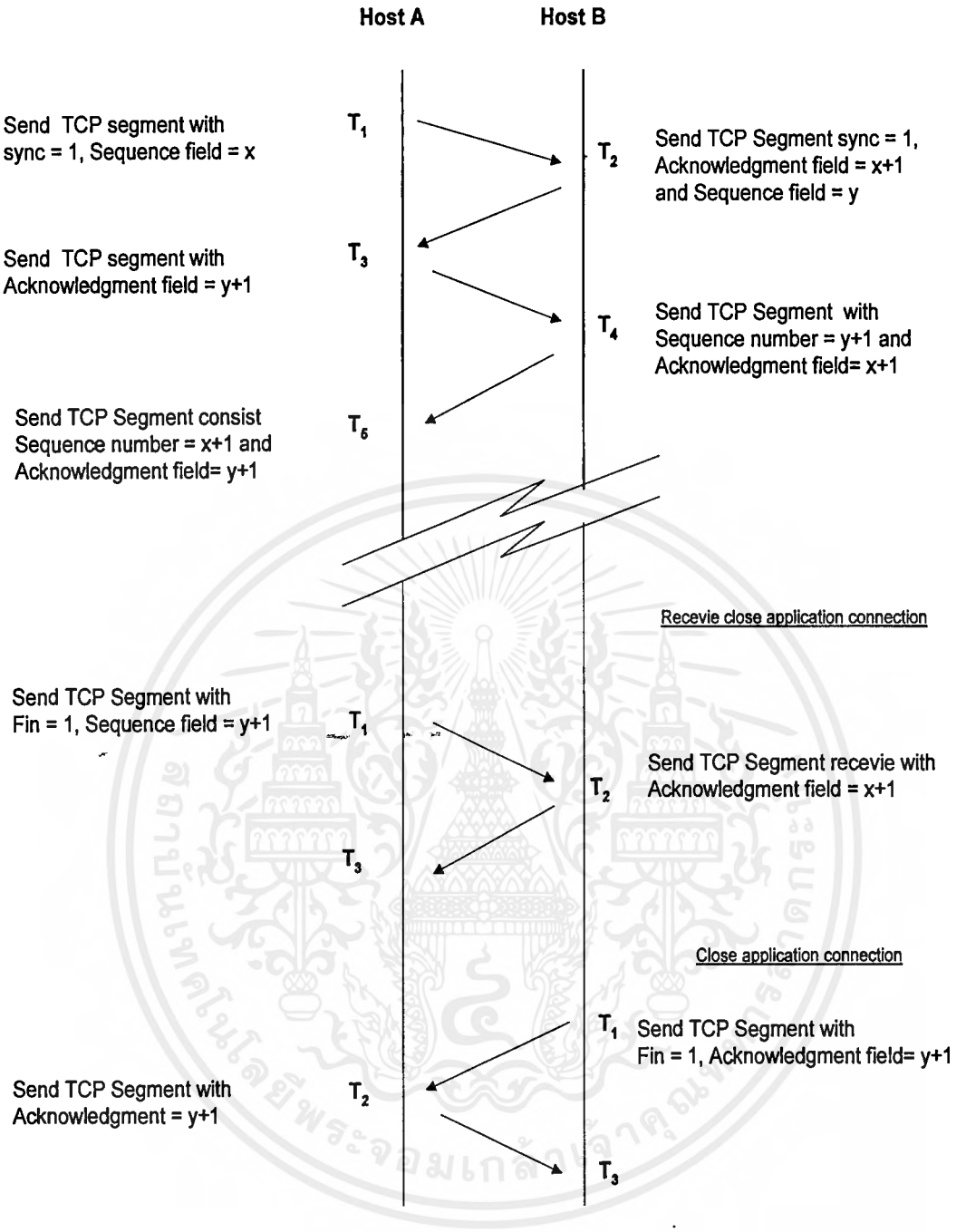
TCP เป็น Protocol จะถูกออกแบบมาให้มีความเชื่อถือในการรับส่งข้อมูล (Reliable) มีการตรวจสอบการรับส่งข้อมูล (Connection-Oriented) โดยใช้การตรวจสอบความผิดพลาด (Error Checking), การตอบรับ (Acknowledging) และ การควบคุมการทำงาน (Flow Control) เป็นตัวตรวจสอบความถูกของการรับส่งข้อมูลในส่วนของดาต้าลิงส์เลเยอร์ (Datalink Layer) สามารถส่งข้อมูลในระบบที่สามารถส่งและรับพร้อมกันได้ (Full Duplex) และส่งข้อมูลแบบไบต์สตรีม (Byte Stream)

จากรูปที่2.3 แสดงการติดต่อระหว่างคอมพิวเตอร์สองเครื่องโดยผ่าน TCP Protocol ก่อนที่จะมีการส่งข้อมูลระหว่างคอมพิวเตอร์สองเครื่องจะมีการส่งสัญญาณ Connection เมื่อสร้างเวอร์ชวลเซอร์กิต (Virtual Circuit) ระหว่างกันก่อนการรับส่งข้อมูลจริง

หลังจาก TCP ทำการเริ่มต้นทำการเชื่อมต่อ (Initial Connection) ก็จะได้ TCP ขึ้นมาสองเส้นทางเพื่อทำการแลกเปลี่ยนข้อมูลแบบ Full-Duplex ระหว่างสองบัฟเฟอร์ (Buffer) ของทั้งคู่ เพื่อใช้ในการรับและส่งข้อมูล ถ้าการส่งข้อมูลมีการผิดพลาดหรือไม่มีสัญญาณตอบรับก็จะมีการนำข้อมูลที่อยู่ใน Buffer ออกมาส่งซ้ำอีกครั้ง เพื่อเป็นการประกันว่าข้อมูลที่ Application ส่งออกไปจะถึงปลายทาง จะเห็นได้ว่า TCP จะถูกใช้กับการส่งข้อมูลที่มีความเชื่อถือได้ใน Application ที่มีการใช้ TCP/IP ในการรับส่งข้อมูล เช่น เทลเน็ต (TELNET), ซิมเบิลเมลทรานสเฟอร์โปรโตคอล (Simple Mail Transfer Protocol หรือ SMTP) เป็นต้น



รูปที่2.2 แสดงส่วนต่างๆ ของ TCP Header

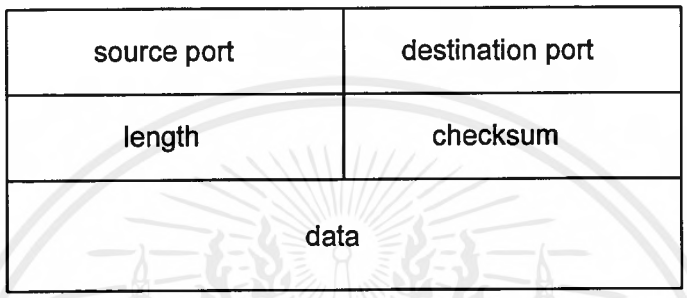


รูปที่2.3 แสดงการส่งสัญญาณตอบรับระหว่าง Host A และ Host B ของ TCP Protocol

2.1.2 Connectionless

UDP ที่กำหนดขึ้นมาโดย ARPANET ซึ่งเป็นส่วนหนึ่งของ TCP/IP Protocol ในลักษณะแบบ UDP/IP ซึ่งรูปแบบการส่งข้อมูลจะไม่มีการหาเส้นทางการติดต่อระหว่างโฮส (Host) และไม่มีการทำ Connection ก่อนการส่งข้อมูล เพื่อเป็นการลดโอเวอร์เฮด (Overhead) ของเพคเก็ต (Packet) ที่ส่งออกไปเพราะฉะนั้น UDP จึงเป็น Protocol ที่มีเฮดเดอร์ (Header) ขนาดเล็กซึ่งประกอบด้วย IP Header, UDP Header และ Data รวมเรียกว่า UDP Datagram ในส่วนของ IP Header จะแบ่งออกเป็นสองส่วนคือ ที่อยู่ต้นทาง (Source Address) และที่อยู่ปลายทาง (Destination Address) UDP Header จะประกอบด้วย

ความยาวของUDP (UDP Segment Length) และส่วนของการตรวจความถูกต้องของข้อมูล (Checksum) เพราะในการส่งข้อมูลของ UDP จะมีการส่งแฮนแชก (Handshake) ตรวจสอบการทำ Connection เมื่อ UDP Segment ถูกส่งไปยังปลายทางที่ IP ที่อยู่บน UDP Segment แล้วก็จะมีการตรวจความถูกต้องของข้อมูลด้วย Checksum ถ้ามีความผิดพลาดข้อมูลก็จะส่งสัญญาณ Acknowledging Error ตามแบบ Flow Control ต่อไป โดยการส่งข้อมูลแบบ UDP นี้จะไม่มี Buffer Protocol ในการรับส่งข้อมูล UDP จึงนิยมนำมาใช้เพราะ UDP สามารถส่งข้อมูลได้เร็วกว่า TCP เพราะมี Header ขนาดเล็กจึงนำไปใช้ในระบบการส่งข้อมูลที่ไม่ไกลนัก เช่น ในระบบ LAN หรือการกระจายข้อความ (Boardcast)



รูปที่2.4 แสดงส่วนต่างๆ ของ UDP Header

2.2 Socket System Calls

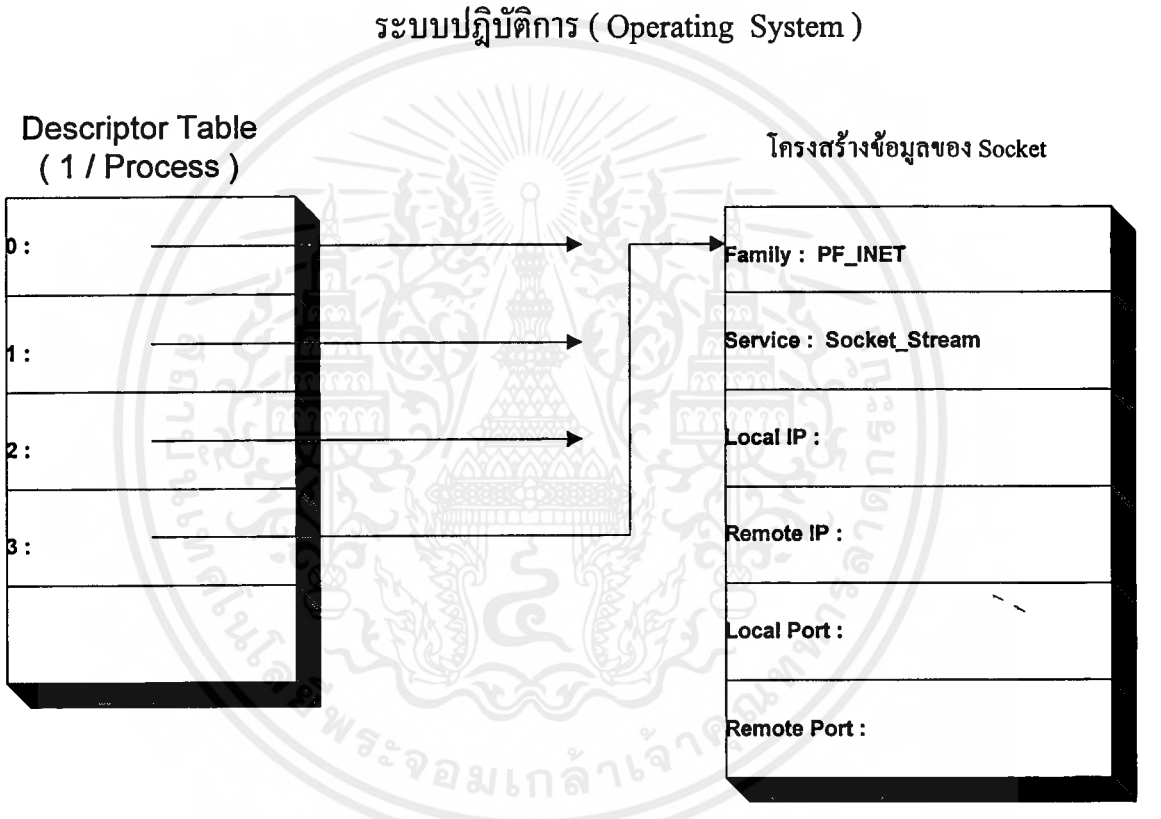
ในปี 1980 ARPA Group มหาวิทยาลัย Berkeley ได้วิจัยโปรแกรม Transport TCP/IP ที่ใช้บนUnixขึ้น โดยจะออกแบบเพื่อใช้ในการติดต่อสื่อสารกับที่ต่างๆ บน Network ซึ่งมักจะถูกทำให้เป็น System Call บนระบบปฏิบัติการ (Operating System หรือ OS) ในลักษณะ Socket Interface หรือเรียกกันอีกอย่างหนึ่งว่า Socket Function the Berkeley Socket Interface มีลักษณะเหมือนฟังชั่น (Function) ทั่วไปที่ใช้ในการติดต่อใน Network ซึ่งมีใช้กันอยู่ทั่วๆไป โดย Socket จะอ้างอิงกับ TCP/IP Protocol

ในเวลาต่อมาได้มีบริษัทฯคอมพิวเตอร์ต่างๆ ได้นำเอา Socket Function นี้ไปใช้และพัฒนากันอย่างกว้างขวาง จนกลายเป็นมาตราฐานในเวลาต่อมา เช่น ซันไมโครซิสเต็มอินคอร์ปอร์เรตเต็ด (SUN Microsystem Incorporated), ดิจิตอลอิควิปเม็นต์คอร์ปอร์เรชั่น (Digital Equipment Corporation) และ ARPA group ยังนำ Socket Interface ไปไว้เป็นส่วนหนึ่งของ Berkeley UNIX Operating System อีกด้วย

2.2.1 โครงสร้างของ Socket

ใน UNIX เวลาที่ Application มีการเรียกใช้ Open Functions จะมีการสร้างไฟล์เดสคริปเตอร์ (File Descriptor) ซึ่งใช้ในการเข้าถึงไฟล์ (Access File) เก็บไว้ในตาราง (Descriptor Table) ในลักษณะหนึ่ง ตัวชี้ (Index Descriptor) จะเป็นตัวชี้ตำแหน่งภายในโครงสร้างข้อมูลของซอคเก็ต (Internal

Data Structure Socket) จะมี Socket Descriptor ซึ่งเป็นตัวเลข Address ที่ชี้ไปยังตำแหน่งของโครงสร้างข้อมูลของ Socket ซึ่งมีลักษณะเหมือน File Descriptor ทุกครั้งที่ Application เรียกใช้ Socket จาก OS จะมีการสร้าง Data Structure ขึ้นเพื่อใช้เก็บรายละเอียด (Information) เพื่อใช้ในการสื่อสาร และนำตำแหน่งของ Data Structure ไปเก็บไว้ใน Descriptor Table ในลักษณะพอยเตอร์ (Pointer) ดังตัวอย่างในรูป2.5 ทุกครั้งที่มีการเรียกใช้คำสั่งสร้างซอคเก็ต (Create Socket) จะต้องการ Initial Connection ถ้า Server เรียกใช้ Socket จะเรียกว่าพาสซีฟซอคเก็ต (Passive Socket) ถ้าฝั่ง Client ทำการ Initiate Connection จะเรียกว่าแอคทีฟซอคเก็ต (Active Socket) หลังจาก Initiate ถ้ามีการ Connect ตอบจะทำการเก็บ Information ลงบน Socket Data Structure ก่อนที่จะทำการใช้ Socket จริง



รูปที่2.5 แสดงโครงสร้างข้อมูลของ Socket และ Descriptor Table หลังเรียกใช้ Socket จาก Operating System

2.2.2 ลักษณะและการใช้งาน Socket System Call

ซอคเก็ตซิสเต็มคอล (Socket System Call) สามารถแบ่งออกเป็นสองกลุ่ม โดยกลุ่มแรกจะทำงานภายใต้ Function และยูทิลิตี้รูทีน (Utility Routines) อีกกลุ่มจะทำงานในลักษณะ Clients/Server

การใช้ Socket System Call จะมีความซับซ้อนบ้างเพราะ Sockets มี พารามิเตอร์ (Parameters) ซึ่งสามารถนำไปใช้ใน Program ได้หลายรูปแบบสามารถใช้ติดต่อสื่อสารระหว่าง Client กับ Server ใน

ลักษณะ TCP หรือ UDP ได้ด้วยการกำหนด Remote Endpoint Address แบบเฉพาะเจาะจงสำหรับ Client และกำหนด Remote Endpoint Address แบบไม่เฉพาะเจาะจงสำหรับ Server

เพื่อให้เข้าใจ Socket ได้มากขึ้น อาจจะเริ่มทดลองใช้ Socket ติดต่อกันในลักษณะ Client/Server โดยใช้ TCP

❋ **The Socket Calls** ในการสร้าง Socket เมื่อใช้ในการสื่อสารเครือข่าย (Network Communication) โดยเริ่มเก็บ Information Socket ใหม่ลงบน Descriptors เช่น Application ต้องการใช้ Protocol แบบใด เช่น โปรโตคอลแฟรมมิลี่อินเทอร์เน็ต (Protocol Family Internet หรือ PF-INET) สำหรับ TCP/IP Service แบบ TCP หรือ UDP ถ้าใช้ Socket ในแบบ Internet Protocol Family

❋ **The Connect Call** หลังจากที่เรียกใช้ Create Function แล้ว ทางฝั่ง Client จะทำการเรียก Connect Function เพื่อทำการติดต่อกับ Remote Server ซึ่งใน Connect Function ของ Client จะระบุ IP Address และ Protocol Port ของ Remote Machine ไว้ หลังจากที่ทำการ Connect กับ Remote Machine ได้แล้วจึงทำการส่ง Data ต่อไป

❋ **The Write Call Write Function** จะถูกใช้ในการส่ง Data บน TCP Connection ระหว่าง Client กับ Server โดยทาง Client จะเรียกว่าจะส่งการร้องขอ (Send Request) ส่วนทาง Server เรียกว่าจะส่งการตอบรับ (Send Replies)

Write Function มีสาม Arguments ในการส่งผ่านข้อมูล คือ Descriptor of Socket, Address of Data และ Length of Data โดยปกติแล้ว Write Function จะทำการสำลองข้อมูล จาก Buffer ของเคอร์แนล (Kernel) OS แล้วส่งออกไปยังในระบบ Network ถ้าเกิด System Buffer เต็มแล้ว Write Function จะทำการหยุดการส่ง Data แต่ใน TCP จะทำการสร้าง Buffer เพิ่มเพื่อรองรับ Data ที่จะส่งมา

❋**The Read Call Read Function** จะถูกใช้เป็นตัวรับข้อมูลระหว่าง Client กับ Server บน TCP Connection โดยปกติหลังจากการเกิดการ Connection เรียบร้อยแล้ว ฝั่ง Server จะมี Read Function ขอรับการติดต่อจาก Client โดย Write Function หลังจากที่ Client ได้ส่งสัญญาณการติดต่อออกไปแล้ว ฝั่ง Client จะใช้ Read Function เพื่อรอรับการตอบรับจาก Server

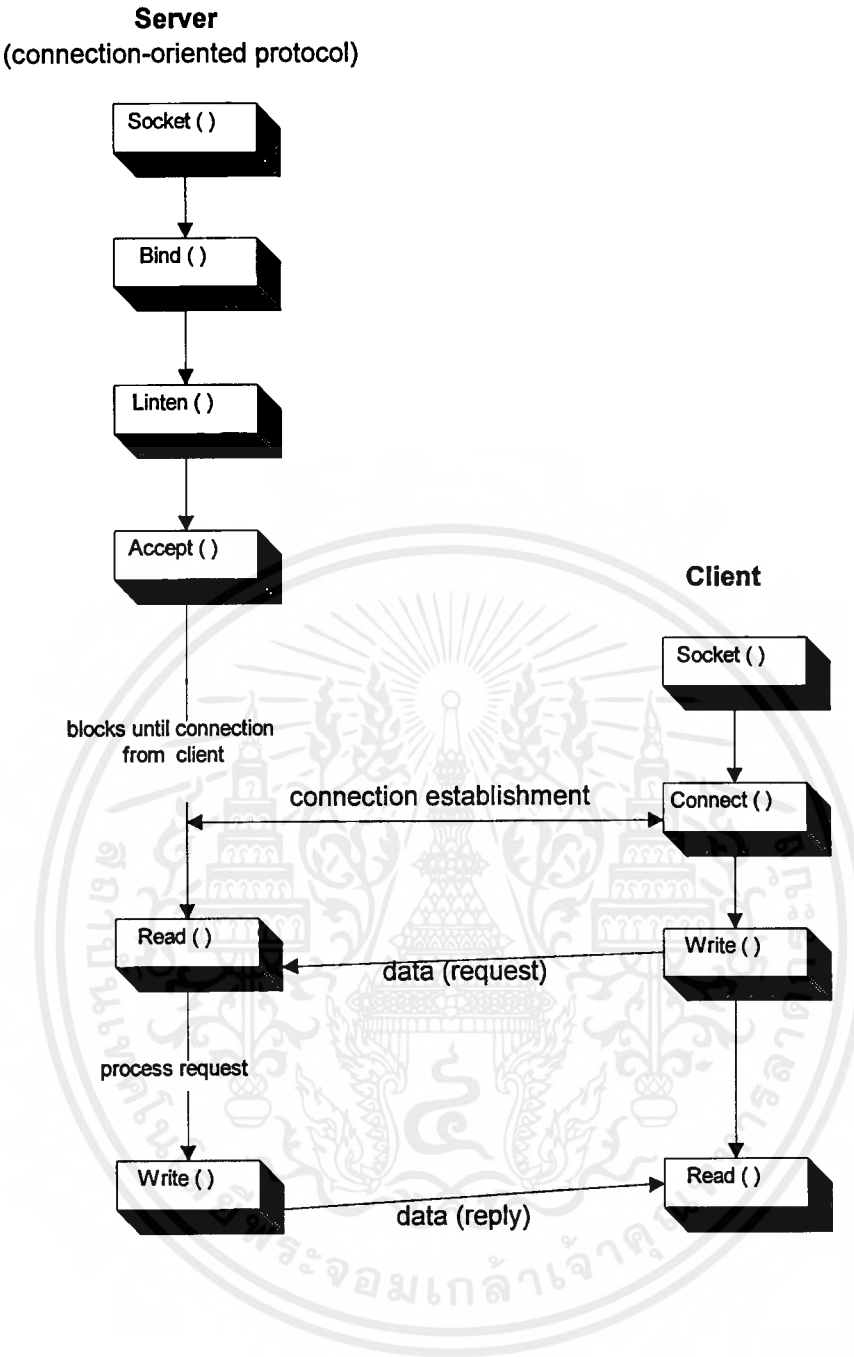
Read Function จะประกอบสาม Argument คือ Descriptor Address ของ Buffer และ ขนาดของ Buffer จะทำการขยายข้อมูลที่ได้รับมาแล้วนำข้อมูลที่ได้จาก Socket ไปไว้ใน Buffer ที่ใช้งาน ถ้าข้อมูลที่ได้รับมาไม่ครบก็จะส่งขนาดที่พบกลับไปใน Client และ Server ค่า Socket ใช้ Read Function ได้รับข้อความ (Message) แบบ UDP แล้วจะไม่มีการนำข้อความมาเก็บไว้ใน Buffers แต่จะส่งไปให้ Application เลยแต่ถ้าเป็นการส่งแบบ Connection- Oriented จะต้องมี สาม Argument คือ Socket Descriptor, ที่อยู่ของข้อมูลใน Buffer และขนาดของ Buffer

❋ **The Close Call. Close Function** Client และ Server ที่ใช้เมื่อจบการใช้ Socket เพื่อจะทำการ คืนค่าที่จองไว้ (Deallocate) ถ้า Socket นี้มีเพียง Process เดียวที่ใช้ก็จะทำการ Terminate และ Deallocate เมื่อเรียกใช้ Close Function แต่ถ้าเป็นการใช้ Socket ร่วมกันหลายๆ Process ทุก Process ต้องเรียกใช้ Close Function เมื่อครบแล้วจึงจะทำการ Terminate และ Deallocate Socket นั้น

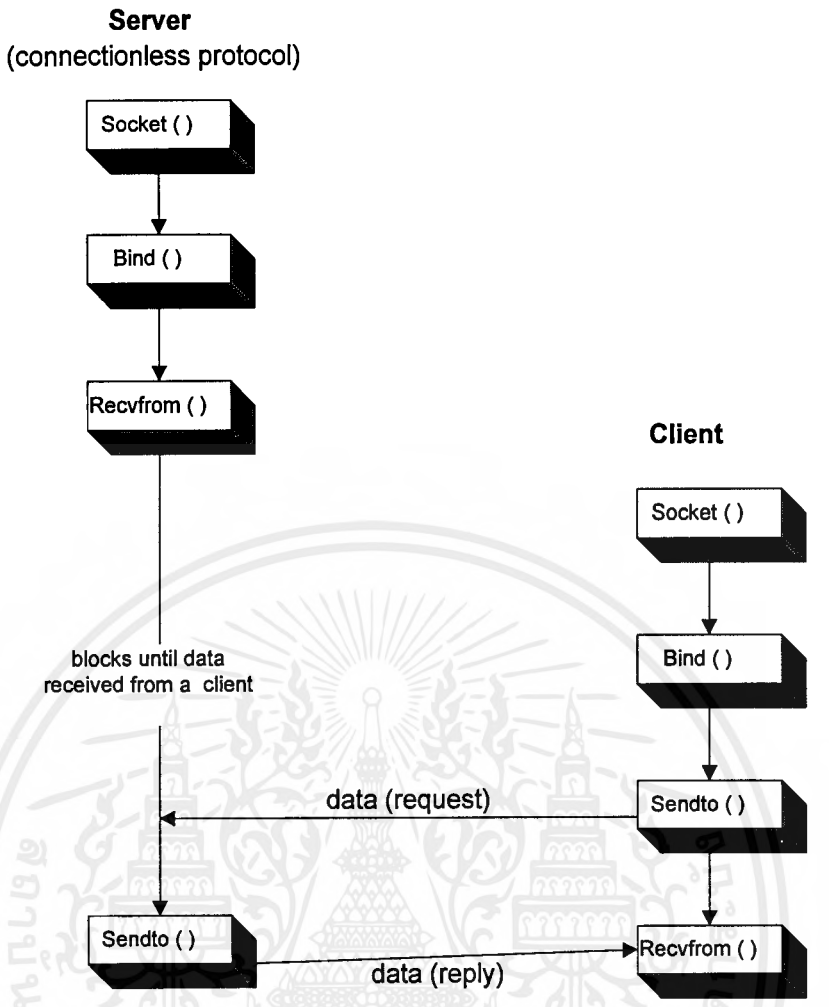
❋ **The Bind Call** ในตอนแรกที่เรียกใช้ Socketนั้น Socket ยังไม่ถูกกำหนด Endpoint Address ของ Local และ Remote Application ต้องใช้ Bind Function บอก Local Endpoint Address ให้กับ Socket โดยผ่าน Socket Descriptor Argument และ Endpoint Address Argument สำหรับ TCP/IP Protocol Endpoint Address จะใช้ซอคเก็ตแอสแดรสอินเทอร์เน็ต (SocketAddr-In) Structure และต้องกำหนด IP และ Port ให้กับ Socket ด้วย ส่วน Socket ที่ Server ใช้ Bind Function เพื่อบอกว่าใช้ Port อะไรเพื่อขอการ Connection

❋ **The Listen Call** เมื่อ Socket พร้อมที่จะทำงาน หลังจากที่เกิดการ Connection ระหว่าง Server และ Client แล้วถ้าเปิด Connection-Oriented ใน Server ใน Passive Mode จะหมายถึงการรอรับการตอบรับการสื่อสารที่เข้ามาของ Client และจะรอรับตลอดเวลาจนกว่าจะมีการติดต่อเข้ามา Server จะทำการตอบรับในเวลาไม่กี่เสี้ยววินาที และถ้าการติดต่อสื่อสารขาดหายไป Server จะต้องใช้ Listen Function จะขนาดของ Queue โดยผ่าน Argument ของ Function ไปยัง OS เพื่อทำการติดต่อสื่อสาร Socket ใหม่ เพื่อเข้าสู่ Passive Mode

❋ **The Accept call** หลังจากที่ Socket ได้ทำการ Bind และเข้าสู่ Passive Mode Server จะใช้ Accept Function สร้าง Connection เพื่อตอบสนองการติดต่อของ Client ที่เข้ามาใหม่ Accept Function จะสร้าง Socket ขึ้นมาใหม่ เพื่อรับการติดต่อจาก Client โดยจะส่ง Descriptor ของ Socket อันใหม่ออกไป และทำการส่งข้อมูลไปยัง Socket ตัวใหม่ หลังจากที่จบการใช้งาน Socket แล้ว Server จะทำการปิด Socket



รูปที่ 2.6 แสดงขั้นตอนการทำงานของ Socket ในแบบ Connection-Oriented Protocol



รูปที่2.7 แสดงขั้นตอนการทำงานของ Socket ในแบบ Connectionless

2.3 Transport Layer Interface (TLI)

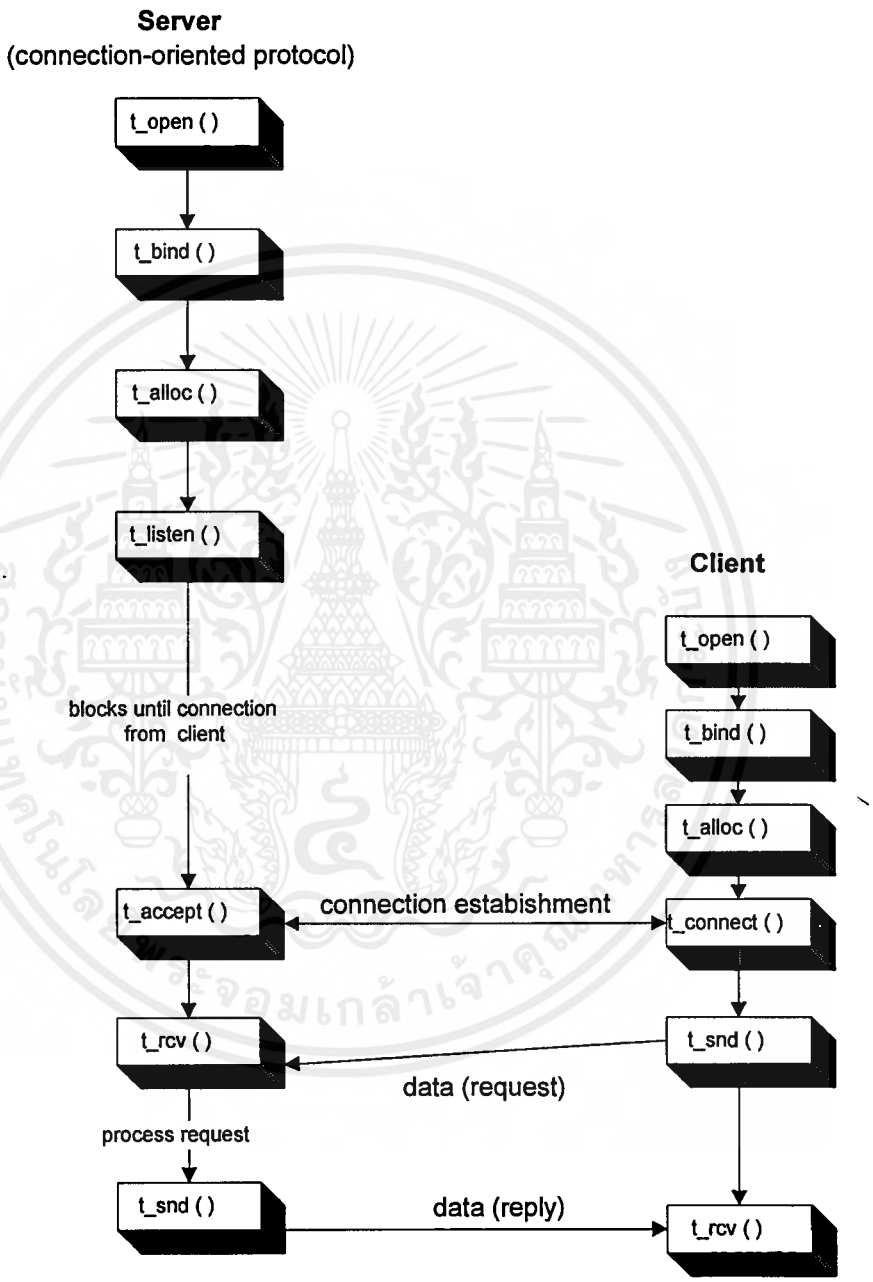
Transport Layer Interface ถูกนำมาเป็นส่วนของ UNIX System เวอร์ชั่น 3.0 ในปี คศ. 1986 TLI เป็น Function ที่ถูกนำมาใช้ในลักษณะไลบรารี่ (Library) เพื่อใช้ในการ ลิงส์ (Link) โดยใช้คำสั่ง cc maim.c -lnsl_s ซึ่งเป็น Function มาตราฐานใน Network Services Library TLI เป็น Function ไม่ได้เป็น System Call เหมือน Socket System Call แต่ TLI Function จะทำงานในลักษณะ Stream Interface ในระบบการสื่อสารข้อมูล เราสามารถใช้ TLI ได้ 2 ส่วน คือ

1. Transport Endpoint เป็นส่วนติดต่อระหว่าง 2 Process ใน TLI Function ซึ่งใน Socket จะเรียกว่า Half-Association

2. Transport Provider เป็นชุดของ Routing ที่ Computer ใช้ในการติดต่อสื่อสารข้อมูลกับ User Process TLI จะใช้เป็น Interface ระหว่าง User Process กับ Transport Provider

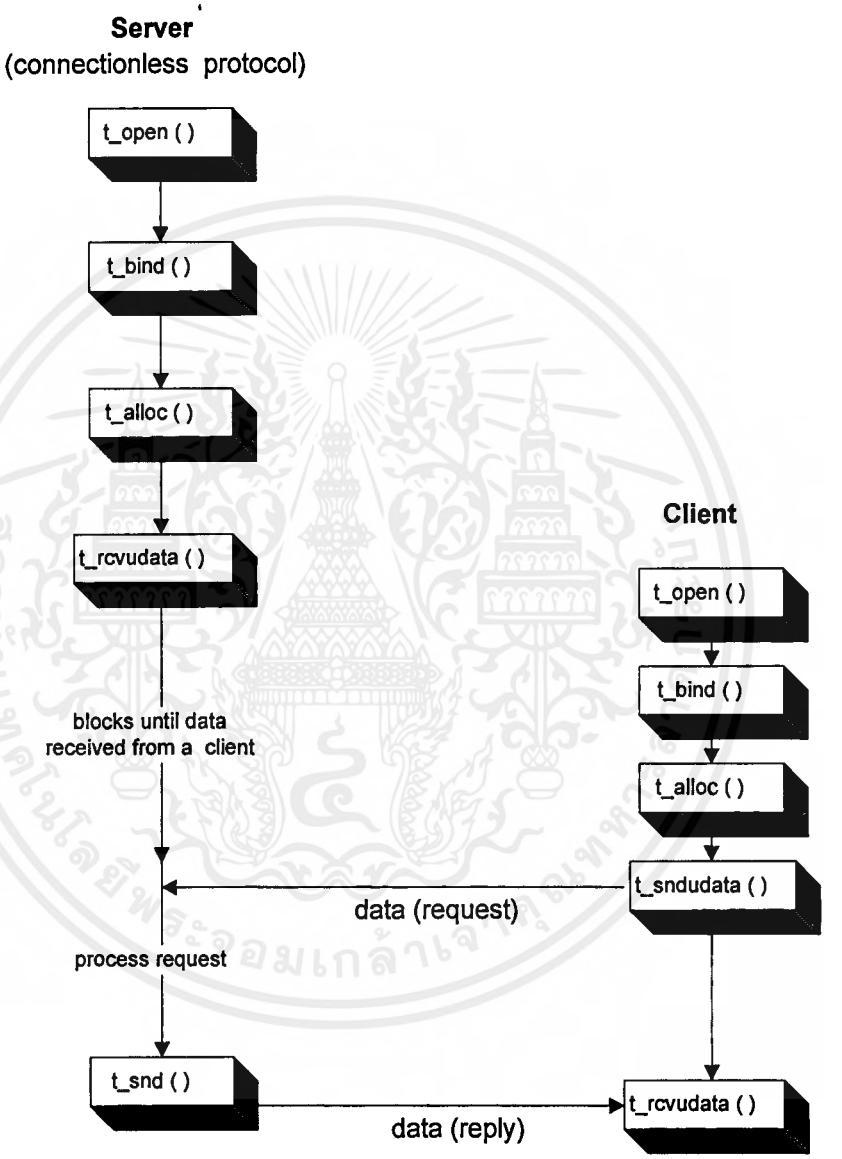
TLI ใน UNIX เวอร์ชั่น 3.X ของ AT&T นั้นจะไม่สามารถใช้ Transport Provider ได้ในส่วนของระบบปกติ แต่ใน UNIX System V Release 4.0 จะสามารถใช้ TCP/IP Transport Provide ได้และมีโปรแกรมทางด้าน Network ที่ใช้ TLI ได้แก่ UUCP และ RFS (Remote File System)

2.3.1 โครงสร้างของ Transport Layer Interface



รูปที่2.8 แสดงการติดต่อระหว่าง Server กับ Client โดยใช้ TLI Function ในลักษณะ Connection-Oriented Protocol

จากรูปที่2.8 ลักษณะลำดับขั้นตอนการติดต่อระหว่าง Server กับ Client จะมีลักษณะคล้ายการติดต่อโดยใช้ Socket ใน TLI ถ้าติดต่อแบบ Connection-Orient จะใช้ Function t_rcv และ t_ snd เป็นตัวรับส่งข้อมูลและมี t_ accept Function กับ t_ connect Function เป็นตัวสร้าง Connection ระหว่าง Server กับ Client ก่อนการส่งข้อมูล



รูปที่2.9 แสดงการติดต่อระหว่าง Server กับ Client โดยใช้ TLI Function ในลักษณะ Connectionless Protocol

2.3.2 Transport Endpoint Address ของ TLI

โครงสร้างของ Transport Endpoint Address ถูกกำหนดเป็นรูปแบบมาตราฐานบน TLI Function ที่จะติดต่อผ่าน TCP/IP Protocol TLI จะใช้โครงสร้างนี้ในการส่งผ่านข้อมูลระหว่าง TLI Function กับ User Code

รูปแบบ

```
/*****************************************************************************/
        * Include (tiuser.h)
          struct netbuf {
                unsigned int maxlen; (ขนาดสูงสุดของ Buffer สำหรับกรณีOverflow)
                unsigned int  Len;   (ขนาดของข้อมูลจริงใน Buffer)
                Char        *buf ;   (ใช้เป็นตัวชี้ตำแหน่งเริ่มต้นของข้อมูล Buffer)
      };
/*****************************************************************************/
```

ถ้าเปรียบ กับ Socket จะเปรียบได้กับโครงสร้างของ Sockaddr Netbuf จะใช้เป็นผ่านค่าใน TLI Function โดยจะบอกขนาดของ Buffer, ขนาดของข้อมูลจริง

2.3.3 การใช้งาน TLI Function

t_error Function ใช้ในการส่ง Error Message เมื่อเกิด Error ขึ้น t_error Function จะมีค่าเป็น -1 ถ้าปกติจะมีค่าเป็น 0

t_open Function จะถูกนำมาเป็นขั้นตอนแรกในการติดต่อ Transport Endpoint และใช้เป็น Function กำหนดค่าต่างๆ ให้กับ Transport Provider เช่น ต้องการเปิด Device File (/dev/tcp,/dev/udp หรือ /dev/Ip) และกำหนดค่าต่างๆ เพื่อใช้ใน TLI Function อื่นต่อไป

t_alloc Function จะถูกเรียกให้เมื่อต้องการสร้าง Buffer ให้กับ Function ต่างๆบน TLI Function โดยได้ค่ากลับมาเป็น Pointer

t_free Function เป็น Function ที่ใช้ในการสร้าง Buffer แต่ง่ายกว่า t_alloc Function แต่มักจะใช้ในการตรวจ Memory ว่าจะใช้เป็น Buffer ได้หรือไม่ โดยจะอ้างอิงกับโครงสร้างของ Netbuf

t_bind Function เป็น Function ที่ใช้กำหนด Address ให้กับ Transport Endpoint

t_conncet Function จะถูกใช้เมื่อ Connection-Oriented Client ทำการสร้าง Connection กับ Server ค่าที่ส่งไปใน Function คือ Address ของ Server , Address ของ Process และข้อมูล User ที่จะส่งของไปพร้อม t_connection Function พร้อมทั้งมี ลำดับที่ได้มาจาก t_listen Function

t_listen Function เป็น Function ที่ Connection-Oriented Server ใช้รอการติดต่อจาก t_conncet Function บน Client เทียบได้กับ Accept System Call.

t_aceept Function จะถูกเรียกโดย t_listen Function เมื่อ t_listen Function ได้รับการติดต่อจาก Client ถ้าใน Concurrent Server t_accept Function จำทำการสร้าง Connection ขึ้นใหม่โดยเรียกจาก

t_open Function เพื่อรอรับการเรียกจาก Client หรือ Process อื่นที่ทำการติดต่อมายัง Server ใน Accept System Call จะกระทำ t_listen, t_open และ t_accept Function พร้อมกันบน Concurrent Server

t_snd Function และ **t_rcv Function** เป็น Function เป็น Function ที่ใช้ในการส่งและรับข้อมูล โดยค่าที่ส่งกลับมาจะเป็นตัวเลขของข้อมูลในการรับ-ส่ง

t_sndudata Function และ **t_rcvudta Function** เป็น Function ที่ใช้ในการรับ-ส่งข่อมูลในการสื่อสารแบบ Connectionless Protocol

t_rcvuderr Function จะถูกเรียกใช้เมื่อเกิด Error ระหว่างการสื่อสาร Connectionless Protocol หรือ ลบสถานะ Error

t_look Function เราจะใช้ Function นี้ในการตรวจสอบ Error ว่าเกิดขึ้นกับ Transport Endpoint มาจาก Transport Provider ใด ค่าที่จะนำมาตรวจสอบได้มาจาก t_errno จะ Set ค่า Look t_look Function จะแปรงค่าเป็น Integer เพื่อไปเทียบหาว่าเกิด Error กับ Transport Provider ใด

t_sndrel Function และ **t_rcvrel Function** สามารถแบ่งการทำงานออกเป็น 2 แบบ คือ

1. Abortive Release เป็นการรับส่งสิ่งที่ไม่รับประกันว่า Data ยังอยู่หรือไม่ (Delivery)
2. Orderly Release เป็นการส่งที่รับประกันข้อมูลอยู่หรือไม่

ถ้าเกิดการสูญหายของข้อมูล t_rcv Function จะมีค่า -1 , t_errno Function จะ Set ค่า TLOOK และ t_ordrel Functionเป็นตัวบอกลำดับ

t_snddis Function และ **t_rcvdis Function** 2 Function นี้ใช้กับ Obortive Release โดยที่ t_snddis จะถูกใช้ใน 2 กรณี คือ

1. ใช้ในการยกเลิกการติดต่อระหว่าง Client กับ Server
2. ใช้ในการยกเลิกการร้องขอการติดต่อ

เมื่อทำการยกเลิกการติดต่อเราจะส่ง NULL ผ่าน t_revdis Function เพื่อยกเลิกการส่งข้อมูลของฝ่ายส่ง

t_closc Function ใช้ในกรณีปิดการส่ง Transport Endpoint และไม่ยกเลิกการ Connection ระหว่าง Host กับ Client จะเป็นยกเลิกการส่งของ Process นี้ แต่ Process อื่นอาจจะยังติดต่อกันอยู่ได้

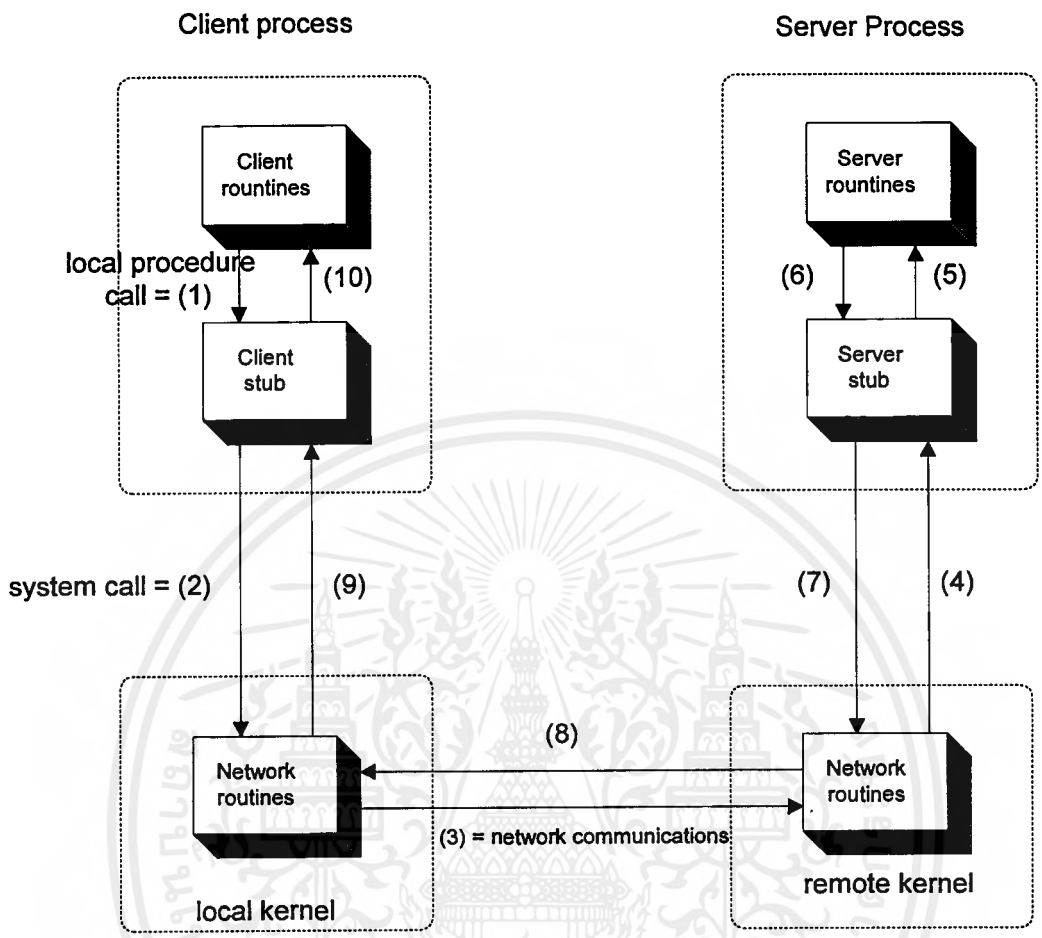
		Sockets	TLI	Messages	FIFOs
Server :	allocate space		t_alloc()		
	Create Endpoint	Socket()	t_Open()	msgGet()	mknod() Open()
	Bind Address	Bind()	t_Bind()		
	specify queue	Listen()			
	wait for Connection	Accept()	t_Listen()		
	Get new fd		t_Open() t_Bind() t_Accept()		
Client :	allocate space		t_alloc()		
	create Endpoint	Socket()	t_Open()	msgGet()	Open()
	Bind Address	Bind()	t_Bind()		
	Connect to Server	Connect()	t_Connect()		
	transfer data	read()	read()	msgrcv()	read()
		write()	write()	msgsnd()	write()
		Recv()	t_rcv()		
		send()	t_snd()		
	daragrams	Recvfrom() sendto()	t_rcvudata() t_sndudata()		
	terminate	close() shutdown()	t_close() t_sndrel() t_snddis()	nsgotl()	close() unlink()

ตารางที่ 2.1 แสดงตารางการเปรียบเทียบระหว่าง Socket Function กับ TLI Function

2.4 Remote Procedure Calls.

ใน Application ทั่วไปจะมีการเรียก Procedure Call เมื่อเทียบกับ Client/Server Model แล้ว Main Program ก็เปรียบได้กับ Client โดยมีการผ่านค่าไปยัง Procedure ที่เป็น Server ใน RPC จะทำงานในลักษณะ Process ที่ทำบน Local System เรียกใช้ Procedure ที่อยู่บน Remote System

2.4.1 ลำดับการทำงานของ RPC



รูปที่2.10 แสดงการทำงานของ Remote Procedure Call (RPC)

จากรูปการทำงานของ Remote Procedure Call เป็นลำดับได้ดังนี้

1. Procedure บน Client จะทำการเรียก Client Stub โดยจะส่งออกไปในรูปมาตราฐานในรูปแบบ Marshaling
2. ข้อมูลที่จะส่งไปยัง Remote System โดย Client Stub ต้องเรียกผ่าน System Call บน Local Kernel
3. ข้อมูลจะถูกส่งผ่านใน 2 ลักษณะคือแบบ Connection-Oriented และแบบ Connectionless
4. ส่วนบน Server Server Stub จะรอการติดต่อจาก Client และจะทำการ Unmashaling และเปลี่ยนแผลงข้อมูลที่ได้จา Client
5. Server Stub จะทำงานเหมือน Local Procedure แต่กระทำบน Server และนำผลที่ได้ผ่านไปยัง Client Stub โดยผ่าน Network ต่อไป
6. เมื่อ Procedure บน Server ทำงานจบก็จะส่งค่าต่างๆไปยัง Server Stub

7. Server Stub จะทำการแปลงค่าต่างๆ ที่ต้องการส่งกลับและ Marshal ส่งกลับไปยัง Client Stub

8. ข้อมูลจะถูกส่งผ่าน Network ไปยัง Client Stub

9. Client Stub จะอ่านข้อมูลที่ได้มาจาก Network โดยผ่าน Local Kernel

10. หลังจากข้อมูลได้ผ่าน Client Stub เรียบร้อยแล้วค่าที่ได้จะถูกส่งไปยัง Client Procedure ที่เรียกใช้ Remote System ต่อไป

สาระของ RPC คือต้องการซ้อน Network Code ทั้งหมดไว้ใน Stub Procedure ใน Application บน Client และ Sever ไม่ต้องมีขั้นตอนที่ยุ่งยากเหมือน Socket หรือแบบอื่นๆ RPC จะช่วยในการเขียน Distributed Application เป็นเรื่องที่ง่ายขึ้น ถ้าเรานำ RPC ไปเปรียบกับ OSI Model แล้วจะอยู่ระหว่าง Transport Layer กับ Application Layer ภายในส่วนของ Spoliation Layer RPC จะเข้าไปจัดการเกี่ยวกับเรื่องการติดต่อสื่อสาร (Networking Detail) โดยจะนำส่วนที่ต้องการส่งออกไปนำไปไว้ใน Argument เพื่อแปลงให้เป็นรูปแบบมาตราฐานเพื่อใช้ในการสื่อสารระหว่าง Client กับ Server โดยที่ Application Layer ไม่ต้องสนใจเรื่อง Byte Order

2.4.2 การติดต่อสื่อสารของ RPC

เมื่อ Procedure บน Client ติดต่อกับ Procedure บน Server หรือการส่งผ่านข้อมูลระหว่าง Client และ Server สิ่งที่เกิดขึ้นไม่ได้มีแค่การเรียกใช้ System Calls, การแปลงข้อมูล และการส่งผ่านข้อมูลแต่ RPC มีหน้าที่มากกว่านั้นคือ

- Parameter Passing
- Binding
- Transport Protocol
- Call Semantics
- Data Representation
- ประสิทธิภาพ (Performance)
- ระบบความปลอดภัย (Security)
- Exception Handling
- Parameter Passing

เป็นการส่ง Parameter ระหว่าง Client กับ Server โดยที่ Client Stub จะนำส่วนข้อมูลจาก Client และ Package ลงออกไปใน Network โดยจะบอกที่อยู่ปลายทางไปกับค่าที่ส่งของไปด้วย

- Binding

Binding เป็นการกระทำของ Client ที่ต้องการรายละเอียดของ Procedure ที่ต้องการติดต่อด้วยบน Remote System ที่ระบุ ประกอบด้วยกัน 2 ส่วนคือ

- m Remote Host ที่เป็น Server ที่ต้องการติดต่อ
- m Process ที่ต้องการติดต่อบน Server

คือการที่ Client ต้องการรายละเอียดข้อมูลของ Server ที่ต้องการติดต่อด้วย มีเทคนิคหนึ่งที่ใช้กันคือการรวมข้อมูลไว้ที่ศูนย์กลาง (Centralized Database) ในปี 1984 Birrell และ Nelson คิดจะนำรายละเอียดของ Server ทั้งหลายนำมาขึ้นทะเบียน Internet โดยมีเจตนาเพื่อรองรับการเรียกใช้งานของ Remote Procedure โดยที่เมื่อไร Server Boot ขึ้นมาจะส่งสัญญาณไปบอกแก่ศูนย์กลาง และเมื่อใดที่ Client ต้องการติดต่อขอรายละเอียด การขอใช้งานก็สามารถมาเอาได้ที่ศูนย์กลางนี้ แต่ก็ยังมีเทคนิคอื่นอีกเช่น ให้จัด Super Server เพื่อใช้เป็นศูนย์กลางเพื่อรอรับการติดต่อการ Client โดย Super Server นี้จะเป็น Server ที่เก็บที่อยู่ของ Server ต่างๆในระบบ

❋ Transport Protocol

RPC ที่คิดค้นกันมานั้นบางชนิดจะสามารถใช้ได้ Transport Layer Protocol เดียวแต่บางชนิดก็สามารถใช้ได้ สอง Protocol เช่น Sun RPC สามารถใช้ได้ทั้ง UDP และ TCP, Xerox Courier สามารถใช้ SSP (Sequenced Packet Protocol) และ Apollo RPC จะใช้ UDP และ Dss

❋ Exception Handling

อีกหน้าที่ของ RPC อีกอย่างหนึ่งคือ เมื่อเกิดความผิดปกติของ Process บน Client หรือ Server ถ้าเป็น UNIX Shell แล้วจะมีการแจ้งข้อความหรือสร้าง Core image File ปัญหาที่เกิดกับ RPC นั้นอาจเกิดจากปัญหาบน Procedure, ปัญหาของ Stub ที่ขาดการติดต่อระหว่าง Client กับ Server หรือเกิดการถูกทำงานของ Process บน Server RPC ก็จะมีรูปแบบการตรวจเช็ค การทำซ้ำของ Stub หรือการยกเลิกการติดต่อแล้วแต่เหตุการณ์

❋ Call Semantics

ในกรณีที่ Local Procedure มีการรับส่งข้อมูลหรือสื่อสารกับ Remote Procedure ในบางครั้งอาจเกิดปัญหาหรือข้อมูลผิดพลาดระหว่างการส่งข้อมูลจึงต้องมีการให้ส่งข้อมูลกลับมาใหม่ในบางกรณี ข้อมูลนั้นให้ส่งอีกครั้งก็จะได้เหมือนเดิมคือไม่มีผลเสียต่อการทำงาน (Idempotent) เช่นการส่งค่าไปคำนวณบน Remote Procedure แต่จะมีอีกกรณีหนึ่งถ้าให้ส่งข้อมูลกลับไปอีกครั้งแล้วข้อมูลจะไม่เหมือนเดิมจะมีผลเสียต่อการทำงาน (Non Idempotent) เช่นการทำงานที่เป็นประเภท Interactive บน RPC จะแบ่งออกได้ 3 ระดับ คือ

1. Exactly Once คือการทำ Remote Procedure สามารถทำงานในครั้งเดียว เช่น การส่งสัญญาณบอกว่า Server Halts
2. At most Once คือการทำงานของ Remote Procedure ทำการส่งค่าซ้ำอีกครั้งหนึ่งแล้วจะได้ค่าไม่เหมือนเดิม
3. At Least Once คือการที่ Remote Procedure ส่งค่ากี่ครั้งก็ยังเหมือนเดิม

❋ ระบบความปลอดภัย (Security)

สำหรับ RPC แล้วสามารถรักษาความปลอดภัยให้กับระบบได้ทุกรูปแบบโดยจะใช้ทำงานบน Remote System ได้โดยสามารถสั่งงานได้จาก Local System ไปยัง Remote System ซึ่งในแต่ละค่ายจะมีเทคนิคที่แตกต่างกันออกไป

❋ Data Representation

การทำงานระหว่าง Client กับ Server บนระบบที่เหมือนกันบางครั้งอาจเกิดปัญหาความเข้ากันไม่ได้ของข้อมูล เมื่อเราใช้ RPC ก็จะเกิดปัญหาความต่างกันของระบบ Client และ Server จึงต้องมีการแปลงข้อมูลโดยที่ RPC จะกำหนดรูปแบบมาตราฐานของข้อมูลที่จะส่งโดยให้ Client ให้ส่งข้อมูลในรูปแบบของ ASCII และ Server ส่งข้อมูลในรูปแบบของ EBCDIC

- ใน TCP/IP Protocol จะส่งข้อมูลทุกชนิดด้วย 16 Bit และ 32 Bit สำหรับ Protocol Header ใน Header Protocol สามาระแบ่งออกเป็น Host Id และ Type Protocol Number

- ใน File Transfer Protocol (FTP) จะใช้ 16 Bit สำหรับ Block Number และ Error Code และ ASCII Code สำหรับข้อมูลที่เป็น Character รวมทั้ง File Name, Modeและ Error Striny

- ในระบบของ Remote Login Client และ Server จะใช้ 16 Bit สำหรับ Window-Size Field

❋ ประสิทธิภาพ (Performance)

ประสิทธิภาพในการทำงานร่วมกันระหว่าง 2 Process นั้น การติดต่อสื่อสารก็มีผลต่อประสิทธิภาพของระบบด้วย แต่โดยปกติแล้วถ้าใช้ RPC ใน Client/Server จะทำให้ประสิทธิภาพลดลงประมาณ 10% และ RPC จะทำให้การเขียน Network Programming ง่ายขึ้น แต่เราก็ไม่สามารถนำ Local Procedure ไปใช้แทนบน Remote Procedure ได้หมดทุกอย่าง

ในปี 1984 Birrell และ Netson ได้ทำการทดสอบประสิทธิภาพของ RPC โดยใช้ LAN ที่เป็น Ethernet โดยในการทดลองจะเขียน Procedure ให้รับส่งข้อมูล 16 Bit word 2 ชุด ผลคือ ระหว่าง Local System และ Remote System มีประสิทธิภาพแตกต่างกัน 100% แต่เมื่อเปลี่ยนเป็นการส่ง 16 Bitword 40 ชุด ผลระหว่าง Local System และ Remote System แตกต่างกัน 33%

Data Type	Sun RPC	Xerox Courier	Apollo NDR
8-bit logical			●
16-bit logical		●	
32-bit logical	●		
8-bit signed integer			●
8-bit unsigned integer			●
16-bit signed integer		●	●
16-bit unsigned integer		●	●
32-bit signed integer	●	●	●
32-bit unsigned integer	●		●
64-bit signed integer	●		
64-bit unsigned integer	●		
Byte order	big-endian	big-endian	big-endian or little-endian
signed integer format	2' s-complement	2' s-complement	2' s-complement
32-bit floating point	●		●
64-bit floating point	●		●
floating-point format		IEEE	IEEE, VAX IBM or Cray
character Type	ASCII	16-Bit NS	ASCII or EBCDIC
enumeration	●	●	●
structure (record)	●	●	●
fixed-length single-dimensional array	●	●	●
variable-length single-dimensional array			●
fixed-length multidimensional array			●
variable-length multidimensional array	●		●
discriminated union	●	●	●
fixed-length opaque data	●	●	
variable-length opaque data			

ตารางที่2.2 แสดงตารางเปรียบเทียบชนิดของข้อมูลที่สามารถใช้ได้กับ RPC

2.5 WinSocks

Windows Sockets เป็นตัวกำหนดการติดต่อกับ Programming บน Windows เพื่อการเข้าถึง Function ในNetwork Windows Socketsหรือมีชื่อเรียกอีกอย่างว่า WinSocks ซึ่งมีรูปแบบ และลักษณะที่เปิดกว้าง ดังนั้นมันจึงอนุญาติให้การพัฒนาเป็นไปได้อย่างอิสระ

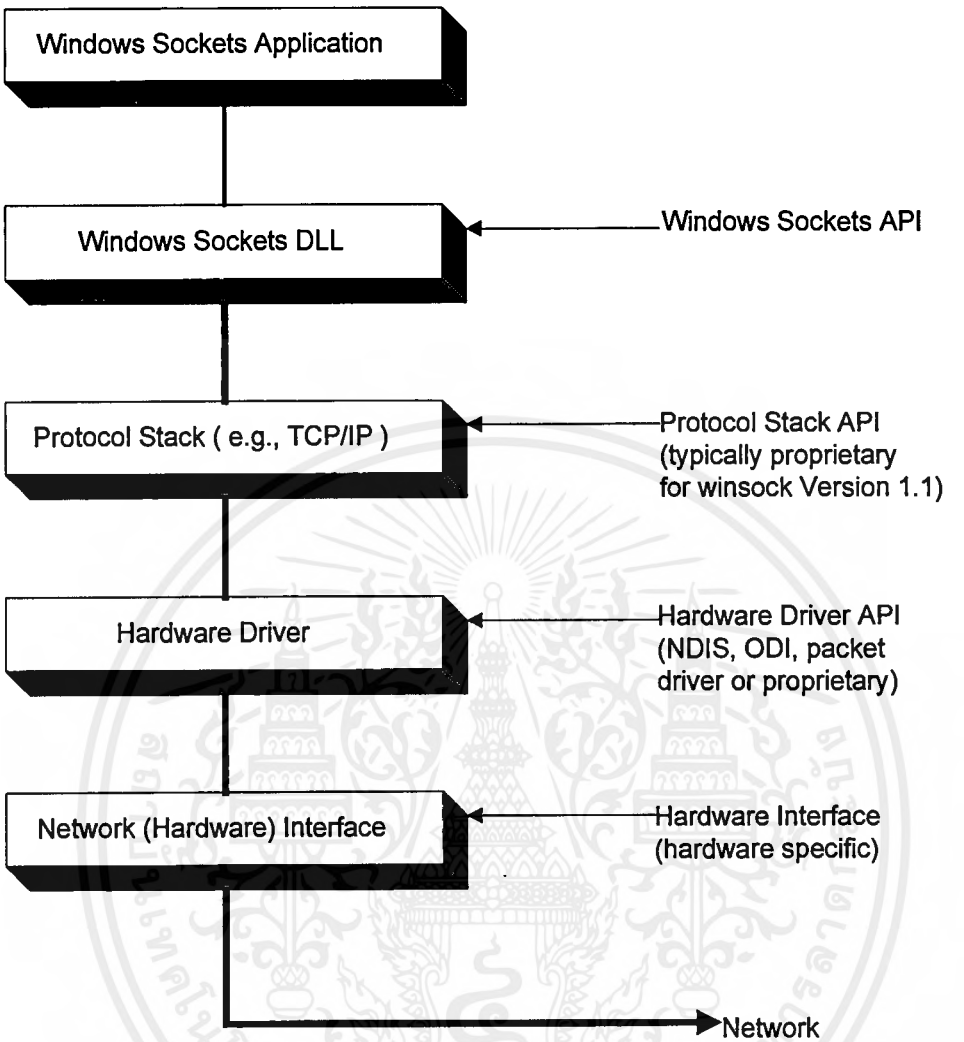
ลักษณะและรูปแบบของ WinSocks ถูกสร้างโดยตรงจากการร่วมมือกันของบริษัทที่เชี่ยวชาญทางด้าน Network หลายบริษัทด้วยกัน เริ่มต้นความคิดจากการเสนอกันของบุคคลหลายกลุ่มด้วยกัน ซึ่งเกิดขึ้นในปี 1991 กลุ่มบุคคลเหล่านั้นประกอบไปด้วย Martin Hall (JSB), Mark Towfig (Micro Dyne Corporation) Geqf Arnold (Sun Microsystems) , David Treadwell (Microsoft) และ Henry Sanders (Microsoft) ซึ่งเป็นผู้สร้างเอกสารและรับผิดชอบการดูแลและบริหาร และสามารถเปิดการติดต่อสำหรับ Network Programming ภายใต้ Microsoft Windows Version 1.1 ได้สำเร็จเรียบร้อยเมื่อวันที่ 20 มกราคม 1993 และได้มีการใช้กันอย่างกว้างขวางในบริษัทและในสถานศึกษา

WinSock ได้พัฒนาให้ง่ายขึ้นอย่างต่อเนื่องเรื่อยมา โดยกลุ่มที่เรียกว่า WinSock Group ซึ่งงานที่ทำปัจจุบันก็ได้แก่การดำเนินงานต่อเนื่องใน Windows Socket2 ซึ่งหน้าที่หลักก็คือการให้การสนับสนุนและการเพิ่ม Protocol Stacks ความสามารถที่มีอยู่ใน Winsock2 ประกอบด้วย

1. สนับสนุนการส่งที่เป็นอิสระสำหรับ Multiple Networks Stacks เช่น TCP/IP,IPX/SPX, DECnet และ OSI
2. Qos (Quality of Service) คือการให้บริการที่มีคุณภาพ ความสามารถในการเข้าถึง Function ที่ใช้งานใน Wireless, ATM, และ ISDN Network และรูปแบบ การพัฒนา MultiEngine
3. สนับสนุน Protocol ที่เป็นอิสระแบบ Multipoint และ Multicast
4. ลักษณะถึงความสามารถ ที่เร็วของ Client และ Server Application

2.5.1 ความสัมพันธ์ระหว่าง WinSocks กับ Network

WinSock เป็นลักษณะ Application Program Interface (API) ที่กำหนดการเรียกใช้โปรแกรม (Program Call) เพื่อใช้ในการเข้าถึง Function ใน Network ได้เช่น Winsock.DLL Wsock32.DLL ซึ่งเป็นตัวแสดงถึงลักษณะเฉพาะถึงการนำ Procedure เข้ามาใช้ในรูปแบบของ WinSock API ซึ่งสามารถเขียนแผนผังแสดงความสัมพันธ์ระหว่าง Winsock.DLL กับส่วนอื่นของ Network ได้ดังรูป 2.10



รูปที่2.11 แสดงแผนผังความสัมพันธ์ระหว่าง Winsock.DLL กับ Network

2.5.2 ลักษณะการทำงานของ WinSocks

WinSocks Application จะทำงานบนระบบ Windows3.1 หรือใน Windows ที่สูงกว่า ซึ่งเราจะต้อง Installed Winsock.DLL และรวมไปถึง Networking Stack ทั้ง Winsock.DLL รวมถึง Networking Stack สามารถที่จะได้มาจากหลายๆ ผู้จำหน่าย (Supplier) ด้วยกัน ซึ่งแต่ละผู้จำหน่ายก็จะมีลักษณะเฉพาะในด้านการนำไปใช้งานหรือรูปแบบด้านการสื่อสารเป็นของตัวเอง ลักษณะของ Memory Configuration จะมีความแตกต่างกันไปตามเวนเดอร์ (Vender) บาง Vender ก็ต้องการ DOS Memory

เมื่อ WinSock Application ทำงาน Windowsจะพยายามค้นหา Winsock.DLL โดยการค้นหาตามตำแหน่งเป็นลำดับขั้นดังนี้

1. The Current Directory
2. The Windows Directory
3. The Windows Systems Directory
4. รายการ Directory บางรายการที่เป็น Path ของสภาพแวดล้อมของตัวแปร
5. Directory บางตัวที่อยู่ในรายการ Directory ที่มีการ Mapped ใน Network มันจะค้นหาตามลำดับ จนพบ Winsock.DLL ซึ่งในบางครั้งเราไม่จำเป็นต้องให้ File นี้ชื่อ WinSock เสมอไปก็ได้ ถ้ามีชื่อนี้แล้วจะเกิดปัญหาตามมา เราสามารถจะเปลี่ยนชื่อ Winsock.DLL เป็นอะไรก็ได้ นามสกุล DLL เพื่อหลีกเลี่ยงปัญหาที่จะเกิด ก็ถือว่าถูกต้องเช่นกัน

2.5.3 WinSocks Functions

- คำสั่งที่ใช้ในการจัดตั้งการเชื่อมต่อแบบ Stream Socket

Socket() = สร้าง Communication Endpoint

Bind () = รวม Address เข้ากับ Socket กระทำบนฝั่ง Server

- คำสั่งด้าน Server

Listen() = Sets Up ลำดับถึงการรับการร้องขอเข้ามา เพื่อรอการเชื่อมต่อจาก Clients

Accept() = รับการร้องขอการเชื่อมต่อ ถ้ามันเป็นลำดับแรกมันก็จะรับเดี๋ยวนั้นเลย

- คำสั่งด้าน Client

Connect() = ร้องขอการเชื่อมต่อกับ Remote Address

- คำสั่งที่ใช้แลกเปลี่ยนข้อมูลบนการเชื่อมต่อแบบ Stream Socket

Send() = การส่งข้อมูล สังเกตุว่า TCP จะจัดการเชื่อมต่อข้อมูลแบบ Stream Buffer ของข้อมูลจะถูกกำหนดขึ้น เมื่อมีการส่ง ข้อมูลซึ่งเป็นลักษณะพื้นฐานใน TCP Algorithms

Recv() = การรับข้อมูล สังเกตุว่ามันจะนำข้อมูลทั้งหมดขึ้นมาตามขนาดที่รับเข้ามา

- คำสั่งที่ใช้ในการปิดการเชื่อมต่อแบบ Stream Socket

CloseSocket() = ปิดการเชื่อมต่อของ Socket และปล่อยทรัพยากร Socket

Shutdown() = ให้หยุดการเคลื่อนย้ายของข้อมูลในทิศทางเดียวหรือทั้งสองทิศ

ทาง เช่น Socket สามารถ Shutdown การส่ง หรือ Shutdown การรับ หรือ Shutdown ทั้งการส่งและการรับ แต่อย่างไรก็ตามเราสามารถให้ความแน่นและปลอดภัยต่อ Data ทั้งหมดก่อนโดยการส่งและรับข้อความก่อนที่จะทำการปิด Socket

- คำสั่งที่ใช้สำหรับ Datagram Sockets

Socket()	= จัดตั้ง Communication Endpoint
Bind()	= รวม Socket เข้ากับ Address
Sendto()	= ส่งข้อมูลไปตาม Address
Recvfrom()	= รับข้อมูล
CloseSocket()	= ปลดปล่อยทรัพยากร Socket
Connect()	= เป็นคำสั่งที่เพิ่มเข้ามาใช้ในการรวม Remote Address กับ Datagram Socket ซึ่งสามารถใช้คำสั่ง Send() และ Recv()ได้

- คำสั่งที่ใช้ในการรับหรือตั้งค่าให้กับ Socket

GetpeerName()	= ค่าที่ได้กลับมาเป็น Address ของ Remote Socket ได้มาเพื่อการเชื่อมต่อ Stream Sockets หรือการรวม Socket (และมีบางกรณีใช้กับ Datagram Socket ได้)
Getsockopt()	= ค่าที่ได้กลับมาเป็นค่าเฉพาะที่ เป็นส่วนที่เพิ่มเข้ามาของ Socket เช่น Socket Type, Socket Linger Option
Setsockopt()	= เป็นการตั้งค่าเฉพาะที่เป็นส่วนที่เพิ่มเข้ามาของ Socket
IoctlSocket()	= อนุญาติให้คุณตั้งค่า Socket Blocking Mode
Select()	= นำเอาส่วนที่เป็น Information เมื่อ Socket พร้อมและเชื่อมต่อเรียบร้อยแล้วหรือเพื่อการอ่านหรือเขียน

- คำสั่งที่ใช้เปลี่ยน Byte Order

ntohl()	= เปลี่ยนค่า 32 bit จาก Network เป็น Host
ntohs()	= เปลี่ยนค่า 16 bit จาก Network เป็น Host
htonl()	= เปลี่ยนค่า 32 bit จาก Host เป็น Network
htons()	= เปลี่ยนค่า 16 bit จาก Host เป็นNetwork

- คำสั่งที่ใช้ในการปรับเปลี่ยน IP Address

inet_addr()	= เปลี่ยนจาก IP Address ในรูปแบบตัวเลขที่มีจุดไปเป็น 4 Byte Hex ในรูปแบบ Network Byte

inet_ntoa() = เปลี่ยนจาก IP Address ในรูปแบบ 4 Byte Hex ไปเป็นตัวเลขที่มีจุด

- คำสั่งที่นำไปใช้กับ Database Information

GetHostbyaddr () = การรับ Host Name, Address, และ Aliases ที่รวมกับ Host Address

GetHostbyName() = การรับ Address Name , Name และ Aliases รวมกับ Protocol Name หรือ Alias

GetHostName() = การรับ ชื่อจาก Local Host

GetprotobyName() = การรับ Protocol Number,Name,Number และ Aliases ที่รวมกับ Protocol Number

GetservbyName() = การรับ Service Port Number,Name, และ Aliases ที่รวมกับ Service Name หรือ Alias

GetservbyPort() = การรับ Service Name, Port Number, และ Aliases ที่รวมกับ Port Number

บทที่3

ทฤษฎีและการออกแบบ Distributed Application

ในทางทฤษฎีในการออกแบบหรือเขียนโปรแกรมที่มีลักษณะการทำงานแบบ Client/Server นั้นจะมีความแตกต่างจากการเขียนโปรแกรมที่ใช้กันทั่วๆ ไปที่ใช้บน Local Machine คือ การเขียนโปรแกรมหนึ่งเป็นตัวส่งตัวแปร (Variable) บนฝั่ง Client และโปรแกรมที่เป็นตัวตอบสนองและรับค่าตัวแปรบนฝั่ง Server ให้สามารถทำงานร่วมกันได้ระหว่าง Client กับ Server จึงต้องมีการกำหนดค่าตัวแปรบางส่วนที่ใช้ร่วมกันให้สอดคล้องกัน เช่น ลักษณะค่าตัวแปรที่จะใช้ส่งไปมาระหว่าง Client กับ Server ลักษณะการส่งข้อมูลหมายเลขของ Process (Port Number) เป็นต้น

3.1 ทฤษฎีและการออกแบบโปรแกรมบน Client

3.1.1 การ Allocating Socket

โปรแกรมที่จะติดต่อระหว่าง Client กับ Server จำเป็นต้องมี Endpoint Address สำหรับ Socket เพื่อใช้ในการติดต่อโปรแกรมที่ทำงานบน Server จะมีการกำหนด Protocol ที่ใช้ติดต่อกับ Client และหมายเลข Port Address ประจำ Process นั้น ซึ่ง Process บน Client จำเป็นต้องทราบเพื่อใช้ติดต่อกับ Process บน Server ได้ถูกต้อง การเลือก Port Number จะต้องเลือกหมายเลข Port Address ที่ Process บนเครื่องนั้นใช้อยู่และไม่เป็นหมายเลข Port ที่ถูกกำหนดให้กับ Service Process ส่วน IP Address ก็เป็นส่วนสำคัญอีกเรื่องหนึ่งที่ต้องมีบน Network เพื่อใช้ติดต่อระหว่างเครื่อง Computer โปรแกรมที่ใช้ TCP Protocol ในการติดต่อจะใช้ Endpoint เพื่อสร้าง Connection TCP Protocol จะใช้ IP ของปลายทางเพื่อหาเส้นทางติดต่อตาม Function ของ Socket ก่อนแล้วจึงส่งสัญญาณกลับมายัง IP Address ของต้นทางเมื่อไปถึง IP Addressปลายทาง หลังจากนั้นจึงทำการส่งข้อมูลกันต่อไป

3.1.2 การติดต่อผ่าน TCP Protocol บน Client

สมมุติว่ามีการติดต่อระหว่าง Client กับ Server ได้แล้ว Application Protocol จะทำการส่งสัญญาณ Request - Response Interaction โดยทาง Client จะส่งสัญญาณขอลำดับ (Sequence) แล้วก็รอสัญญาณบอกลำดับที่จะกลับมาจากฝั่ง Server โดยปกติ Client จะใช้ Write Function เพื่อใช้ในการส่งข้อมูลและร้องขอ ส่วน Read Function จะใช้เป็นตัวรอการตอบสนอง

```
/*****************************************************************************/
        /* Example Code Segment */
#define BLEN 120    /* buffer length to use  */
char    *req = "Request of some sort";
char    buf[BLEN];      /* buffer for answer        */
char    *bptr;          /* pointer to buffer        */
```

```
int     n;              /* Number of Bytes read */
int     buflen          /* space left in buffer     */
bptr = buf;
buflen = BLEN;
/* send Request */
write(s, req, strlen(req));
/* read Response (may come in many pieces) */
while ((n = read(s, bptr, buflen) > 0)
{               bptr += n;
                buflen -= n;
}
/*************************************************************************/
```

จาก Code ตัวอย่างเป็นการส่งข้อความเล็กๆ โดย Client ไปยัง Server และตอบรับด้วยสัญญาณสั้นๆ ไม่เกิน 120 Bytes ภายในตัวอย่างจะประกอบด้วยสัญญาณที่ถูกส่งผ่านโดย Write Function และใช้ Read Function วนลูป (Loop) เพื่อรอรับข้อมูล เมื่อรับข้อมูลแล้วจะทำการลดค่าใน Buffer และนำข้อมูลใน Buffer ออกมาแสดง หรือต้องทำซ้ำในการรับข้อมูลในกรณีเกิดความเสียหายหรือสูญหายของข้อมูลที่ได้รับ ในการส่งข้อมูลของ TCP Protocol จะส่งข้อมูลขนาดเล็กเพราะ TCP Protocol ไม่ได้เป็น Block-Oriented Protocol แต่เป็น Stream-Oriented Protocol มันจะรับประกันลำดับและป้องกันความสูญหายของข้อมูล โดย Write Function ก่อนการส่งข้อมูล TCP จะทำการกำหนดขนาดของ Segment ที่จะใช้ข้อมูลจากคำนวณจากขนาดของข้อมูลใน Buffer ทางด้าน Application ที่เป็นฝ่ายรับได้รับข้อมูลขนาดเล็ก เมื่อข้อมูลได้รับเป็นปกติก็จะส่งสัญญาณไปบอกฝ่ายส่งว่าปกติด้วย Write Function แต่ถ้าข้อมูลที่ได้รับมีขนาดใหญ่ก็จะส่งผลการรับข้อมูลออกไปเป็นลำดับไปยังฝ่ายส่ง ด้วย Write Function เหมือนกัน

3.1.3 การติดต่อผ่าน UDP Protocol บน Client

โปรแกรมบน Client ที่ใช้ UDP ในการติดต่อมีพื้นฐานอยู่ 2 แบบ คือ Connected และ Unconnected ในสภาวะที่ Client สร้าง Connected จะเป็นการเรียกใช้ Remote Endpoint (ภายใน Remote Endpoint จะประกอบไปด้วย IP Address และ Protocol Port Number ของ Server) ตัว Remote Endpoint จะถูกใช้เป็นตัวรับและส่งข้อมูลของ Client ส่วนในสภาวะ unconnected จะเป็นการเลิกติดต่อของ Client ในการส่งข้อมูล Client ก็จะส่ง Remote Endpoint ไปยังปลายทางเหมือนการส่งข้อความออกไป การส่งจะไม่ยุ่งยากโดยโปรแกรมบน Client จะติดต่อกับ Server เพียงที่เดียวแล้วส่งข้อมูลทั้งหมดหลายๆ ชุดในเวลาเดียวกัน เวลาทำการ Unconnected ก็จะส่งสัญญาณยกเลิกการติดต่อไปยัง Server แล้วรอจากตอบรับจาก Server ก็ถือเป็นการจบการติดต่อ

การ Connected ของ UDP Client เป็นการส่งชนิด Datagram จึงไม่มีการ Initials ก่อนการแลกเปลี่ยนข้อมูลและไม่มีการทดสอบการติดต่อของ Remote Endpoint Address เวลาที่โปรแกรมใช้

Sock_Dgram Socket ทำการ Connected ภายในจะมีเพียง Address ของปลายทาง การทำ Connected สำเร็จ จึงไม่ได้หมายถึง Remote Endpoint Address ถึงปลายทางสำเร็จ

การส่งข้อมูลของ UDP Client ก็จะใช้ Write Function และ Read Function ในกรณีรอรับการตอบรับจาก Server การทำการ Read Function จะรับข้อมูลขนาดใหญ่เมื่อรับข้อมูลแล้วก็จะกลับไปรับข้อมูลต่อไปโดยไม่มีการกลับมารับข้อมูลเดิมอีก

3.2 การออกแบบโปรแกรมบน Server

ทฤษฎีและรูปแบบของ Server คือการสร้างและ Bind Socket เพื่อขอการติดต่อจาก Client ตามหมายเลข Port โดยจะทำการวนลูปแบบ Infinite Loop เพื่อรอการติดต่อจาก Client และเมื่อได้รับการติดต่อจาก Client ก็จะทำการตอบรับ Client ต่อไป แต่ในการทำงานบนระบบ Multi User จะต้องมีการติดต่อกับ Client หลายตัวในเวลาเดียวกันจึงต้องมีรูปแบบการทำงานของ Server ที่แตกต่างรูปแบบกันออกไปตามสถานการณ์

3.2.1 Concurrent และ Iterative Server

-Iterative Server หมายถึง Server ที่ถูกออกแบบมาให้ Process หนึ่งสามารถรับการร้องขอได้ครั้งละหนึ่ง Client เท่านั้น

-Concurrent Server หมายถึง Server ที่ถูกออกแบบให้ Process หนึ่งสามารถรับการร้องขอได้หลาย Client ในเวลาเดียวกัน

ในการจัดการเรื่อง Concurrent บน Server จะเป็นเรื่องของ Application Protocol ประสิทธิภาพของ Server มีจำนวนน้อยที่จะขึ้นอยู่กับประสิทธิภาพของ I/O ใน Single Server การทำงานของ Process จะใช้ asynchronous I/O ไปพร้อมกับการติดต่อสื่อสาร ใน Server ที่ต้องติดต่อกับ Client หลายๆ ตัวจะต้องมีการจัด Concurrency โดย Concurrent Server จะมีการให้ Handle Multiple Request Concurrency โดยไม่เกี่ยวกับ Multiple Concurrent Process ของระบบปฏิบัติการ เพราะฉะนั้นการออกแบบและสร้าง Concurrent Server จึงมีความยากลำบากกว่าและเหตุผลที่เราไม่นำ Iterative Server มาใช้เป็น Concurrent Server เพราะ Iterative Server ไม่สามารถหน่วงเวลา (Delay) ให้กับ Distributed Application และจะเกิดปัญหาคอขวด (Bottleneck) ในกรณีมีโปรแกรมของ Client หลายตัว

3.2.2 Connection-Oriented Server

ประโยชน์หลักของการติดต่อแบบ Connection-Oriented คือ Transport Protocol มี Handle Packet เพื่อป้องกันการสูญหายของข้อมูลและตรวจสอบลำดับ,การสูญหายข้อมูลด้วยการทำงานของ Transport Protocol เองโดยที่ Server ไม่ต้องสนใจในเรื่องนี้ เมื่อ Server ได้รับการติดต่อจาก Client จะสร้างช่องทางการติดต่อกับ Client ต่อจากนั้นจึงทำการส่งข้อมูลติดต่อกับ Client เมื่อต้องการเลิกการติดต่อ Server จะเป็นตัวปิดการติดต่อกับ Client การติดต่อเราจะใช้ TCP Protocol จึงมีความเชื่อถือในการติดต่อ

มีความแก้เมื่อเกิดความผิดพลาดและมีการจัดลำดับของข้อมูลได้อย่างถูกต้อง แต่ถ้า Client เกิดขาดการติดต่อหรือ System Down ทางด้าน Server ก็จะปิดการติดต่อกับ Client เช่นกัน

Connection-Oriented การออกแบบจะแยก Socket ออกจากกัน แต่ Connectionless จะใช้การติดต่อหลายๆ Hosts โดยใช้เพียง Socket เดียว ในช่วงที่มีการกำหนด Socket ขึ้น หรือจะควบคุมการสร้าง Connection จะต้องเป็นช่วงที่ OS ทำงานเต็มประสิทธิภาพ เมื่อเปรียบเทียบประสิทธิภาพระหว่าง UDP กับ TCP แล้ว TCP จะช้ากว่าในช่วงแรกเริ่ม และตอนยกเลิกการติดต่อแต่ Server ก็สามารถจะรู้ได้ว่า Client เป็นปกติหรือไม่ในช่วงที่มีการตอบรับการส่งข้อมูลระหว่างกันของ Client กับ Server ได้ตลอดเวลาที่มีการติดต่อ ซึ่ง Server จะตอบรับทุกสัญญาณที่ส่งมาจาก Client ตัวอื่นๆ เพราะฉะนั้นในส่วนของโปรแกรมจะต้องให้ Loop รับการติดต่อจากภายนอกตลอดเวลาเช่นกัน

3.2.3 Connectionless Server

ในการรับส่งข้อมูลแบบ Connectionless จะมีทั้งข้อดีและข้อเสีย ใน Connectionless Server จะไม่ค่อยมีปัญหายุ่งยากเพราะมันจะยุ่งเกี่ยวกับปัญหาที่เกิดขึ้นใน Layer ที่อยู่ต่ำกว่า Transport Layer โดยที่จะมีการส่งข้อมูลซ้ำให้กับ Client ในกรณีที่มีการส่งสัญญาณมาบอกว่าข้อมูลไม่ถึงปลายทางเท่านั้นทาง Server จะทำการแบ่งข้อมูลออกเป็นส่วนย่อยเพื่อเพิ่มประสิทธิภาพในกรณีที่ต้องมีการส่งข้อมูลซ้ำ ในการออกแบบจึงต้องอาศัยประสบการณ์มากเพราะ TCP/IP จะทำงานภายใต้สภาพแวดล้อมที่เป็น Internet ส่งข้อมูลเป็นจุดต่อจุด เปลี่ยนแปลงรวดเร็ว และมีการกำหนดตายตัวในการทำงาน โปรแกรมเมอร์หลายคนถึงต้องเรียนรู้และทดลองทำอย่างหนักซึ่งในการเขียนโปรแกรมบน LAN จะใช้เวลาทำงานไม่มากและมีค่าตัวแปรไม่มากนัก ส่วนในระบบ Internet มีการรับส่งข้อมูลนานกว่าและมีตัวแปรมากมายในการทำงาน เพราะฉะนั้นการส่งข้อมูลซ้ำจึงต้องมีการคิดวิธีไว้อย่างดีโดยต้องอาศัยความซับซ้อนของ TCP Protocol ช่วยในการทำงาน

3.3 การออกแบบโปรแกรมโดยใช้ RPC

3.3.1 การออกแบบ Distributed Program

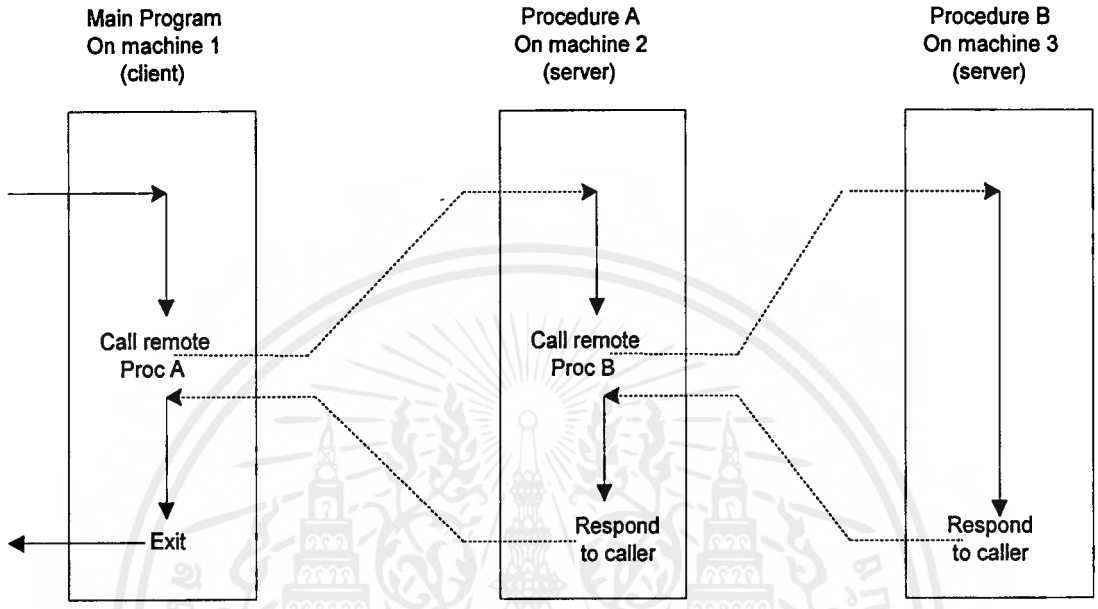
Programmer สามารถแบ่งการออกแบบ Distributed Application ได้ 2 ลักษณะ

-Communication-Oriented Design เป็นการออกแบบการติดต่อสื่อสารในลักษณะ Message Format และ Syntax ระหว่าง Client และ Server

-Application-Oriented Design เป็นการออกแบบ Application โดยการออกแบบจะเป็นไปในลักษณะมีระเบียบแบบแผนตามเงื่อนไข การออกแบบจะเริ่มสร้างและทดสอบบน Single Machine ก่อนแล้วจึงทำการแยกโปรแกรมเองเป็นส่วนๆ เพื่อนำไปทำงานบนเครื่องต่างๆ ในลักษณะ Client-Server ต่อไป

Remote Procedure Call Model มีลักษณะใกล้เคียง Application-Oriented ซึ่งจะช่วยแก้ปัญหาทางด้านการติดต่อสื่อสารได้ Programmer จะต้องนำ RPC มาช่วยในการสร้างร่วมกับ Application ก่อนการ

แบ่งโปรแกรมเพราะ RPC จะเป็น Procedure ที่แบ่งในโปรแกรมในส่วนของ Local และ Remote ตามโครงสร้างหลังของโปรแกรมเดิมโดย RPC จะทำในส่วนที่เป็น Distributed Environment



รูปที่ 3.1 แสดงการทำงานของ RPC Model ด้วย Remote Procedure Call.

เมื่อเวลาที่ Remote Program จะทำการติดต่อสื่อสารจะต้องติดต่อผ่าน UDP หรือ TCP ซึ่งมี Port Number ขนาด 16 bits เป็น Endpoints ในตอนแรกทางด้าน Server จะต้องทำการ Passive Socket และ Bind Socket เข้ากับ Protocol Port เพื่อรอการติดต่อจากโปรแกรมทางด้าน Client Protocol Port Number จะถูกกำหนดเป็นจุดนัดหมายระหว่างโปรแกรมบน Server กับโปรแกรมบนฝั่ง Client ในระบบ Multi User ที่มีการติดต่อ การส่งผ่านข้อมูลมากมายบนเครื่อง computer เดียวกัน เราจึงมีการกำหนด Port Number เพื่อให้แต่ละ Computer แต่ละค่ายใช้ติดต่อร่วมกันกับโปรแกรมที่มีการทำงานเหมือนกันเพื่อให้ Computer ที่ต่างค่ายกันติดต่อกันได้

Name	Assigned Number	Description
Portmap	100000	Port mapper
rstatd	100001	rstat, rup, and perfmenter
rusersd	100002	Remote users
nfs	100003	Network File system
ypserv	100004	yp (now called NIS)
mountd	100005	mount, showmount
dbxd	100006	DBXprog (debugger)
ypBind	100007	NIS Binder
walld	100008	rwall, shutdown
yppasswdd	100009	yppasswd
etherstatd	100010	ethernet statistics
rquotad	100011	rquotaprog, quota, rquota
sprayd	100012	spray
selection_svc	100015	selection Service
dbsessionmgr	100016	unify, netdbmd, dbms
rexd	100017	rex, Remote_exec
office_auto	100018	alice
lockd	100020	klmprog
lockd	100021	nlmprog
statd	100024	status monitor
bootparamd	100026	bootstrap
pcnfsd	150001	NFS of PC

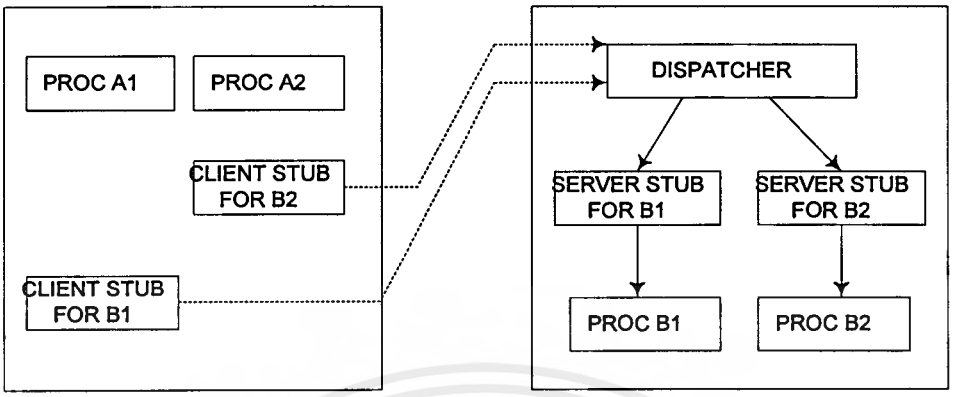
ตารางที่3.1 แสดงตัวอย่าง RPC Program Number ที่กำหนดโดย SUN Microsystem,Inc.

3.3.2 Marshaling

-Stub Code เป็น Special Code ที่เกิดขึ้นในขั้นตอนการใช้ RPC Application Development Tools (Rpcgen) Compiler XDR File จะได้ผลออกมาเป็น Object File สองFiles คือส่วนของ Client Stub และ Server Stub เพื่อใช้ Link กับ Main Program ในขั้นตอนต่อไป

เมื่อโปรแกรมทางด้าน Client มีการเรียกใช้ Client Stub เพื่อจะติดต่อกันโปรแกรมทางด้าน Server Client Stub จะทำการจัดเก็บ (Package Up) ข้อมูลที่ใช้ติดต่อกับ Remote Procedure แล้วจัดให้เป็น

รูปมาตราฐาน (External Data Representation) เพื่อส่งต่อไปบน Network ต่อไป ข้อมูล (Argument) ที่เกิดจากการจัดเก็บเราเรียกว่า Marshaling



รูปที่3.2 แสดง ตัวอย่างการติดต่อของ RPC ระหว่าง Client กับ Server โดยผ่าน Stub ของทั้งสองฝั่ง

3.4 External Data Representation (XDR)

สถาปัตยกรรมของ Computer แต่ละชนิดได้ออกแบบการเก็บข้อมูลที่ต่างกัน (Data Representation) เช่น การเก็บ Byte ของ Integer เก็บ Lowest Address Memory ก่อน บางเครื่องจะเก็บ Upper Address Memory ก่อน โดยที่ Comelier จะใช้ Native Data Representation เป็นตัวกำหนดและจัดเก็บ ในขั้นตอนการ Generate Code เพื่อใช้ Fetch,Store และเปรียบเทียบตัวแปร การเขียนโปรแกรมบน Client และ Server ตัวโปรแกรมจะต้องมีการแลกเปลี่ยนข้อมูล เพราะ Endpoint ของทั้งสองฝั่งต้องเห็นข้อมูลที่ติดต่อระหว่าง Client กับ Server จะเห็นเป็นลักษณะ Exact Representation ถ้า Native Data ของทั้งสองเครื่องมีความแตกต่างกัน การส่งข้อมูลเพื่อให้สามารถส่งตัวแปรระหว่างกันได้เป็นปกติของสถาปัตยกรรม

Data Type	Size	Description
int	32 bits	32 bits signed binary integer
unsigned int	32-bits	32 bits unsigned binary integer
bool	32 bits	Boolean value (false or true) represented by 0 or 1
enum	arb	Enumeration Type with values define by integers (e.g.,RED=1,WHILE=2,BLUE=3)
hyper	64 bits	64 bits signed binary integer
unsigned hyper	64 bits	64 bits unsigned binary integer
float	32 bits	Single precision floating point Number
double	64 bits	Double precision floating point Number
opaque	arb.	Unconverted data (i.e., data in the sender's native representation)
String	arb.	String of ASCII characters
Fixed array	arb.	A fixed-size array of any other data Type
Counted array	arb.	Array in which the Type has fixed upper limit, but individual arrays may vary up to that size
Structure	arb.	A data aggregate, like C's struc
Discriminated union	arb.	A data structure that allows one of several alter native forms, like C's union or Pascal's
void	0	Used if no data is present where a data item is Optional (e.g.,)
symbotic constant	arb.	A smpbolic constant and associated
Option data	arb.	Alloe zero or one accury of occurrences of item

ตารางที่3.2 แสดงข้อมูลในลักษณะต่างของ XDR

โดยปกติแล้ว Single Program จะถูก Compile และ Execute ตามลักษณะและโครงสร้างตามแบบของเครื่อง Computer นั้นๆ แต่ถ้ามีโครงสร้างของข้อมูลในลักษณะ Multiple Representation ซึ่งใช้บน Program บน Client หรือ Server จะทำการส่งข้อมูลไปยังโปรแกรมที่อยู่บนเครื่องอื่นๆ ซึ่งมีความแตกต่างของ Data Representation ทำให้เกิดปัญหาของการรับส่งข้อมูลระหว่างโปรแกรม เราจึงจำเป็นต้องมีการกำหนดมาตราฐานของข้อมูลในการส่งขึ้น (Standard Representation) ในการส่งข้อมูลในระบบ Network

ขึ้นโดยใช้ Protocol Header เป็น TCP/IP Protocol ซึ่งเราเรียกว่า Symmetric Data Conversion โดยโปรแกรมนี้ถูกกำหนดเป็นมาตราฐาน Protocol Header Represent Data บนทุกๆ Computer ต่อไป หลักการทำงานของ Symmetric Data Conversion จะถูกใช้เป็น Application Software ของเครื่อง Client และServer โดยเครื่อง Client หรือ Server ที่จะทำการส่งข้อมูลไปยังระบบ Network จะใช้ Symmetric Data Conversion แปลง Native Representation ให้เป็นแบบข้อมูลมาตราฐานเสียก่อน แล้วจึงส่งไปในระบบ Network เมื่อเครื่องที่รับข้อมูลแล้วก็จะทำการแปลงข้อมูลกับมาเป็น Native Representation เหมือนเดิม ข้อมูลที่ไปในลักษณะ Standard Representation ในระบบ Network จะเรียกว่า External Data Representation

การใช้ Standard External Data Representation มีทั้งประโยชน์และข้อเสียคือถ้ามีการใช้ XDR จะมีความยืดหยุ่นระหว่าง Client กับ Server คือ เครื่องบน Client และ Server สามารถติดต่อกันได้แม้ว่าสถาปัตยกรรมของ Client และ Server จะแตกต่างกัน โดยที่ Program บนฝั่ง Client และ Server ไม่จำเป็นต้องทราบสถาปัตยกรรมของอีกฝั่งหนึ่ง แต่ถ้าในกรณีที่เครื่องบน Client และ Server มีสถาปัตยกรรมที่เหมือนกันก็ไม่มีความจำเป็นต้องใช้ XDR เป็นตัวแปรง Native Data Representation ไปเป็น Standard External Data Representation เพราะเป็นการสร้าง overhead ให้กับการติดต่อระหว่าง Program บนฝั่ง Client กับ Server

โปรแกรมที่จะมารองรับการใช้ XDR เพื่อใช้ในการแปลง Symmetric Data จะต้องมีการดูแลข้อมูลใน External ก่อนการส่งลงไปบน Network เช่น โปรแกรมที่เป็นฝ่ายรับจะต้องทำหน้าที่แปลงข้อมูลที่รับมาเพื่อแปลงกลับเป็น Native Representation โดย XDR จะถูกเรียกใช้ในลักษณะ Function จาก XDR Library ตามตาราง3.2 ซึ่งจะเป็น Function ที่จำเป็นในการเปลี่ยนแปลงข้อมูล

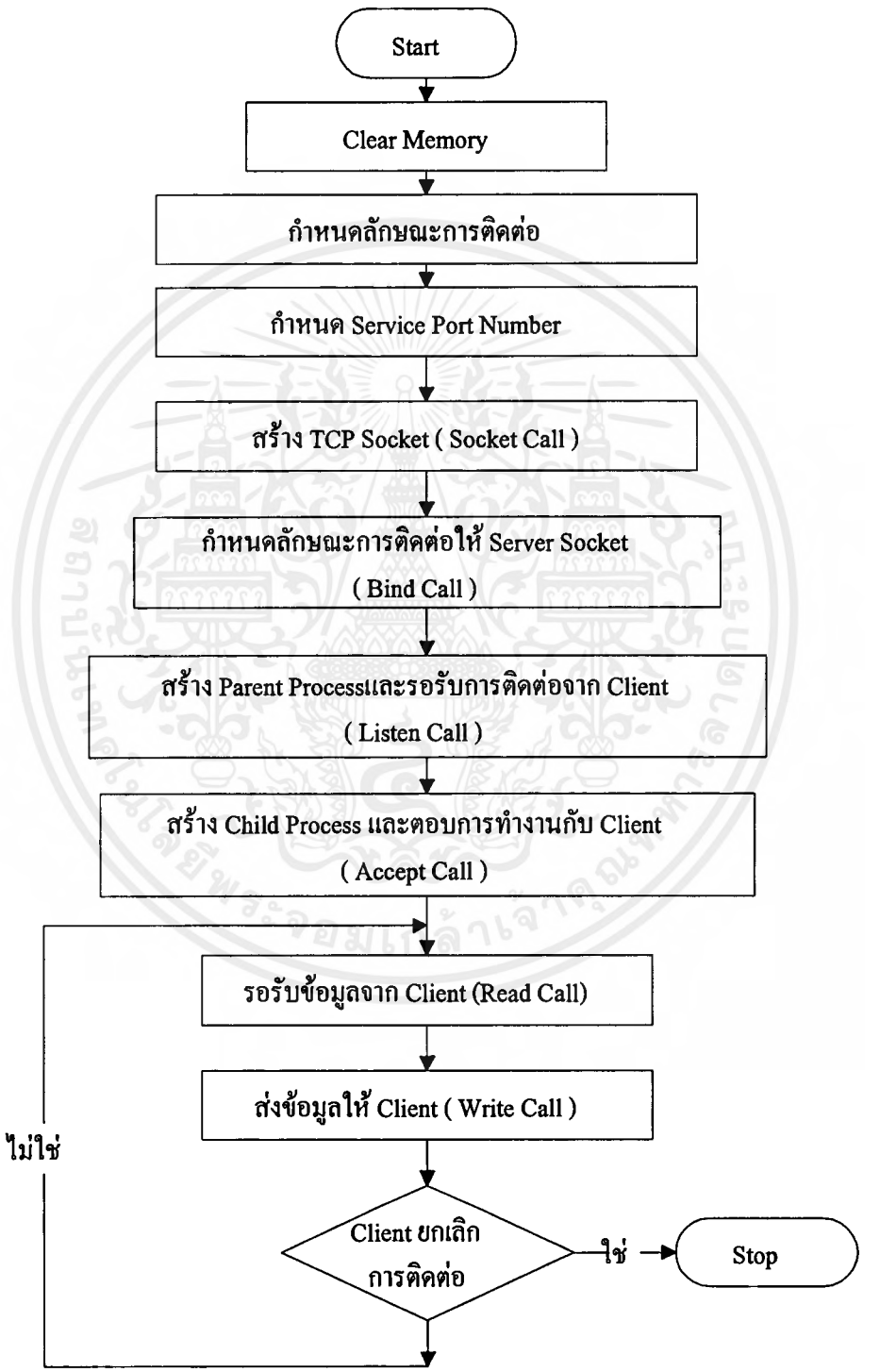
Procedure	Arguments	Data Type Converted
xdr_bool	xdrs,ptrbool	Boolean (int in C)
xdr_Bytes	xdrs,ptrstr	Counted Byte string
	strsize, maxsize	
xdr_char	xdrs,ptrchar	Character
xdr_double	xdrs,ptrdouble	Double precision floating point
xdr_enum	xdr, ptrint	Variable of enumerated Data Type (an int in C)
xdr_float	xdrs, ptrfloat	Single precision floating point
xdr_int	xdrs, ip	32-bit integer
xdr_long	xdrs,ptrlong	64-bit integer
xdr_opaque	xdrs, ptrchar, count	Bytes sent without conversion
xdr_pointer	xdrs,ptrobj, objsize,xdrobj	A pointer (used in linked Data structures like lists or trees
xdr_short	xdrs,ptrshort	16-bit integer
xdr_string	xdrs,ptrstr,maxsize	A C string
xdr_u_char	xdrs,ptrchsr	Unsigned 8-bit integer
xdr_u_int	xdrs,ptrint	Unsigned 32-bit integer
xdr_u_long	xdrs,ptrulong	Unsigned 64-bit integer
xdr_u_short	xdrs,ptrushort	Unsigned 16-bit integer
xdr_union	xdrs,ptrdiscrim,ptrunion, choicefcn,default	
xdr_vector	xdrs,ptrarray,size element,elemproc	Fixed length array
xdr_void	- none-	Not a conversion (used to denote empty part of a Data structure

ตารางที่3.3 แสดง XDR Data Conversion Routine ใน XDR Library

บทที่4

โปรแกรม Client/Server Implementation

4.1 แสดงขั้นตอนการทำงาน ของ TCP Server Program บน Unix



รูปที่4.1 Flowchart แสดงขั้นตอนการทำงานของ TCP Server Program บน Unix

โปรแกรม TCP Server บน Unix

```
/*  Stream socket for server implement  */
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<signal.h>
#include<stdio.h>
#include<netdb.h>
/* Declare Variable */
int s, ls;                    /* connected and listen socket descriptor */
struct hostent *client_name;          /* pointer to host information for remote hosts */
struct servent *service_port;         /* pointer to services information */
long timevar;                         /* contains time returns by time() */
char *ctime();                        /* declare time formatting routine */
struct  linger linger;                /* graceful close used setting SO-LINGER */
struct  sockaddr_in myaddr_in;          /* for local socket address */
struct  sockaddr_in peeraddr_in;    /* for peer socket address */

#define BUFLEN  1024        /* size of buffer */
main(argc, argv)
int argc;
char *argv[];
{
        int addrlen;
        /* clear out address structure */
        memset ((char *)&myaddr_in, 0,sizeof(struct sockaddr_in));
        memset ((char *)&peeraddr_in, 0,sizeof(struct sockaddr_in));
        /* setup address structure for socket */
        myaddr_in.sin_family = AF_INET;
        myaddr_in.sin_addr.s_addr = INADDR_ANY;
        service_port = getservbyname("port_tcp", "tcp");
        if(service_port == NULL){
```

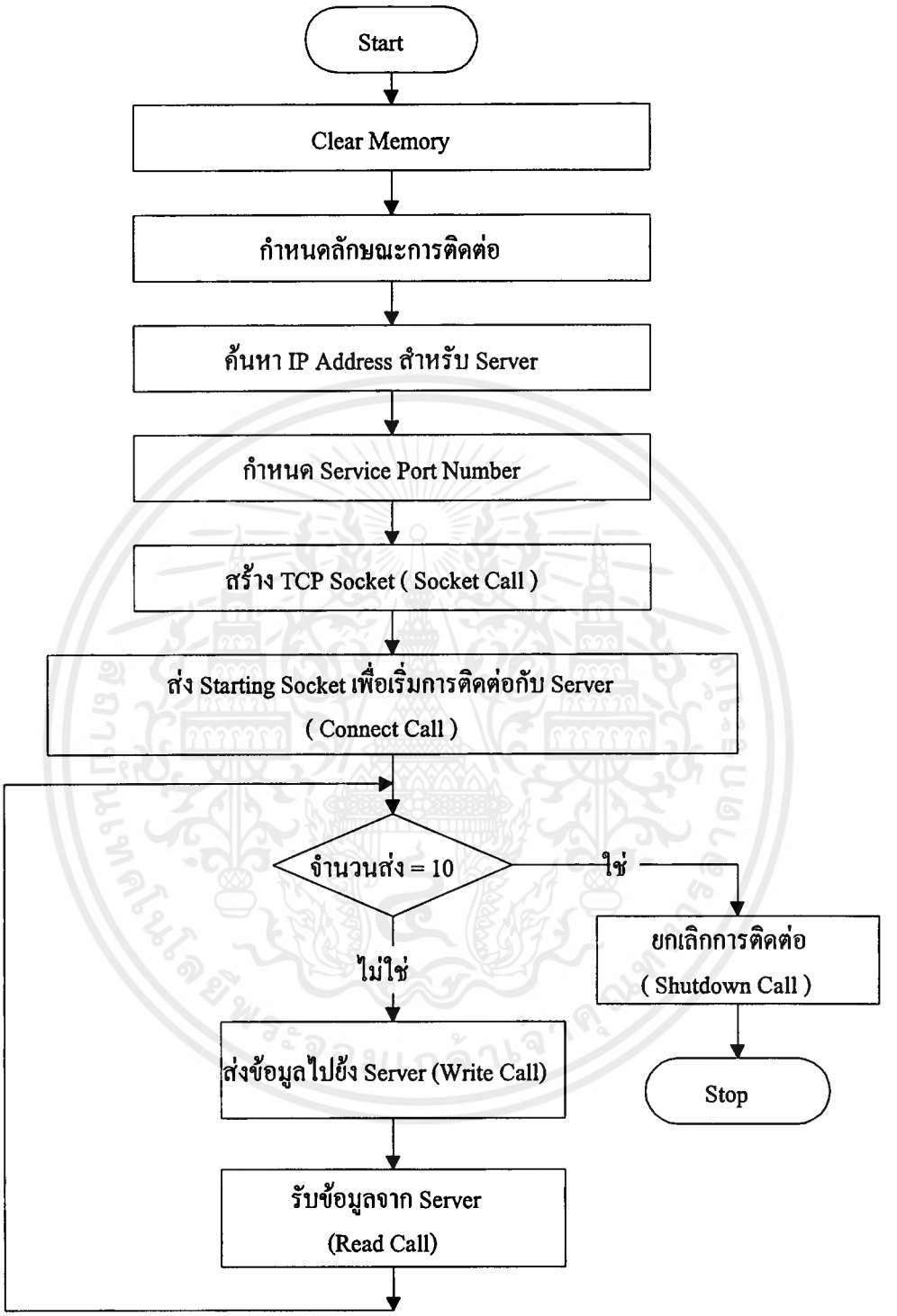
```
                fprintf(stderr, "%s: port_tcp not found in /etc/services \n",argv[0]);
                exit(1);
        }
    myaddr_in.sin_port = service_port->s_port;
    ls = socket(AF_INET, SOCK_STREAM, 0);
    if(ls == -1){
        perror(argv[0]);
        fprintf(stderr, "%s: unable to listen on socket\n", argv[0]);
        exit(1);
    }
        /* bind socket */
        if(bind (ls, &myaddr_in, sizeof(struct sockaddr_in)) == -1) {
                perror(argv[0]);
                fprintf(stderr, "%s: unable to bind address\n",argv[0]);
                exit(0);
        }
        /* listen socket so remote users can contact. */
        if(listen(ls, 5) == -1){
                perror(argv[0]);
                fprintf(stderr, "%s: unable to listen on socket\n", argv[0]);
                exit(1);
        }

    setpgrp();              /* set parent process */
        /* fork new process for new client connecting */
    switch(fork()) {
    case -1:
        perror(argv[0]);
        fprintf(stderr, "%s: unable to fork daemon\n", argv[0]);
        exit(1);
    case 0:
        fclose(stdin);
        fclose(stderr);
        signal(SIGCLD, SIG_IGN);
```

```
        for(;;) {
            addrlen = sizeof(struct sockaddr_in);
                    /*Accept call will be block until
                     *a new connection arrives,
                     *return the address  of peer connecting.
                     */
                    s = accept(ls, &peeraddr_in, &addrlen);
            if( s == -1) exit(1);
            switch(fork()) {
            case -1:
                exit(1);
            case 0:             /* Child process come here */
                server();
                exit(0);
                    /* Child process out of file descriptor space
                     * to close the new accept socket.
                     */
            default:
                close(s);
                    }

        }
    default:    /*parent process come here. */
        exit(0);
        }
}
        /* Server procedure handle the new process from
         * accept socket connecting from remote client.
         */
server()
{
    int reqcnt = 0;             /* keeps count of request */
    char buf[BUFLEN];           /* buffer keeps data 1024 bytes */
        char *inet_ntoa();
```

```
        char *hostname;              /* points to the remote hosts name */
    int len, len1;
        /* close the listen socket  */
    close(ls);
        /* Search Internet address from the remote client
         *by peer socket .
         */
    client_name = gethostbyaddr((char *) &peeraddr_in.sin_addr,
                    sizeof(struct in_addr),
                    peeraddr_in.sin_family);
    if(client_name == NULL) {
                hostname = inet_ntoa(peeraddr_in.sin_addr);
    } else {
        hostname = client_name->h_name;
     }
    time (&timevar);
    printf("Connect from %s CLIENT on port %u at %s",
        hostname, ntohs(peeraddr_in.sin_port), ctime(&timevar));
          /*  Set socket close ,when data sent on finish  */
    linger.l_onoff = 1;
    linger.l_linger =1;
    if(setsockopt(s, SOL_SOCKET, SO_LINGER, &linger,
            sizeof(linger)) == -1) {
errout: printf("connection with %s aborted on error \n", hostname);
    exit(1);
    }
        /*  When the remote client shutdown for sender, read call return zero  */
        while(len = read(s, buf, BUFLEN)){
                if(len == -1)       goto errout;        /* check no success recvice */
                sleep(1);
                reqcnt++;           /* Increment the request counter */
                time(&timevar);
                printf(" SERVER return %d time at %s", *buf, ctime(&timevar));
                /* write call send buf return to server  */
```

```
                len1 = write(s, buf, BUFLEN);
                if(len1 == -1) goto errout;  /* return buf to client */
                *buf = '\0';
        }
        /* close ls socket when connection finish */
    close(ls);
        /* print message finish */
    time(&timevar);
    printf("completed %s PORT %u, %d packets,TIME at %s \n",
        hostname, ntohs(peeraddr_in.sin_port), reqcnt, ctime(&timevar));
}
```

4.2 แสดงขั้นตอนการทำงาน ของ TCP Client Program บน Unix



รูปที่4.2 Flowchart แสดงขั้นตอนการทำงานของ TCP Client Program บน Unix

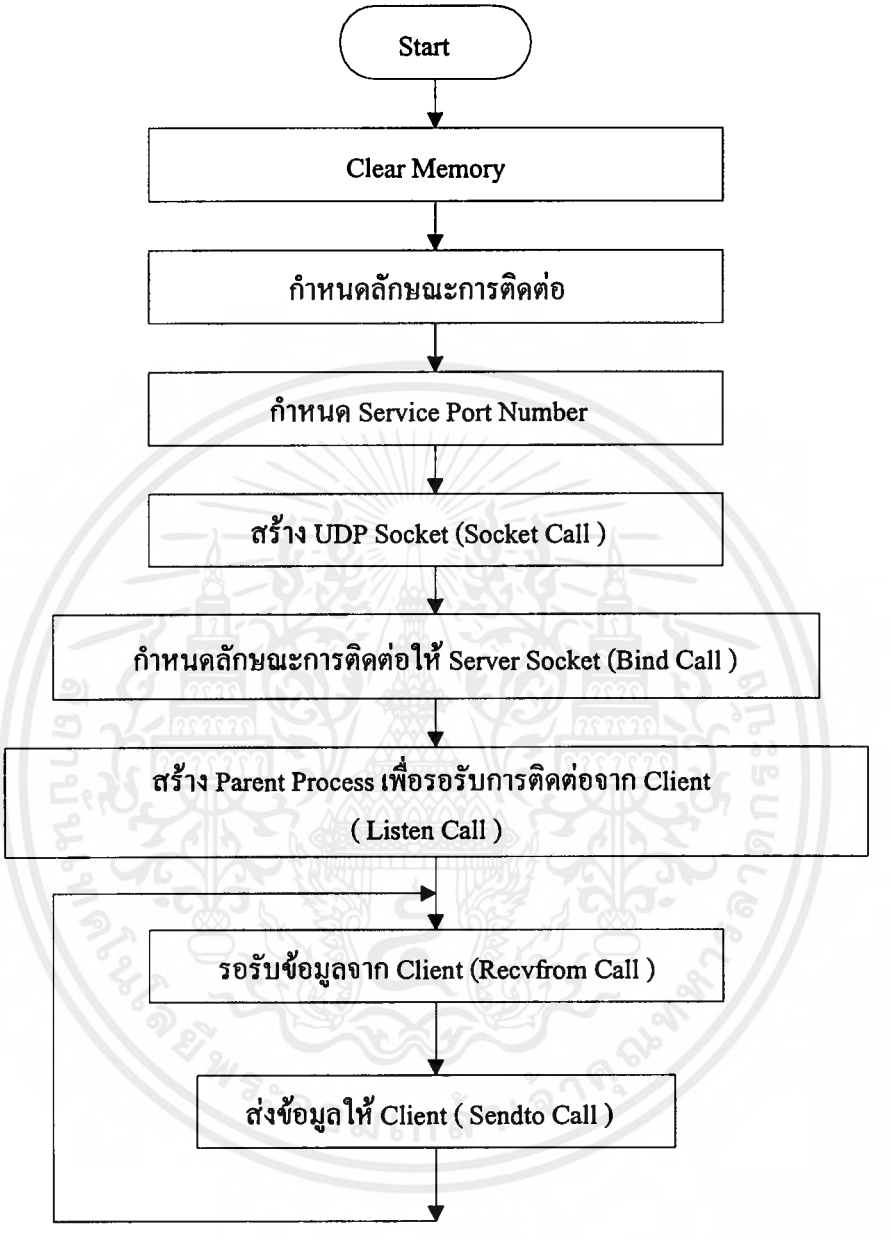
โปรแกรม TCP Client บน Unix

```
/* Stream socket for client implement */
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<stdio.h>
#include<netdb.h>
int s;                          /* connected socket descriptor */
struct hostent *server_name;    /* pointer for name's the remote host */
struct servent *service_port;   /* pointer for services information */
long timevar;                   /* contain time return time() */
char *ctime();
struct sockaddr_in myaddr_in;   /* local socket address */
struct sockaddr_in peeraddr_in; /* peer socket address */
#define BUFLEN 1024             /* size of buffer */
main(argc, argv)
int argc;
char *argv[];
{
    int addrlen, i, check;
    char buf[BUFLEN];
    if(argc != 2) {
        fprintf(stderr, "usage: %s<remote host>\n", argv[0]);
        exit(2);
    }
      /* clear out address structure */
    memset((char *)&myaddr_in, 0, sizeof(struct sockaddr_in));
    memset((char *)&peeraddr_in, 0, sizeof(struct sockaddr_in));
      /* Setup address structure socket for will connect */
    peeraddr_in.sin_family = AF_INET;
          server_name = gethostbyname(argv[1]);
    if(server_name == NULL) {
        fprintf(stderr, "%s: %s not found in /etc/hosts \n",
```

```
                argv[0], argv[1]);
        exit(1);
    }
    peeraddr_in.sin_addr.s_addr = ((struct in_addr *)(server_name->h_addr))->s_addr;
        /* Search number port from name port */
        service_port = getservbyname("port_tcp", "tcp");
    if(service_port == NULL) {
            fprintf(stderr, "%s: port_tcp not found in /etc/services\n",
                        argv[0]);
            exit(1);
    }
    peeraddr_in.sin_port = service_port->s_port;   /* get number port to endpoint */
        /* create  the socket */
    s = socket( AF_INET, SOCK_STREAM, 0);
    if(s == -1) {
        perror(argv[0]);
        fprintf(stderr, "%s: unable to create socket\n", argv[0]);
        exit(1);
    }
                /* Try connect to remote server at peeraddr */
    if(connect(s, &peeraddr_in, sizeof(struct sockaddr_in)) == -1) {
        perror(argv[0]);
        fprintf(stderr, "%s: unable to  connect to remote\n", argv[0]);
        exit(1);
    }
    addrlen = sizeof(struct sockaddr_in);
    if(getsockname(s, &myaddr_in, &addrlen) == -1) {
        perror(argv[0]);
        fprintf(stderr, "%s: unable to read socket address\n", argv[0]);
        exit(1);
    }
    time(&timevar);
    printf("Connect to %s SERVER on port %u at %s",
            argv[1], ntohs(myaddr_in.sin_port), ctime(&timevar));
```

```
        sleep(2);
                /* send buf and receives buf from the remote server */
        for(i=1; i<=10; i++) {
          *buf = i;
          check = write(s, buf, BUFLEN ) ;
                if( check == -1 ) {
              fprintf(stderr, "%s connection aborted on error ", argv[0]);
                fprintf(stderr, " on send number %d\n",i );
              exit(1);
         }
          check = read(s, buf, BUFLEN);
          if(check == -1) {
            perror(argv[0]);
            fprintf(stderr, "%s: errer reading result\n", argv[0]);
            exit(1);
          }
          printf("Received result number %d\n ", *buf);
            }
                /* Disconnect with the remote server */
            if(shutdown(s, 1) == -1) {
                    perror(argv[0]);
                fprintf(stderr,  "%s: unable to shutdown socket\n", argv[0]);
                exit(1);
            }
                /* Print message end of connection the remote server */
         time(&timevar);
         printf("All done at %s", ctime(&timevar));
}
```

4.3 แสดงขั้นตอนการทำงาน ของ UDP Server Program บน Unix



รูปที่4.3 Flowchart แสดงขั้นตอนการทำงานของ UDP Server Program บน Unix

โปรแกรม UDP Server บน Unix

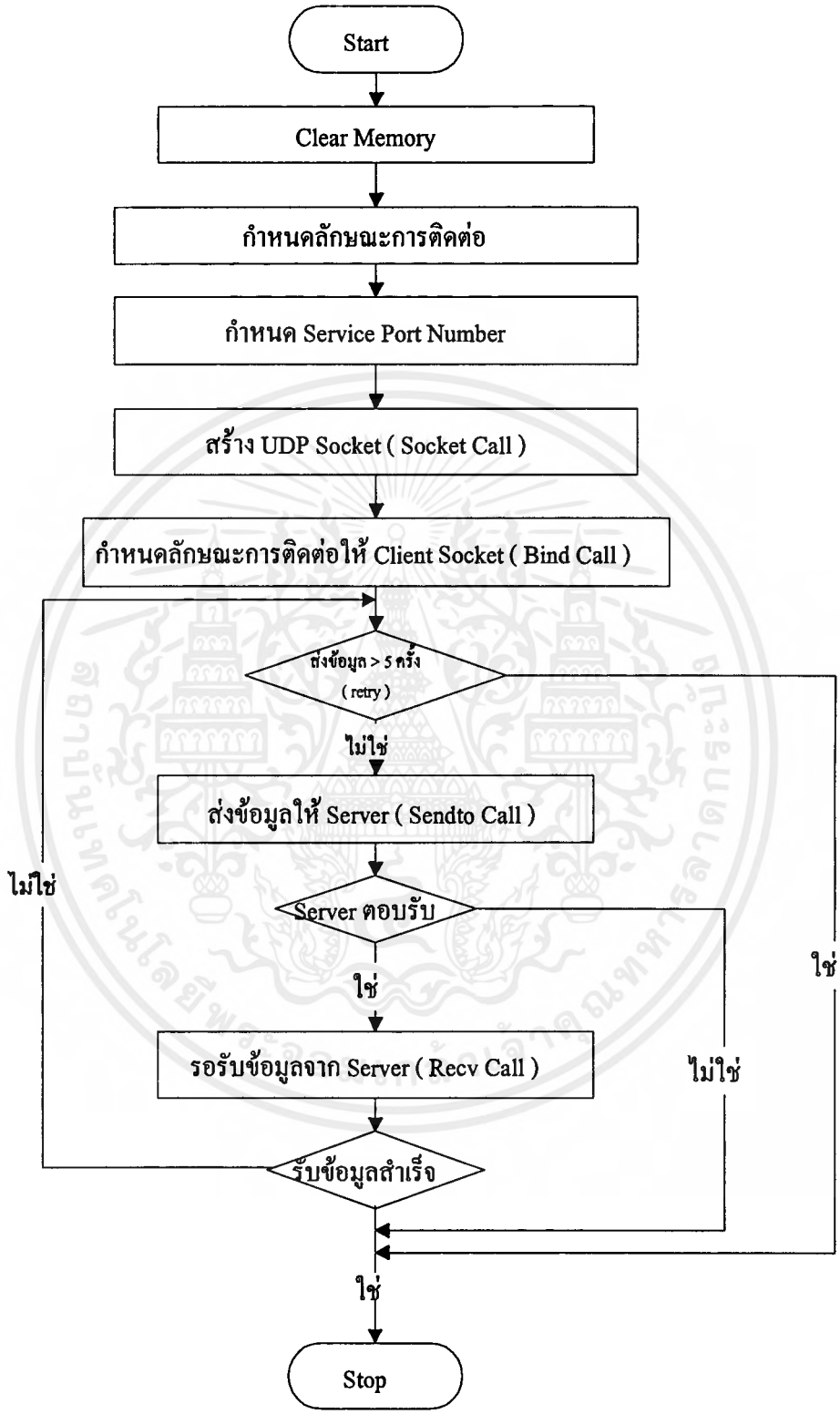
```
/* Datagram socket server implement */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>

int s;                          /* socket descriptor */
#define BUFFERSIZE      1024    /* maximum size of packets to be received */
int cc;                         /* contains the number of bytes read */
char buffer[BUFFERSIZE];        /* buffer for packets to be read into */
struct hostent *client_name;    /* pointer host information for the remote client */
struct servent *service_port;   /* pointer service information */
struct sockaddr_in myaddr_in;   /* for local socket address */
struct sockaddr_in clientaddr_in;   /* for client's socket address */

main(argc, argv)
int argc;
char *argv[];
{
        int addrlen;
        char *ctime();
        long timevar;           /* contain time return by time() */
        char *inet_ntoa();
        char *hostname;         /* points to remote host's name */
                /* clear out address structures */
        memset ((char *)&myaddr_in, 0, sizeof(struct sockaddr_in));
        memset ((char *)&clientaddr_in, 0, sizeof(struct sockaddr_in));
                /* Set up address structure for the socket. */
        myaddr_in.sin_family = AF_INET;
        service_port = getservbyname ("port_udp", "udp");
        if (service_port == NULL) {
```

```
            printf("%s: port_udp not found in /etc/services\n",
                                argv[0]);
            exit(1);
}
myaddr_in.sin_port = service_port->s_port;
            /* Create the socket. */
s = socket (AF_INET, SOCK_DGRAM, 0);
if (s == -1) {
            perror(argv[0]);
            printf("%s: unable to create socketn", argv[0]);
            exit(1);
}
            /* Bind the server's address to the socket. */
if (bind(s, &myaddr_in, sizeof(struct sockaddr_in)) == -1) {
            perror(argv[0]);
            printf("%s: unable to bind address\n", argv[0]);
            exit(1);
}
setpgrp();          /* set parent process */
switch (fork()) {   /* Get new process for parent process */
case -1:            /* Unable to fork, for some reason. */
            perror(argv[0]);
            printf("%s: unable to fork daemon\n", argv[0]);
            exit(1);
case 0:             /* The child process (daemon) comes here. */
                    /* Close stdin, stdout, and stderr so that they will
                     * not be kept open.  From now on, the daemon will
                     * not report any error messages.  This daemon
                     * will loop forever, waiting for requests and
                     * responding to them.
                     */
            close(stdin);
            close(stdout);
            close(stderr);
```

```
        /* This will open the /etc/hosts file and keep
         * it open.  This will make accesses to it faster.
         */
    sethostent(1);
    for(;;) {
        addrlen = sizeof(struct sockaddr_in);
                /* Recvfrom call will block until
                 * new the remote client connect
                 * arrives,return the address
                 * of client on clientaddr_in pointer
                 * and end of buffer on cc integer.
                 */
        cc = recvfrom(s, buffer, BUFFERSIZE - 1, 0,
                                &clientaddr_in, &addrlen);
        if ( cc == -1) exit(1);
        /* Search  the remote name from /etc/hosts */
        client_name = gethostbyaddr((char *)&clientaddr_in.sin_addr,
                                sizeof(struct in_addr),
                                clientaddr_in.sin_family);
        if( client_name == NULL) {
                hostname = inet_ntoa(clientaddr_in.sin_addr);
        } else {
                hostname = client_name->h_name;
        }
                /* Make null terminated in buffer */
        buffer[cc]='\0';
        sleep(1);
        time(&timevar);
printf(" Request from %s message<%s> at %s",
                        hostname, buffer, ctime(&timevar));
        sendto (s, buffer, BUFFERSIZE,
                        0, &clientaddr_in, addrlen);
    }
```

```
        default:        /* Parent process comes here. */
                exit(0);
        }
}
```

4.4 แสดงขั้นตอนการทำงาน ของ UDP Client Program บน Unix



รูปที่4.4 Flowchart แสดงขั้นตอนการทำงานของ UDP Client Programบน Unix

โปรแกรม UDP Client บน Unix

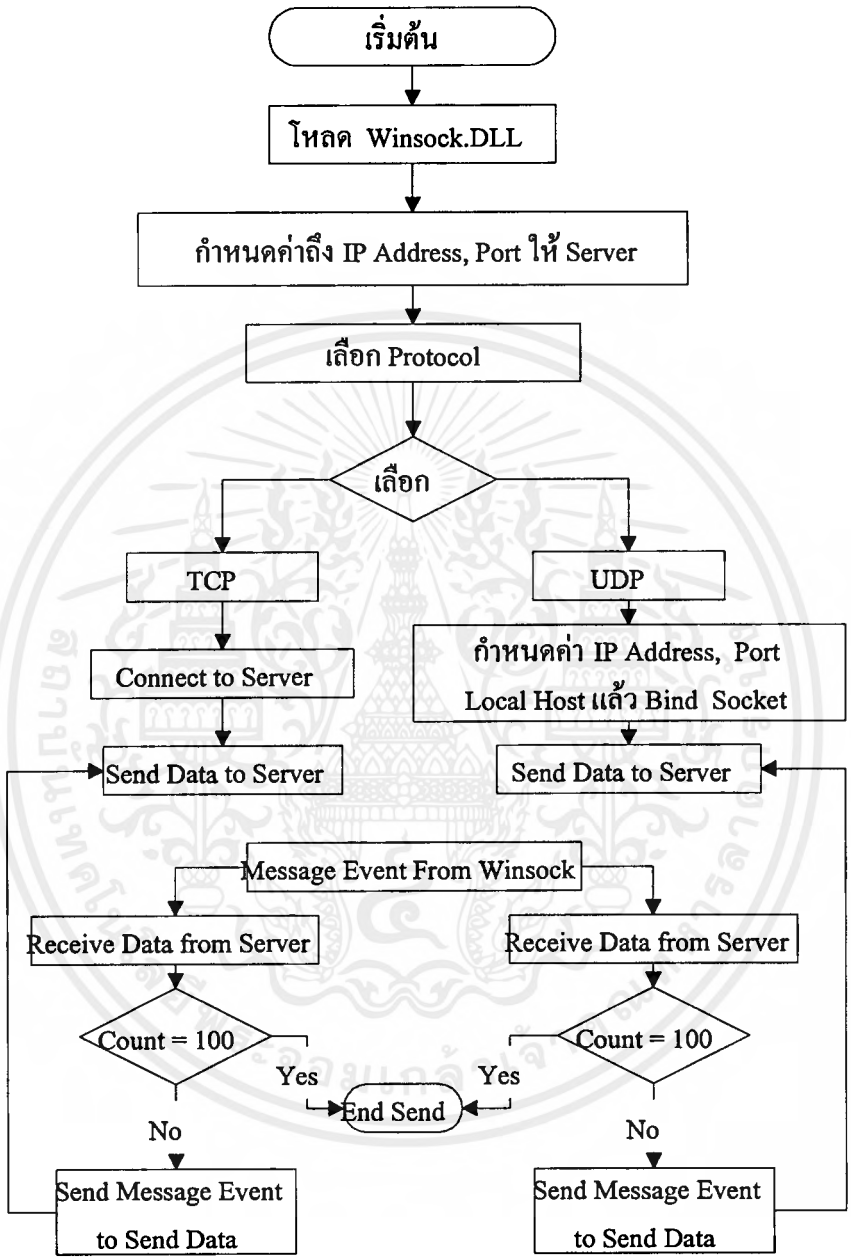
```
/* Datagram socket for client implement */
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <stdio.h>
#include <signal.h>
#include <netdb.h>

extern int errno;
int s;                                  /* socket descriptor */
struct hostent *server_name;            /* pointer to host information for name server host */
struct servent *service_port;           /* pointer to service information */
struct sockaddr_in myaddr_in;           /* for local socket address */
struct sockaddr_in servaddr_in;         /* for server socket address */
#define RETRIES         5               /* number of times to retry before give up */
#define BUFFERSIZE 1024                 /* maximum size of packet to be read into */
char buffer[BUFFERSIZE];                /* buffer for packet to be read into */
/*
 *                      H A N D L E R
 *
 *      This routine is the signal handler for the alarm signal.
 *      It simply re-installs itself as the handler and returns.
 */
handler()
{
        signal(SIGALRM, handler);
}

main(argc, argv)
int argc;
```

```
char *argv[];
{
        int i;
        int retry = RETRIES;            /* holds the retry count */
        char *inet_ntoa();
        if (argc != 3) {
                fprintf(stderr, "Usage: %s <nameserver> <message> \n", argv[0]);
                exit(1);
        }
                /* clear out address structures */
        memset ((char *)&myaddr_in, 0, sizeof(struct sockaddr_in));
        memset ((char *)&servaddr_in, 0, sizeof(struct sockaddr_in));
                /* Set up the server address. */
        servaddr_in.sin_family = AF_INET;
                /* Get the host information for the server's host name */
        server_name = gethostbyname (argv[1]);
        if ( server_name == NULL) {
                fprintf(stderr, "%s: %s not found in /etc/hosts\n",
                                argv[0], argv[1]);
                exit(1);
        }
        servaddr_in.sin_addr.s_addr = ((struct in_addr *)(server_name->h_addr))->s_addr;
                /* Search port number from /etc/services */
        service_port = getservbyname ("port_udp", "udp");
        if (service_port == NULL) {
                fprintf(stderr, "%s: port_udp not found in /etc/services\n",
                                argv[0]);
                exit(1);
        }
        servaddr_in.sin_port = service_port->s_port;
                /* Create the socket. */
        s = socket (AF_INET, SOCK_DGRAM, 0);
        if (s == -1) {
                perror(argv[0]);
```

```
                fprintf(stderr, "%s: unable to create socket\n", argv[0]);
                exit(1);
        }
                /* Bind socket to some local address so that the
                 * server can send the reply back.  A port number
                 * of zero will be used so that the system will
                 * assign any available port number.  An address
                 * of INADDR_ANY will be used so we do not have to
                 * look up the Internet address of the local host.
                 */
        myaddr_in.sin_family = AF_INET;
        myaddr_in.sin_port = 0;
        myaddr_in.sin_addr.s_addr = INADDR_ANY;
        if (bind(s, &myaddr_in, sizeof(struct sockaddr_in)) == -1) {
                perror(argv[0]);
                fprintf(stderr, "%s: unable to bind socket\n", argv[0]);
                exit(1);
        }
                /* Set up alarm signal handler. */
        signal(SIGALRM, handler);
                /* Send the request to the nameserver. */
again:  if (sendto (s, argv[2], strlen(argv[2]), 0, &servaddr_in,
                                sizeof(struct sockaddr_in)) == -1) {
                perror(argv[0]);
                fprintf(stderr, "%s: unable to send request\n", argv[0]);
                exit(1);
        }
                /* Set up a time-out so I don't hang in case the packet
                 * gets lost.  After all, UDP does not guarantee
                 * delivery.
        alarm(5);
                /* Wait for the reply to come in.  We assume that
                 * no messages will come from any other source,
                 * so that we do not need to do a recvfrom nor
```

```
                * check the responder's address.
        if (recv (s, &buffer, BUFFERSIZE, 0) == -1) {
                if (errno == EINTR) {
                        /* Alarm went off and aborted the receive.
                         * Need to retry the request if we have
                         * not already exceeded the retry limit.
                         */
                        if (--retry) {
                                goto again;
                        } else {
                                printf("Unable to get response from");
                                printf(" %s after %d attempts.\n",
                                                argv[1], RETRIES);
                                exit(1);
                        }
                } else {
                        perror(argv[0]);
                        fprintf(stderr, "%s: unable to receive response\n",
                                                        argv[0]);
                        exit(1);
                }
        }
        alarm(0);
        printf(" Echo message < %s > from server\n",buffer);
}
```

4.5 แสดงขั้นตอนการทำงาน ของ TCP & UDP Client Program บน PC



รูปที่4.5 Flowchart แสดงขั้นตอนการทำงานของ TCP & UDP Client Program บน PC

โปรแกรม TCP & UDP Client บน PC

```
Unit UClient;
interface
{ Identifies all units used by the program. }
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Menus, StdCtrls, ExtCtrls, Winsock;

{ The Constant reserved word defines an identifier whose value
 cannot change within the block containing the declaration }
Const MaxBuffer = 1024;  { Max Buffer }
     WM_WINSOCKTCP  = WM_USER + 1; { WinSock Event In TCP Type }
     WM_WINSOCKUDP  = WM_USER + 2; { WinSock Event In UDP Type }
     WM_SENDTCP     = WM_USER + 3; { Message Event Send In TCP Type }
     WM_SENDUDP     = WM_USER + 4; { Message Event Send In UDP Type }
     WM_SETCURSOR   = WM_USER + 5; { Message Event Set Cursor }

{A type declaration specifies an identifier that denotes a type.
 A variable's type defines the set of values it can have and the
 operations that can be performed on it. }
type
  TFClient = class(TForm)
    Memoshow: TMemo;
    MainMenu: TMainMenu;
    SelectProtocol: TMenuItem;
    RemoteSystem: TMenuItem;
    Exit1: TMenuItem;
    Panel1: TPanel;
    Editsend: TEdit;
    Label1: TLabel;
    MTCP: TMenuItem;
    TCP1: TMenuItem;
```

```
    UDP1: TMenuItem;
    MUDP: TMenuItem;
    Exit2: TMenuItem;
    CreateTCP: TMenuItem;
    ConnectTCP: TMenuItem;
    BindUDP: TMenuItem;
    DisconnectTCP: TMenuItem;
    CloseSocketUDP: TMenuItem;

    procedure TCP1Click(Sender: TObject);          { Select TCP Protocol }
    procedure UDP1Click(Sender: TObject);          { Select UDP Protocol }
    procedure Exit1Click(Sender: TObject);
    procedure FormActivate(Sender: TObject);
    procedure RemoteSystemClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure CreateTCPClick(Sender: TObject);
    procedure ConnectTCPClick(Sender: TObject);
    procedure DisconnectTCPClick(Sender: TObject);
    procedure BindUDPClick(Sender: TObject);
    procedure EditsendKeyPress(Sender: TObject; var Key: Char);
    procedure CloseSocketUDPClick(Sender: TObject);
  private
    { Private declarations }
    Sd_UDP,Sd_TCP: TSocket;                    { Connected socket descriptor }
    { SockAddrIn structure is specification for Internet address family
     Use to assign the components of an Internet Address }
    SAddr_Loc, SockAddr: PSockAddrIn;
    SAddr_to, SAddr_From: PSOCKAddrIn;
    { A pointer to the WSADATA data structure that is to
     receive details of the Windows Sockets implementation. }
    WSData: TWSAData;
    Version: Word;                  { Version of Winsock.DLL }

    { Data Buffer For Send & Receive }
```

```
  BufferTCP, BufferUDP: Array[0..MaxBuffer-1] Of Char;
  ResultTCP, ResultUDP, LenBufferUDP: Integer;  { Return Result from function }
  CountSendTCP, CountSendUDP: Integer;  { Counter For Send Data TCP & UDP }
  TextTCP, TextUDP: String;             { Save Text From EditSend.Text }
 HostTCP,HostUDP: PHOSTENT;                  { Keep  Address & Name Structure  }
 SHostTCP, SHostUDP: String;           { Keep Server Name }

  Procedure CheckErrorCode(Text: String);
  { Windows Message For Event TCP From Winsock }
  Procedure EventTCP(Var M: TMessage); Message  WM_WINSOCKTCP;
   { Windows Message For Event UDP From Winsock }
  Procedure EventUDP(Var M: TMessage); Message  WM_WINSOCKUDP;
   { Windows Message For Send Event TCP }
  procedure SendDataTCP(Var M: TMessage);  Message  WM_SENDTCP;
   { Windows Message For Send Event UDP }
  procedure SendDataUDP(Var M: TMessage); Message  WM_SENDUDP;
   { Windows Message For Set Cursor }
  Procedure SetCursorX(VAr M: TMessage); Message WM_SETCURSOR;
 public
  { Public declarations }
  { Variable Save Value IP ADDRESS and PORT }
  IP_Text, Port_Text : String;
 End;

var
 { Declaration Type to FClient }
 FClient: TFClient;
{ The implementation part of a unit defines the block of all public procedures }
implementation
 Uses URemote;
{$R *.DFM}

procedure TFClient.TCP1Click(Sender: TObject);  { Select TCP protocol }
begin
```

```
IF Not TCP1.Checked Then    { Indicates whether the check menu is selected }
Begin
 { TCP1 MenuItem is Selected }
 { Enabled MTCP MenuItem controls whether the control
   responds to mouse, keyboard, and timer events. }
 TCP1.Checked := True;
 MTCP.Enabled := True;
 {**********************}
 { UDP1 MenuItem is UnSelected }
 { Disabled MUDP MenuItem controls whether the control
   responds to mouse, keyboard, and timer events. }
 UDP1.Checked := False;
 MUDP.Enabled := False;
end;
end;

procedure TFClient.UDP1Click(Sender: TObject);    { Select UDP protocol }
begin
 IF Not UDP1.Checked Then    { Indicates whether the check menu is selected }
 begin
 { UDP1 MenuItem is Selected }
 { Enabled MUDP MenuItem controls whether the control
   responds to mouse, keyboard, and timer events. }
 UDP1.Checked := True;
 MUDP.Enabled := True;
 {**********************}
 { TCP1 MenuItem is UnSelected }
 { Disabled MTCP MenuItem controls whether the control
   responds to mouse, keyboard, and timer events. }
 TCP1.Checked := False;
 MTCP.Enabled := False;
end;
end;
```

```
procedure TFClient.Exit1Click(Sender: TObject);  { Close All Application }
begin
 { Closes a sockets. it releases the socket descriptor }
 IF DisConnectTCP.Enabled Then ResultTCP := CloseSocket(Sd_TCP);
 IF CloseSocketUDP.Enabled Then ResultUDP := CloseSocket(Sd_UDP);
{ Releases memory allocated for a dynamic variable. }
 Dispose(SockAddr);
 Dispose(SAddr_Loc);
 Dispose(SAddr_to);
 Dispose(SAddr_From);
 Close;   { Close All Application }
end;

procedure TFClient.FormActivate(Sender: TObject);  { Start Application }
begin
{ Create Pointer to Variable }
 New(SockAddr);
 New(SAddr_Loc);
 New(SAddr_to);
 New(SAddr_From);
 Version := MAkeword(1,1);
 ResultTCP := WSAStartUp(Version, WsData);   { Load Winsock.Dll }
 If ResultTCP = 0 Then                { WsaStartUp Successful }
 begin
  { Show Description Winsock.DLL }
  MemoShow.Lines.Add('Start Load..  Winsock.DLL OK');
  MemoShow.Lines.Add(WsData.szDescription);
  MemoShow.Lines.Add(WsData.szSystemStatus);
 end Else
 begin
  MemoShow.Lines.Add('Start.. Can not Load Winsock.DLL ');
 end;
end;
procedure TFClient.RemoteSystemClick(Sender: TObject);
```

```
begin
{ Show a form *FRemote* as a modal form. }
 FRemote.Showmodal;
end;

procedure TFClient.FormClose(Sender: TObject; var Action: TCloseAction);
begin
{  The Windows Sockets WSACleanup function
  terminates use of the Windows Sockets DLL.  }
 WSACleanup();
end;

procedure TFClient.CreateTCPClick(Sender: TObject);
 Var IPAddress: Array[0..20] of Char;
Begin
{ The Windows Sockets socket function creates a TCP socket .
  If the function succeeds, socket returns a descriptor referencing the new socket.
  If the function fails, a value of INVALID_SOCKET is returned }
 Sd_TCP := Socket(PF_INET, SOCK_STREAM, 0);
 If Sd_TCP = INVALID_SOCKET Then  CheckErrorCode('Create')
 Else
 begin
  MemoShow.Lines.Add('TCP Create Socket OK ');
  StrPCopy(IPAddress,IP_Text);
  { Set IP Address, Port for TSockAddrIn Structure To SockAddr Variable }
  SockAddr.sin_family := PF_INET;
  SockAddr.sin_addr.s_addr := Inet_Addr(IPAddress);  { IP Server }
  SockAddr.sin_port := htons(StrtoInt(Port_Text));         { Port Server }
 end;
end;

procedure TFClient.CheckErrorCode(Text: String);   { Check error Code }
Var ErrorCode: Integer;
    S: String;
```

```
Begin
 S := '';
 { Gets the error status for the last operation that failed. }
 ErrorCode := WsaGetLastError();
 Case ErrorCode OF
  WSAEADDRNOTAVAIL:  {10049}
    S := 'The specified address is not available from the local computer.';
  WSAECONNREFUSED:  {10061}
    S := 'The attempt to connect was forcefully rejected.';
  WSAENOTSOCK:      {10038}
    S := 'The descriptor is not a socket.';
 End;
 If S <> '' Then S := '   *' + S + '*';
 S := InttoStr(ErrorCode) + ':  ERROR ' + Text + S;
 MemoShow.Lines.Add(S);
end;

procedure TFClient.ConnectTCPClick(Sender: TObject);
begin
 { Local Connect to Server }
 SendMessage(Handle, WM_SETCURSOR, crHourGlass, 0);  { Set HourGlass Cursor }
 ResultTCP := Connect(Sd_TCP,SockAddr^,Sizeof(SockAddr^)); { Connect to Server }
 SendMessage(Handle, WM_SETCURSOR, CrDefault, 0);      { Set Default Cursor }
  If ResultTCP = SOCKET_ERROR Then CheckErrorCode('Connect')
 Else
 begin
  CreateTCP.Enabled := False;
  ConnectTCP.Enabled := False;
  DisconnectTCP.Enabled := True;
  EditSend.SetFocus;
  { The Windows Sockets WSAAsyncSelect function requests event
    notification for a socket. }
  ResultTCP := WsAAsyncSelect(Sd_TCP,Handle,WM_WINSOCKTCP,FD_CONNECT OR
FD_READ);
```

```
 If ResultTCP = SOCKET_ERROR Then CheckErrorCode('WsaAsynSelect ');
 end;
end;


{ Procedure Receive Message  Event WM_WINSOCKTCP   **TCP** }
Procedure TFClient.EventTCP(Var M: TMessage);
Begin
 Case (M.LParam) Of
 FD_CONNECT :    { Receive notification of completed connection }
  begin
   SendMessage(Handle, WM_SETCURSOR, crHourGlass,0); { Set HourGlass Cursor }
   { Retrieves the hostname By IP Address from Database File   (Hosts file) }
   HostTCP := Gethostbyaddr(@SockAddr.sin_addr.s_addr, 4, PF_INET);
   If HostTCP = Nil Then SHostTCP := Inet_ntoa(SockAddr.sin_addr)
   Else SHostTCP := HostTCP.h_name;
   SendMessage(Handle, WM_SETCURSOR, CrDefault, 0);   { Set Default Cursor }
   MemoShow.Lines.Add('CONNECT TO #' + SHostTCP + '# SERVER ...OK');
  end;
 FD_READ :       { Receive notification of readiness for reading }
  begin
   BufferTCP := '';
   { Receives data from a socket. }
   ResultTCP := Recv(Sd_TCP,BufferTCP,MaxBuffer,0);
   If ResultTCP = SOCKET_ERROR Then CheckErrorCode('Recv')
    Else MemoShow.Lines.Add('Return From Server : ' + BufferTCP);
   { Check Amount Send}
   If CountSendTCP < 100 Then
   begin
    Inc(CountSendTCP);
    { the window procedure is called immediately as a subroutine.
     If the specified window was created by a different thread,
     Windows switches to that thread and calls the
     appropriate window procedure.  }
```

```
    SendMessage(Handle,WM_SENDTCP,0,0); { Message to SendDataTCP procedure }
   end Else CountSendTCP := 0;
  end; {read}
 end; {Case }
end;

{ Procedure Receive Message  Event WM_WINSOCKUDP  **UDP** }
Procedure TFClient.EventUDP(Var M: TMessage);
Begin
 Case (M.LParam) Of
  FD_READ :  { Receive notification of readiness for reading }
   begin
    BufferUDP := '';
    LenBufferUDP := SizeOf(SAddr_From^);
    { Receives a Datagram and stores the source address. }
    ResultUDP := RecvFrom(SD_UDP,BufferUDP,MaxBuffer,0,SAddr_From^,LenBufferUDP);
    If ResultUDP = SOCKET_ERROR Then CheckErrorCode('RecvFrom')
     Else MemoShow.Lines.Add('Return From #' + SHostUDP + '# Server: ' + BufferUDP);
    If CountSendUDP < 100 Then
    begin
     Inc(CountSendUDP);
     SendMessage(Handle,WM_SENDUDP,0,0); { Message to SendDataUDP procedure }
    end Else CountSendUDP := 0;
   end; {read}
 End; {Case}
End;

procedure TFClient.DisconnectTCPClick(Sender: TObject);   { End Connect TCP }
begin
 { Closes a sockets. it releases the tcp socket descriptor }
 ResultTCP := CloseSocket(Sd_TCP);
 If ResultTCP = SOCKET_ERROR Then CheckErrorCode('CloseSocket')
 Else
 begin
```

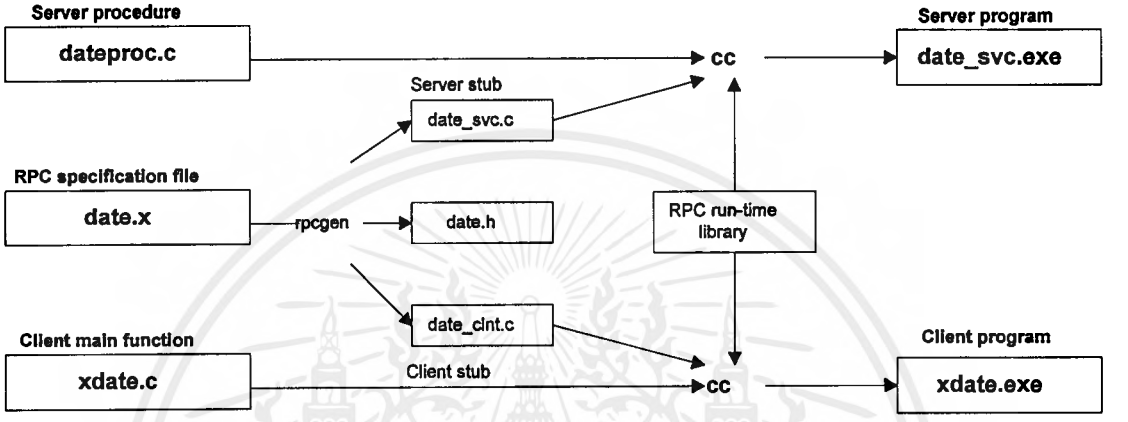
```
 MemoShow.Lines.Add('Disconnect OK ....');
 DisconnectTCP.Enabled := False;
 CreateTCP.Enabled := True;
 ConnectTCP.Enabled := True;
 end;
end;

{ Create,Bind Local Socket and Set Address Server }
procedure TFClient.BindUDPClick(Sender: TObject);
Var IPAddress: Array[0..20] of Char;
begin
{ The Windows Sockets socket function creates a UDP socket .
 If the function succeeds, socket returns a descriptor referencing the new socket.
 If the function fails, a value of INVALID_SOCKET is returned }
 Sd_UDP := Socket(PF_INET, SOCK_DGRAM, 0);
 If Sd_UDP = INVALID_SOCKET Then CheckErrorCode('Create')
 Else
 begin
  MemoShow.Lines.Add('UDP Create Socket OK ');
  BindUDP.Enabled := False;
  CloseSocketUDP.Enabled := True;
  EditSend.SetFocus;
  MemoShow.Lines.Add('Create Socket OK');
  { Set Address,Port For Local Address }
  SAddr_Loc.sin_family := PF_INET;
  SAddr_Loc.sin_addr.s_addr := htonl(INADDR_ANY);
  SAddr_Loc.sin_port := htons(0);   { Assign By System }
  { Associates a local address with a socket. }
  ResultUDP := Bind(Sd_UDP,SAddr_Loc^,Sizeof(SAddr_Loc^));
  If ResultUDP = SOCKET_ERROR Then CheckErrorCode('Bind')
  Else
  begin
   MemoShow.Lines.Add('UDP Bind OK');
   { Request that the Windows Sockets DLL should send a message
```

```
    to the window hWnd whenever it detects any of the network events
    specified by the lEvent parameter. The message that should be
    sent is specified by the wMsg parameter. }
   ResultUDP := WsAAsyncSelect(Sd_UDP,Handle,WM_WINSOCKUDP, FD_READ);
   If ResultUDP = SOCKET_ERROR Then CheckErrorCode('WsaAsyncSelect');
  end;
 end;
 { Set Addr_In For Remote Machine **Server** }
 StrPCopy(IPAddress,IP_Text);
 SAddr_To.sin_family := PF_INET;
 SAddr_To.sin_addr.s_addr := Inet_Addr(IPAddress);
 SAddr_To.sin_port := htons(StrtoInt(Port_Text));;
  SendMessage(Handle, WM_SETCURSOR, crHourGlass, 0); { Set HourGlass Cursor }
  HostUDP := Gethostbyaddr(@SAddr_To.sin_addr.s_addr, 4, PF_INET);
  { Retrieves the hostname By IP Address from Database File (Hosts file) }
  If HostUDP = Nil Then SHostUDP := Inet_ntoa(SockAddr.sin_addr)
  Else SHostUDP := HostTCP.h_name;
  SendMessage(Handle, WM_SETCURSOR, CrDefault, 0);   { Set Default Cursor }
end;

procedure TFClient.EditsendKeyPress(Sender: TObject; var Key: Char);
Begin
 If Key = Chr(13) Then     { Check KeyPressed Enter }
 Begin
  IF TCP1.Checked Then
  begin
   CountSendTCP := 0;
   TextTCP := EditSend.Text;
   SendMessage(Handle,WM_SENDTCP,0,0); { Message to SendDataTCP procedure }
  end
  ELse IF UDP1.Checked Then
  begin
   CountSendUDP := 0;
   TextUDP := EditSend.Text;
```

```
  SendMessage(Handle,WM_SENDUDP,0,0); { Message to SendDataUDP procedure }
 end;
end;
End;

procedure TFClient.SendDataTCP(Var M : Tmessage);
begin
 StrPCopy(BufferTCP, TextTCP + Inttostr(CountSendTCP));
 { Sends data on a connected socket.}
 ResultTCP := Send(SD_TCP, BufferTCP ,Length(BufferTCP) ,0);
 If ResultTCP = SOCKET_ERROR Then CheckErrorCode('Send')
End;
procedure TFClient.SendDataUDP(Var M : Tmessage);
begin
 StrPCopy(BufferUDP, TextUDP + Inttostr(CountSendUDP));
 { Sends data to a specific destination. }
 ResultUDP := Sendto(Sd_UDP, BufferUDP, Length(BufferUDP), 0, SAddr_to^, Sizeof(SAddr_to^));
 If ResultUDP = SOCKET_ERROR Then CheckErrorCode('Sendto')
end;

procedure TFClient.CloseSocketUDPClick(Sender: TObject);
begin
 { Closes a sockets. it releases the udp socket descriptor }
 ResultUDP := CloseSocket(Sd_UDP);
 If ResultUDP = SOCKET_ERROR Then CheckErrorCode('CloseSocket')
 Else
 begin
  MemoShow.Lines.Add('CloseSocket UDP OK ....');
  BindUDP.Enabled := True;
  CloseSocketUDP.Enabled := False;
 end;
end;
```

```
Procedure TFClient.SetCursorX(Var M: TMessage);   { Set Cursor }
begin
 EditSend.Cursor := M.WParam;
 Panel1.Cursor   := M.WParam;
 Memoshow.Cursor := M.WParam;
 FClient.Cursor  := M.WParam;
end;

End.  { End Program }
```

4.6 แสดงขั้นตอนการทำงาน ของ RPC program บน UNIX



รูปที่4.6 Flowchart แสดงขั้นตอนการComplie RPC program บน UNIX

โปรแกรม RPC server procedure บน UNIX

```
/*
 * dateproc.c - remote procedures; called by server stub.
 */

#include <rpc/rpc.h>
#include <time.h>

/*
 * Return the binary date and time.
 */

long *bin_date_1(void)
{
        static long timeval;        /* must be static */
        timeval = time((long *)0);
        return(&timeval);
}

/*
 * Convert a binary time and return a human readable sring.
 */

char **str_date_1(bintime)
{
        long *bintime;
        static char *ptr;           /* must be static */
        ptr = ctime(bintime);
        return (&ptr);
}
```

โปรแกรม RPC client main function บน UNIX

```
/*
 * xdate.c - client program for remote date service.
 */

#include <stdio.h>
#include <rpc/rpc.h>
#include "date.h"

main(int argc, char *argv[])
{
        CLIENT  *cl;                    /* RPC handle */
        char    *server;
        long    *lresult;  /* return value from bin_date_1() */
        char    **sresult;              /* return value from str_date_1() */

        if ( argc != 2 ) {
                fprintf(stderr, "usage: %s hostname\n", argv[0]);
                exit(1);
        }

        server = argv[1];

        /*
         * Create the client "handle."
         */

        if ( (cl=clnt_create(server, DATE_PROG, DATE_VERS, "udp")) == NULL ) {
                /*
                 * Could't establish connection with server.
                 */
                clnt_pcreateerror(server);
```

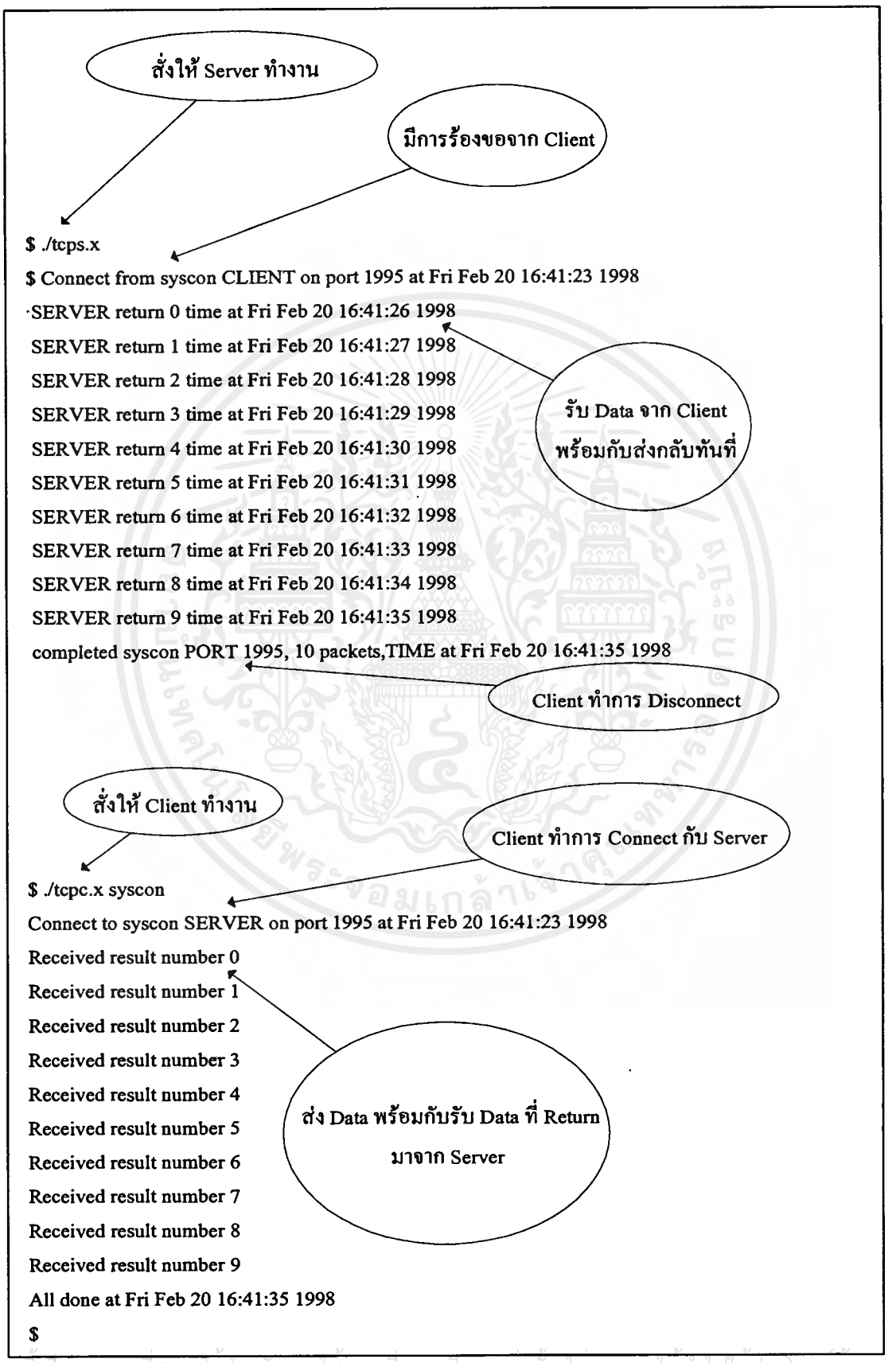
```
                exit(2);
        }

        /*
         * First call the remote procedure "bin_date".
         */

        if ( (lresult=bin_date_1(NULL,cl)) == NULL ) {
                clnt_perror(cl, server);
                exit(3);
        }
        printf("time on host %s = %ld\n", server, *lresult);

        /*
         * Now call the remote procedure "str_date".
         */

        if ( (sresult=str_date_1(lresult,cl)) == NULL ) {
                clnt_perror(cl, server);
                exit(4);
        }
        printf("time on host %s = %s", server, *sresult);

        clnt_destroy(cl);
        exit(0);
}
```

โปรแกรม RPC specification บน UNIX

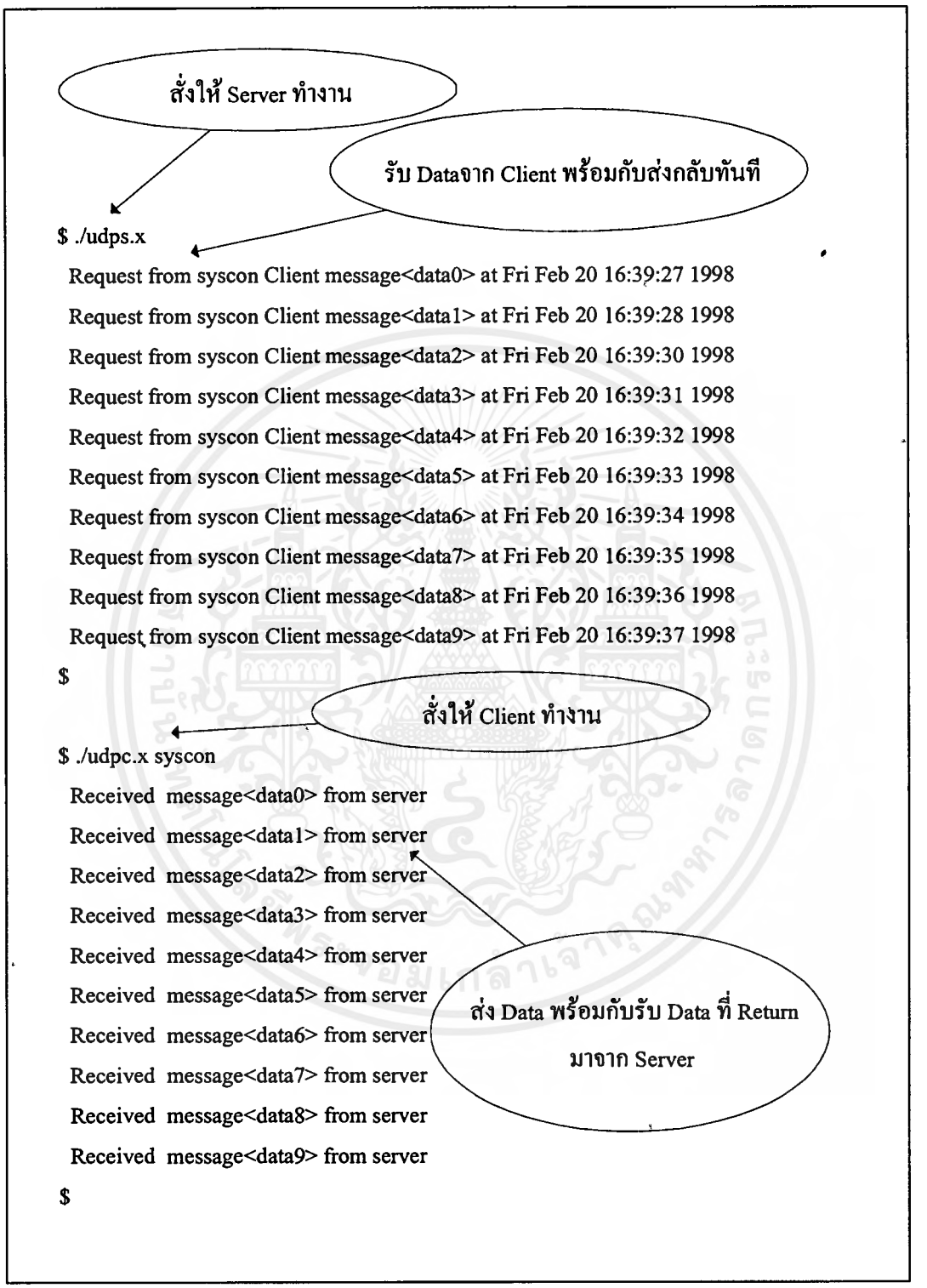
```
/*
 * date.x - Specification of remote date and time service.
 */

/*
 * Define 2 procedures:
 *      bin_date_1() returns the binary time and date (no auguments).
 *      str_date_1() takes a binary time and returns a human-readable string.
 */

program DATE_PROG {
        version DATE_VERS {
                long    BIN_DATE(void) = 1;             /* procedure number = 1 */
                string  STR_DATE(long) = 2;             /* procedure number = 2 */
        } = 1;                                                  /* version number = 1
*/
} = 0x31234567;                                                 /* program number =
0x31234567 */
```

4.7 ผลลัพธ์การทำงานของ Client/Server

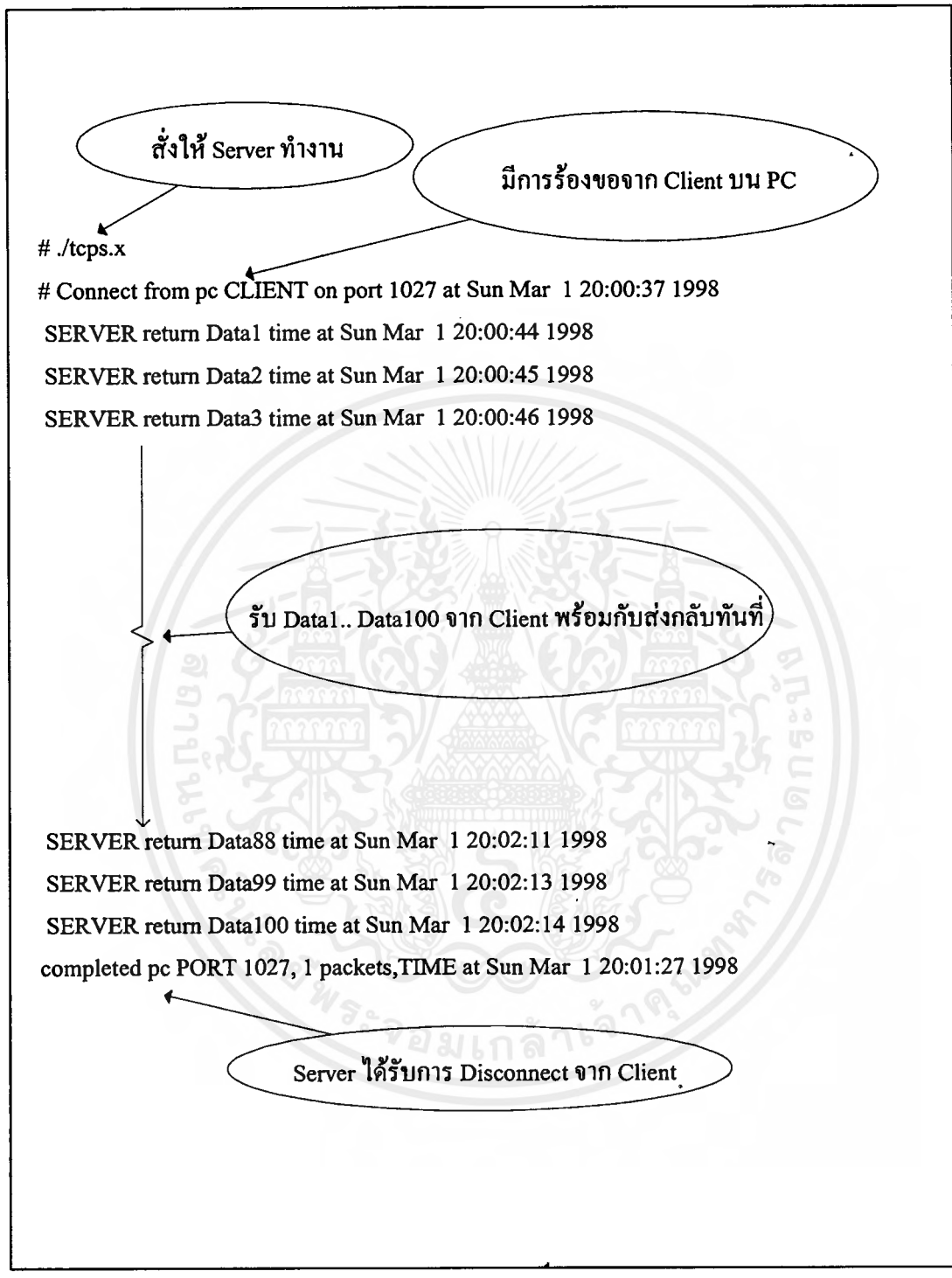
4.7.1 ผลลัพธ์การทำงานของ Client/Server แบบ TCP บน Unix



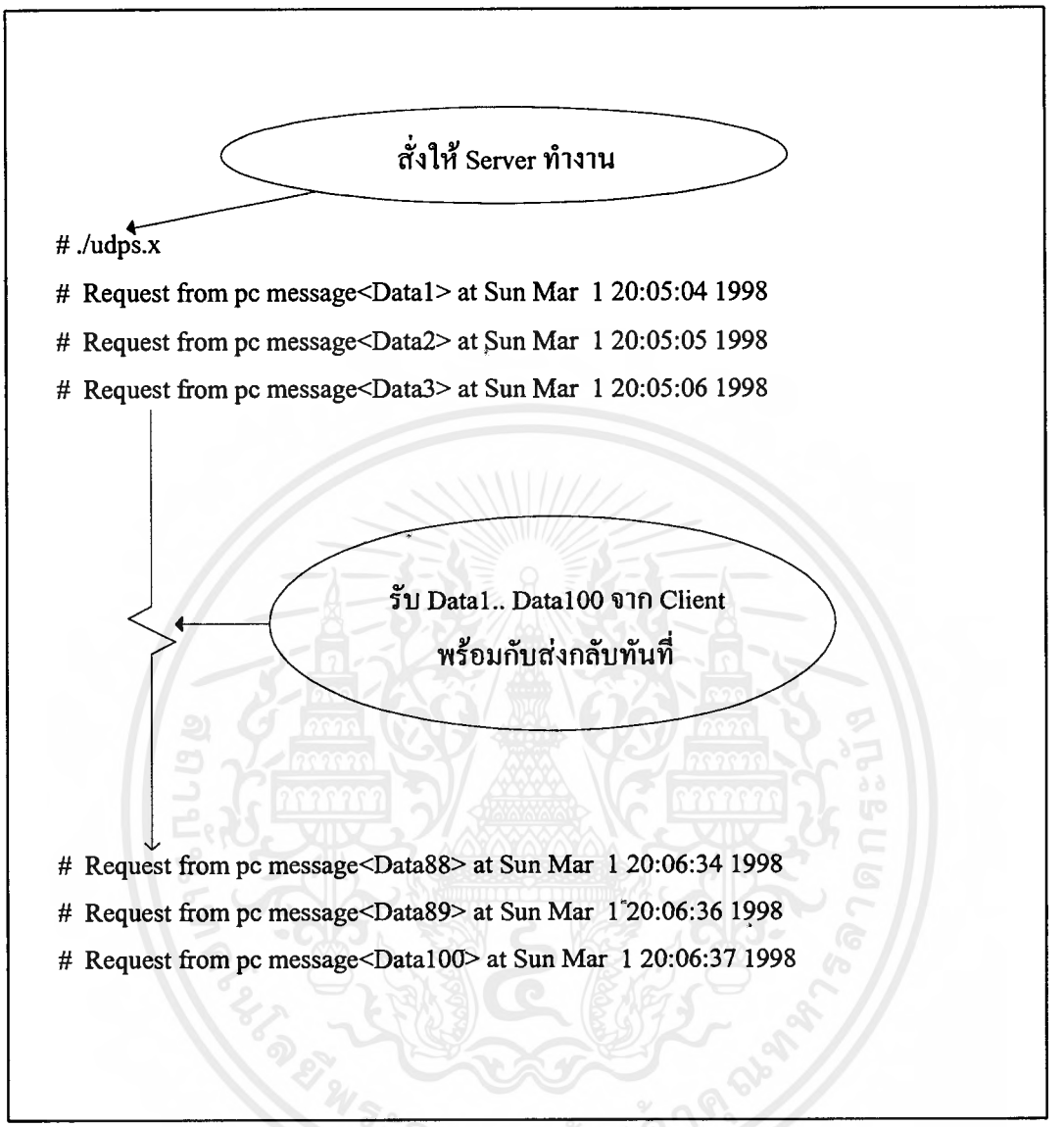
4.7.2 ผลลัพธ์การทำงานของ Client/Server แบบ UDP บน Unix



4.7.3 ผลลัพธ์การทำงานของ TCP Server บน Unix ที่เชื่อมต่อกับ TCP Client บน PC

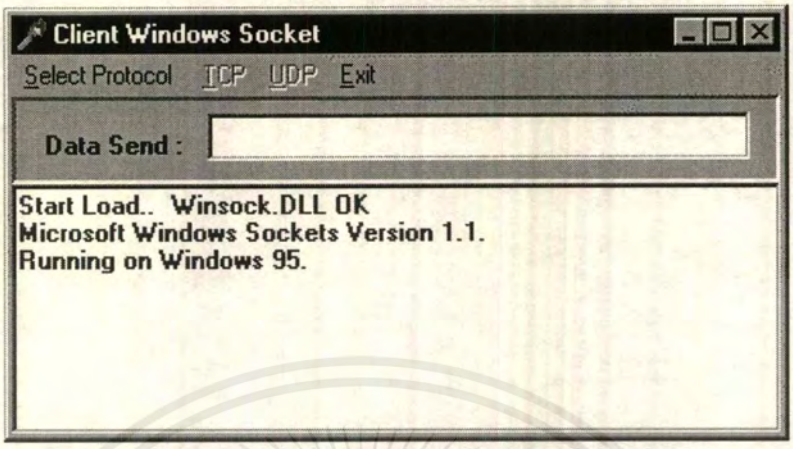


4.7.4 ผลลัพธ์การทำงานของ UDP Server บน Unix ที่เชื่อมต่อกับ UDP Client บน PC

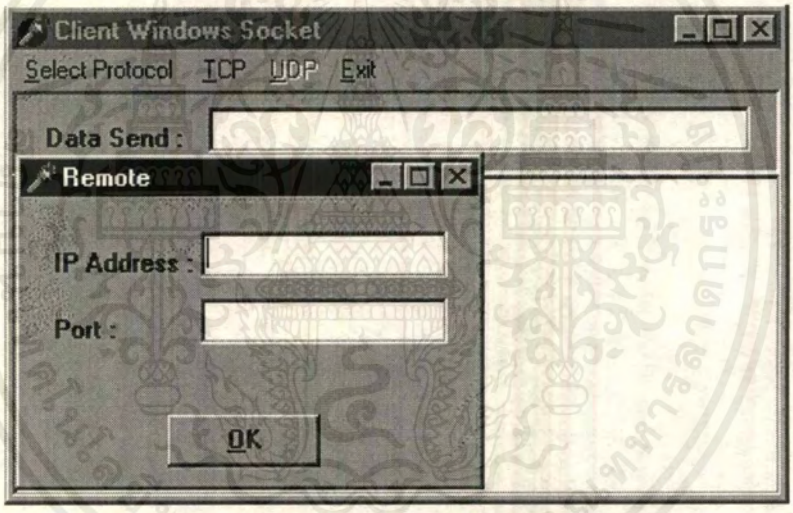


4.7.5 ผลลัพธ์การทำงานของ TCP & UDP Client บน PC ที่เชื่อมต่อกับ Server บน Unix

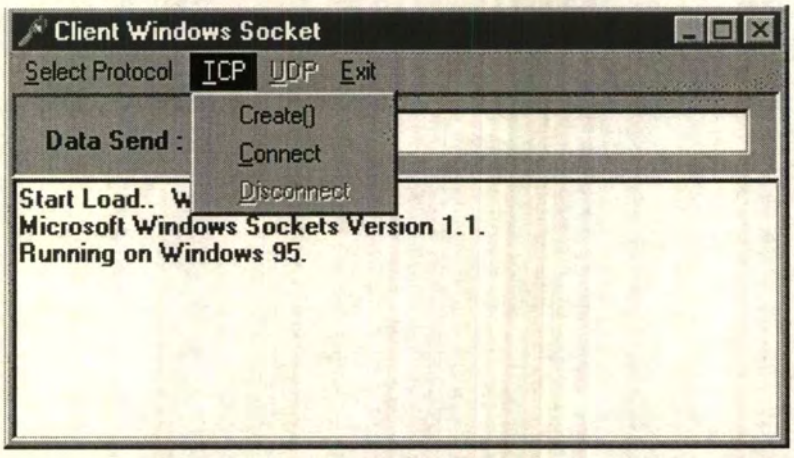
เริ่มต้นโปรแกรม ด้วยการโหลด Winsock.DLL



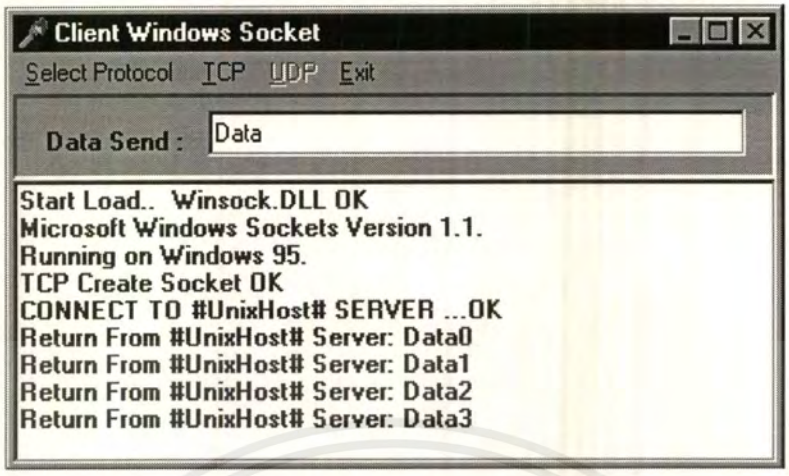
กำหนด Address และ Port ที่จะติดต่อ



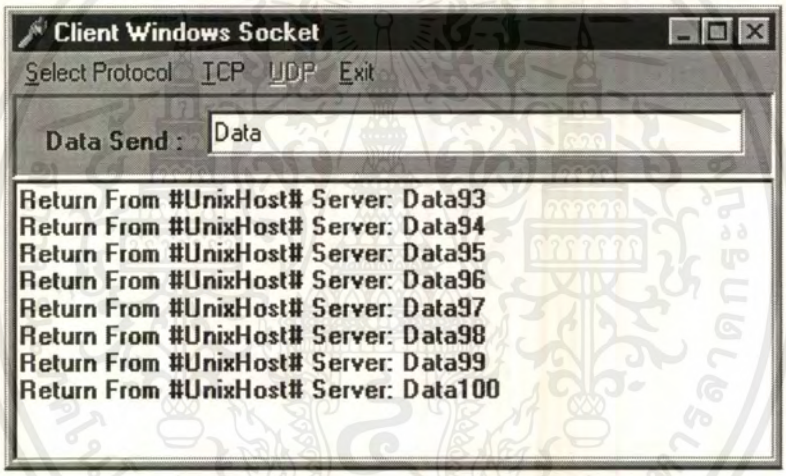
เลือกการติดต่อแบบ TCP และกำหนดค่าให้ Socket พร้อมที่จะติดต่อ



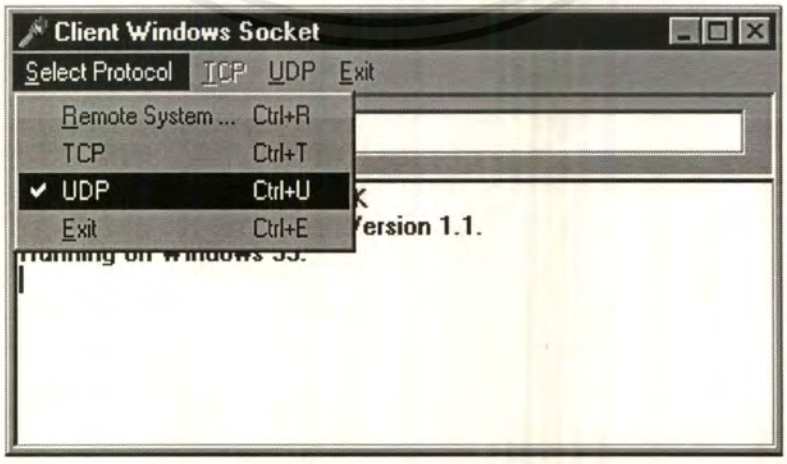
Connect To Server และ ส่ง Data พร้อมกับรับ Data ที่ Return From Server



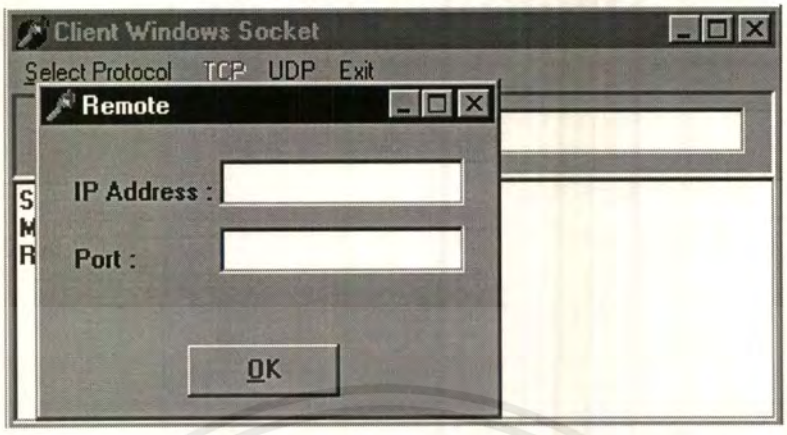
รับ Data กลับมาจนครบ



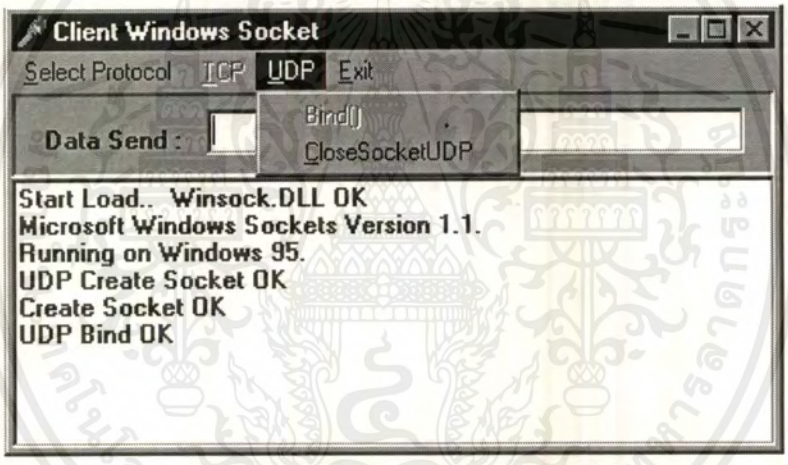
เลือกการติดต่อแบบ UDP



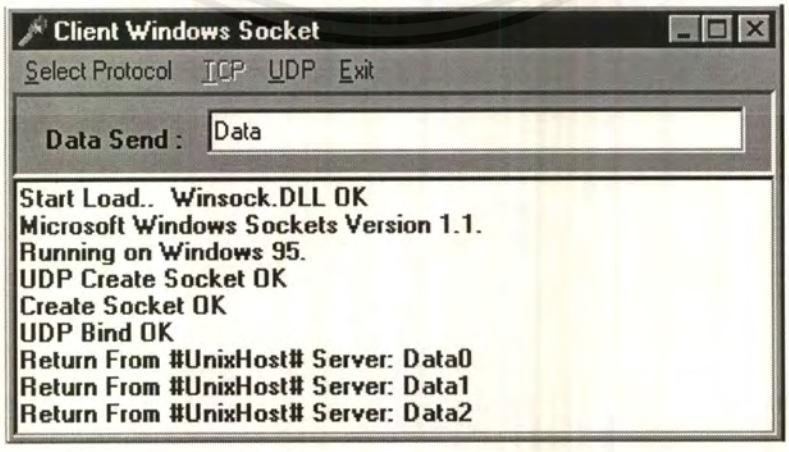
กำหนด Address และ Port ที่จะติดต่อ



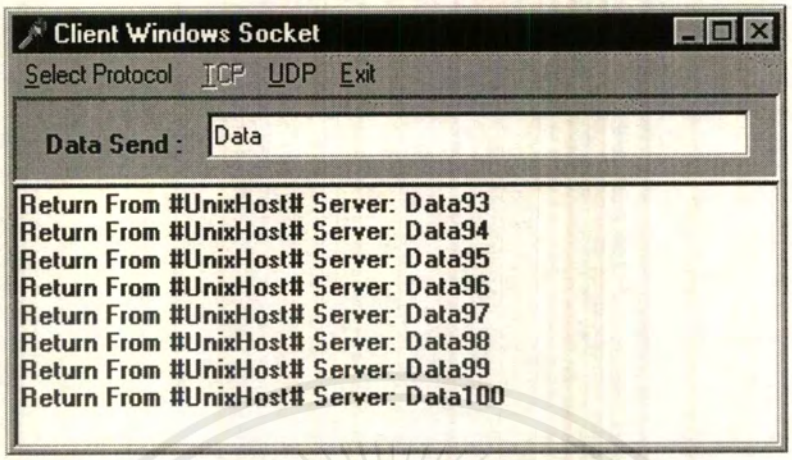
รวม Address และ Port เข้ากับ Socket เพื่อส่งไปยัง Server



ส่ง Data ออกไปยัง Server พร้อมกับรับ Data กลับ



รับ Data กลับมาจนครบ



4.7.6 ผลลัพธ์การทำงานของ RPC xdate program บน client ที่เป็น Unix

time on host hp = 890655253

time on host hp = Mon Mar 23 19:14:13 1998

บทที่5
สรุปและวิจารณ์

จากการศึกษา และค้นคว้าการติดต่อสื่อสารในรูปแบบของ Client/Server ทำให้ได้ทราบถึงการทำงานของ Client/Server ในลักษณะของ TCP/IP Protocol ซึ่งมีรูปแบบการรับและการส่งข้อมูลอยู่ สองลักษณะด้วยกัน ประกอบไปด้วย Connection-Oriented เป็นการรับส่งข้อมูลที่ถูกออกแบบมาให้มีความน่าเชื่อถือสูง มีการตรวจสอบการรับส่งข้อมูล, การตอบรับ และการควบคุมการทำงาน เพื่อตรวจสอบความถูกต้องของการรับส่งข้อมูล ส่วนแบบที่สองเรียกว่า Connectionless ซึ่งลักษณะการรับส่งข้อมูลจะแตกต่างจากแบบแรก กล่าวคือ จะไม่มีการหาเส้นทางการติดต่อระหว่าง Host และไม่มีการทำการจัดตั้งการเชื่อมต่อก่อน ที่จะทำการรับส่งข้อมูล ส่งผลให้การรับส่งข้อมูลไม่มีความน่าเชื่อถือ เช่นแบบ Connection-Oriented แต่ช่วยในการลด Overhead ของ Packet ได้ ซึ่งเหมาะสำหรับการรับส่งที่ไม่ไกลนัก เช่น ในระบบ LAN หรือใช้ในการกระจายข้อความ (Broadcast) และการสร้าง Client/Server ในอีกแบบหนึ่ง ซึ่งมี Tool ช่วยในการสร้าง เพื่อลดการเกิด Deadlock มีชื่อเรียกว่า RPC ซึ่งทำให้การจัดการเกี่ยวกับ Client/Server มีความง่ายขึ้น

นอกจากจะมีการเรียกใช้ Function ต่างๆของ Socket แล้ว Program ที่ทำงานบนสภาพแวดล้อม Windows ยังมีการเรียกใช้ Message ซึ่งก็คือ การแจ้งเหตุที่เกิดขึ้นกระทำโดยตัว Windows ส่งไปยัง Application อย่างเช่น เหตุการณ์การกด Mouse, เหตุการณ์การรับข้อมูลของ Sockets ล้วนเป็นตัวอย่างที่ Windows ทำการส่ง Message ไปยัง Application เพื่อแจ้งให้ Application ทราบว่ามีเหตุการณ์อะไรเกิดขึ้น ส่งผลให้ Application ทำงานตามเหตุการณ์ที่เกิดขึ้นในลักษณะ Thread ทำให้ Application ไม่จำเป็นต้องทำการวนลูปรอรับเหตุการณ์นั้น ซึ่งรายละเอียดต่างๆ ของ Message ได้แสดงไว้ในภาคผนวก ข

เอกสารอ้างอิง

ก. เอกสารอ้างอิงที่เป็นหนังสือภาษาอังกฤษ

1. Pat Bonner, "Network Programming With Windows Sockets", Prentice-Hall PTR A Simon & Schuster Company 493 p., 1996.
2. Douglass E. Comer & David Stevens, "Internetworking With TCP/IP Volume III", Prentice-Hall PTR ,
3. Jonathan Matcho, Scott Strool, Brian Salmanowitz "Special Edition Using Delphi2" , Que Book
4. HP Computer System Training Course, Programming with UNIX BSD Sockets, Student Workbook printed in USA 10/23/95

ข. เอกสารอ้างอิงที่เป็น World Wide Web (WWW)

1. http://www.stardust.com
2. http://sunsite.unc.edu/WinSock
3. http://www.delphi2.com
4. http://www.intel.com/IAL
5. http://www.stardust.com
6. http://www.borland.com
7. http://www.borland.com
8. http://www.borland.com

ค. เอกสารอ้างอิงที่เป็น File Transfer Protocol (FTP)

1. ftp://ftp.microsoft.com/bussys/winsock/spec11
2. ftp://ftp.stardust.com/pub/winsock/version1/docs/spec
3. ftp://sunsite.unc/edu/pub/micro/pc-stuff/ms-windows/winsock/winsock-1.1

ภาคผนวก ก

- **WinSocks Functions**

- คำสั่งที่ใช้ในการจัดตั้งการเชื่อมต่อแบบ Stream Socket

Socket () = สร้าง Communication Endpoint

Bind () = รวม Address เข้ากับ Socket กระทำบนฝั่ง Server

- คำสั่งด้าน Server

Listen() = Sets Up ลำดับถึงการรับการร้องขอเข้ามา เพื่อรอการเชื่อมต่อจาก Clients

Accept() = รับการร้องขอการเชื่อมต่อ ถ้ามันเป็นลำดับแรกมันก็จะรับเดี๋ยวนั้นเลย

- คำสั่งด้าน Client

Connect() = ร้องขอการเชื่อมต่อกับ Remote Address

- คำสั่งที่ใช้แลกเปลี่ยนข้อมูลบนการเชื่อมต่อแบบ Stream Socket

Send() = การส่งข้อมูล สังเกตุว่า TCP จะจัดการเชื่อมต่อข้อมูลแบบ Stream Buffer ของข้อมูลจะถูกกำหนดขึ้น เมื่อมีการส่งข้อมูลซึ่งเป็นลักษณะพื้นฐานใน TCP Algorithms

Recv() = การรับข้อมูล สังเกตุว่ามันจะนำข้อมูลทั้งหมดขึ้นมาตามขนาดที่รับเข้ามา

- คำสั่งที่ใช้ในการปิดการเชื่อมต่อแบบ Stream Socket

CloseSocket() = ปิดการเชื่อมต่อของ Socket และปล่อยทรัพยากร Socket

Shutdown() = ให้หยุดการเคลื่อนย้ายของข้อมูลในทิศทางเดียวหรือทั้งสองทิศทาง เช่น Socket สามารถ Shutdown การส่ง หรือ Shutdown การรับ หรือ Shutdown ทั้งการส่งและการรับ แต่ อย่างไรก็ตามเราสามารถให้ความแน่นและปลอดภัยต่อ Data ทั้งหมดก่อนโดยการส่งและรับข้อความก่อนที่จะทำการปิด Socket

- คำสั่งที่ใช้สำหรับ Datagram Sockets

Socket() = จัดตั้ง Communication Endpoint

Bind() = รวม Socket เข้ากับ Address

Sendto() = ส่งข้อมูลไปตาม Address

Recvfrom() = รับข้อมูล

CloseSocket() = ปลดปล่อยทรัพยากร Socket

Connect() = เป็นคำสั่งที่เพิ่มเข้ามาใช้ในการรวม Remote Address กับ Datagram Socket ซึ่งสามารถใช้คำสั่ง Send() และ Recv() ได้

- คำสั่งที่ใช้ในการรับหรือตั้งค่าให้กับ Socket

GetpeerName() = ค่าที่ได้กลับมาเป็น Address ของ Remote Socket ได้มาเพื่อการเชื่อมต่อ Stream Sockets หรือการรวม Socket (และมีบางกรณีใช้กับ Datagram Socket ได้)

Getsockopt() = ค่าที่ได้กลับมาเป็นค่าเฉพาะที่เป็นส่วนที่เพิ่มเข้ามาของ Socket เช่น Socket Type, Socket Linger Option

Setsockopt() = เป็นการตั้งค่าเฉพาะที่เป็นส่วนที่เพิ่มเข้ามาของ Socket

IoctlSocket() = อนุญาติให้คุณตั้งค่า Socket Blocking Mode

Select() = นำเอาส่วนที่เป็น Information เมื่อ Socket พร้อมและเชื่อมต่อเรียบร้อยแล้วหรือเพื่อการอ่านหรือเขียน

- คำสั่งที่ใช้เปลี่ยน Byte Order

ntohl() = เปลี่ยนค่า 32 bit จาก Network เป็น Host

ntohs() = เปลี่ยนค่า 16 bit จาก Network เป็น Host

htonl() = เปลี่ยนค่า 32 bit จาก Host เป็น Network

htons() = เปลี่ยนค่า 16 bit จาก Host เป็น Network

- คำสั่งที่ใช้ในการปรับเปลี่ยน IP Address

inet_addr() = เปลี่ยนจาก IP Address ในรูปแบบตัวเลขที่มีจุดไปเป็น 4 Byte Hex ในรูปแบบ Network Byte

inet_ntoa() = เปลี่ยนจาก IP Address ในรูปแบบ 4 Byte Hex ไปเป็นตัวเลขที่มีจุด

- คำสั่งที่นำไปใช้กับ Database Information

GetHostbyaddr() = การรับ Host Name, Address, และ Aliases ที่รวมกับ Host Address

GetHostbyName() = การรับ Address Name , Name และ Aliases รวมกับ Protocol Name หรือ Alias

GetHostName() = การรับ ชื่อจาก Local Host

GetprotobyName() = การรับ Protocol Number,Name,Number และ Aliases ที่รวมกับ Protocol Number

GetservbyName() = การรับ Service Port Number,Name, และ Aliases ที่รวมกับ Service Name หรือ Alias

GetservbyPort() = การรับ Service Name, Port Number, และ Aliases ที่รวมกับ Port Number

ภาคผนวก ข

- **Message บน Windows**

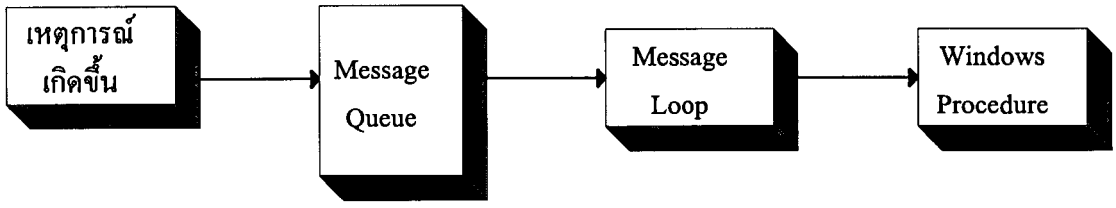
Message คือ การแจ้งเหตุการณ์ที่เกิดขึ้นจากการที่ Windows ส่งไปยัง Application เช่น เหตุการณ์การกด Mouse, เหตุการณ์การรับข้อมูลของ Sockets ล้วนเป็นตัวอย่างที่ Windows ทำการส่ง Message ไปยัง Application เพื่อแจ้งให้ Application ทราบว่ามีเหตุการณ์อะไรเกิดขึ้น ส่งผลให้ Application ทำงานตามเหตุการณ์ที่เกิดขึ้นในลักษณะ Thread ทำให้ Application ไม่จำเป็นต้องทำการวนลูปรอรับเหตุการณ์นั้น ซึ่งมีโครงสร้างของ Record Type ที่ถูกกำหนดมาจาก Windows ดังนี้

```
Type
 Tmsg = packed record
        hwnd:  HWND;          // เป็น handle ของ Windows
        message:  UINT;       // เป็นค่าคงที่ของ Message ที่กำหนดขึ้น
        wParam:  WPARAM;      // เป็น Parameter ของ Message
        lParam:  LPARAM;
        time:     Dword;      // เวลาในขณะที่ Message ถูกสร้างขึ้น
        pt:       TPoint;     // ตำแหน่งของ Mouse เมื่อ Message ถูกสร้างขึ้น
 end;
```

ขั้นตอนการทำงานของ Message มีอยู่ 5 ขั้นตอน คือ

1. เมื่อมีเหตุการณ์เกิดขึ้นในระบบ
2. Windows ทำการเคลื่อนย้ายเหตุการณ์ไปยัง Message Queue
3. Application ก็จะทำการรับ Message จาก Queue
4. Application ก็จะส่ง Message ไปยัง Procedure ของ Application
5. Windows Procedure ก็จะ Action

แสดงลักษณะการทำงานของ Message



ระบบ Delphi Message

ใน Delphi จะมีการสร้าง Record Type ที่เก็บ Information ของ Message เป็นของตัวเอง โดยประกอบไปด้วย Record Tmsg ของ Windows รวมอยู่ข้างในด้วย Record type ของ Delphi มีชื่อว่า Tmessage record type มีลักษณะโครงสร้างดังนี้

```
type
  Tmessage = record
    Msg: Cardinal;
    Case Integer of
      0: (
        WParam: Longint;
        Lparam: Longint;
        Result: Longint );
      1: (
        WParamLo: Word;
        WParamHi: Word;
        LParamLo: Word;
        LParamHi: Word;
        ResultLo: Word;
        ResultHi: Word );
  end;
```

ลักษณะการใช้ หรือการสร้าง Procedure สำหรับ รับเหตุการณ์ต่างๆ กระทำได้ด้วยการกำหนดค่าคงที่สำหรับเหตุการณ์นั้นด้วย Wm_User + X โดยที่ X เป็นค่าตัวเลขที่ต้องการระบุ และทำการสร้าง Procedure โดยมีรูปแบบและลักษณะการสร้างดังนี้

```
Const
        Sx_MyMessage = Wm_User + 1;

Tform1 = Class(Tform)
      .
      .
      .
private
  Procedure PMyMessage(var Msg: Tmessage); message  Sx_MyMessage;
end;

Procedure TForm1.PMyMessage(var Msg: Tmessage);
begin
   Action ;
end;
```