การประเมินสมรรถนะและเปรียบเทียบของโพรโตคอล TCP รุ่นต่างๆ สำหรับโครงข่าย IP ของดาวเทียม

นายสนามชัย มาสุธน

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมไฟฟ้า ภาควิชาวิศวกรรมไฟฟ้า
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2548

ISBN 974-17-3728-9

PERFORMANCE EVALUATION AND COMPARISON OF VARIOUS TCP VERSIONS FOR SATELLITE IP NETWORK

Mr. Sanamchai Masuthon

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering Program in Electrical Engineering
Department of Electrical Engineering
Faculty of Engineering
Chulalongkorn University
Academic Year 2005

ISBN 974-17-3728-9

หัวข้อวิทยานิพนธ์	การประเมินสมรรถนะและเปรียบเทียบของโพรโตคอล TCP รุ่นต่างๆ สำหรับโครงข่าย IP ของดาวเทียม
โดย	นายสนามชัย มาสุธน
สาขาวิชา	วิศวกรรมไฟฟ้า
อาจารย์ที่ปรึกษา	รองศาสตราจารย์ ดร.ประสิทธิ์ ทีฆพุฒิ

คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย อนุมัติให้นับวิทยานิพนธ์ฉบับนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญามหาบัณฑิต

.....คณบดีคณะวิศวกรรมศาสตร์

(ศาสตราจารย์ ดร.ดิเรก ลาวัณย์ศิริ)

คณะกรรมการสอบวิทยานิพนธ์

.....ประธานกรรมการ

(ศาสตราจารย์ ดร.ประสิทธิ์ ประพิณมงคลการ)

.....อาจารย์ที่ปรึกษา

(รองศาสตราจารย์ ดร.ประสิทธิ์ ทีฆพุฒิ)

.....กรรมการ

(ผู้ช่วยศาสตราจารย์ ดร.สุภาวดี อร่ามวิทย์)

.....กรรมการ

(ดร.นงลักษณ์ พินัยนิติศาสตร์)

สนามชัย มาสุธน : การประเมินสมรรถนะและเปรียบเทียบของโพรโตคอล TCP รุ่นต่างๆ สำหรับโครงข่าย IP ของดาวเทียม (PERFORMANCE EVALUATION AND COMPARISON OF VARIOUS TCP VERSIONS FOR SATELLITE IP NETWORK) อ. ที่ปรึกษา : รศ.ดร.ประสิทธิ์ ทีฆพุฒิ, 80 หน้า. ISBN 974-17-3728-9.

โพรโตคอล TCP/IP เป็นโพรโตคอลที่จัดอยู่บน transport layer และเป็นโพรโตคอลที่นิยมใช้บนอินเตอร์เน็ท ซึ่งส่วนมากโพโตรคอล TCP ได้พัฒนาให้ใช้งานบนโครงข่ายภาคพื้นดินโดยเฉพาะ การพัฒนาโพรโตคอล TCP ให้สามารถใช้งานบนโครงข่ายไร้สายหรือโครงข่ายดาวเทียมกำลังทำการวิจัยกันอย่างกว้างขวาง ในวิทยานิพนธ์เล่มฉบับนี้จะกล่าวถึงปัญหาทางด้านสมรรถนะของโพรโตคอล TCP ที่ถูกนำไปใช้บนโครงข่ายดาวเทียมและแสดงวิธีการแก้ไขแบบต่างๆ และได้หาแบบแผนควบคุมความคับคั่งแบบต่างๆเพื่อให้เหมาะกับการใช้งานในสภาพแวดล้อมบนโครงข่ายดาวเทียม โดยได้เลือกแบบแผนควบคุมความคับคั่ง TCP-Newreno TCP-Peach และ TCP-Vegas มาทำการทดลองหาสมรรถนะทั้งบนโครงข่ายภาคพื้นดินและโครงข่ายดาวเทียม โดยในการทดลองจะใช้เครื่องมือที่เรียกว่า NS-2 Simulator เป็นเครื่องมือในการทำการวิจัย

การออกแบบสภาพแวดล้อมในการทดลองได้ออกแบบให้ครอบคลุมสภาพแวดล้อมจากพื้นฐานไปจนถึงสภาพแวดล้อมที่เหมือนจริง ทั้งโครงข่ายภาคพื้นดินและโครงข่ายดาวเทียม โดยจะใช้พารามิเตอร์ goodput fairness และ friendliness เป็นตัวชี้วัดหาสมรรถนะ

ภาควิชา วิศวกรรมไฟฟ้า	ลายมือชื่อนิสิต สนามชัย มาสุธน
สาขาวิชา วิศวกรรมไฟฟ้า	ลายมือชื่ออาจารย์ที่ปรึกษา
ปีการศึกษา 2548	ลายมือชื่ออาจารย์ที่ปรึกษาร่วม

4670534021 : MAJOR ELECTRICAL ENGINEERING

KEY WORD: CONGESTION CONTROL/THROUGHPUT/TCP/IP/SATELLITE

SANAMCHAI MASUTHON : PERFORMANCE EVALUATION AND COMPARISON OF VARIOUS TCP VERSIONS FOR SATELLITE IP NETWORK . THESIS ADVISOR : ASSOC. PROF.DR.PRASIT TEEKAPUT, Ph.D. 80 pp. ISBN 974-17-3728-9.

TCP/IP is the protocol used as the transport layer and most used in the internet. Most TCP has been designed and improved based on the terrestrial networks. The improvement of TCP performance in wireless communications has been an active research area. In this thesis, some performance problems of TCP in satellite environment will be discussed and shown how to impair. We determine the TCP congestion control scheme that is suitable for wireless and wire-wireless hybrid networks that is the satellite IP network. In this thesis, to evaluate and compare three TCP congestion control schemes, which are TCP-Newreno, TCP-Peach and TCP-Vegas by using NS-2 Simulator. Simulation scenarios are carefully designed for both terrestrial network and satellite network. The study analyzed three performance measures which are goodput, fairness and friendliness of the TCP.

Department	Electrical Engineering	Student's signature	
Field of study	Electrical Engineering	Advisor's signature	P. Teekaput
Academic year	2005	Co-advisor's signature	

กิตติกรรมประกาศ

ผู้จัดทำวิทยานิพนธ์ขอขอบพระคุณ รศ.ดร. ประสิทธิ์ ทีฆพุฒิอาจารย์ที่ปรึกษาวิทยานิพนธ์ ที่ได้กรุณาให้คำปรึกษาข้อแนะนำ วิธีการแก้ปัญหาตลอดจนแนวทางในการทำวิทยานิพนธ์

ขอขอบคุณห้องปฏิบัติการวิจัยโทรคมนาคม ซึ่งเป็นสถานที่ทำวิจัย รวมถึงเพื่อน รุ่นพี่และรุ่นน้องทุกท่านที่มีส่วนช่วยเหลือในการให้ข้อคิดเห็น และคำแนะนำต่างๆ ในการเรียนรวมถึงการทำวิทยานิพนธ์

ขอขอบคุณโครงการวิจัยร่วมภาครัฐและเอกชนที่ได้ให้ความช่วยเหลือในด้านต่างๆ เช่น อุปกรณ์เพื่อทำการวิจัยตลอดจนทุนอุดหนุนการวิจัย ทำให้การดำเนินการวิจัยมีประสิทธิผลดียิ่งขึ้น

สุดท้ายนี้ ผู้วิจัยขอกราบขอบพระคุณบิดามารดา ที่ได้มอบความรัก ความอบอุ่น และกำลังใจตลอดมา ตลอดจนพี่สาวน้องสาวและน้องสาวของข้าพเจ้าที่ให้การสนับสนุนเสมอ

สารบัญ

หน้า

สารบัญตาราง

สารบัญรูปภาพ

หน้า

หน้า

หน้า

หน้า

บทที่ 1

บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

โพรโตคอล TCP/IP เป็นโพรโตคอลที่ใช้กันแพร่หลายมากที่สุดบนโครงข่ายอินเตอร์เนตในยุคปัจจุบัน เริ่มพัฒนาขึ้นจากโครงการที่มีชื่อว่า ARPAnet (Advance Project Agency Network)

มาตราฐานโพรโตคอล TCP (Transmission Control Protocol) ถูกกำหนดขึ้นโดย IETF (Internet Engineering Task Force) รายละเอียดอยู่ใน RFC (Request For Comment) เลขที่ 793 เนื่องจากในยุคต่อไปความต้องการการใช้งานอินเตอร์เนตความเร็วสูงในรูปแบบของการส่งข้อมูล ภาพ เสียง และวิดีโอ มีปริมาณเพิ่มมากขึ้น ทำให้เกิดความคับคั่งของช่องสัญญาณอย่างหลีกเลี่ยงไม่ได้ ความคับคั่งในช่องสัญญาณที่เกิดขึ้นนี้ส่งผลให้ความเร็วในการรับส่งข้อมูลลดลง ซึ่งมีสาเหตุหลักมาจากโพรโตคอล TCP ที่รับบทบาทเป็นตัวควบคุมการส่งข้อมูลจากต้นทางไปยังปลายทาง (end-to-end connection) และ TCP ยังมีหน้าที่รับผิดชอบในการประกันการส่งข้อมูลจากต้นทางไปให้ไปถึงยังปลายทางได้แน่นอน (Reliability) และถูกต้อง ดังนั้น TCP จึงเป็นโพรโตคอลสำคัญที่ควรนำมาศึกษาและแก้ไขให้สอดคล้องกับสภาพแวดล้อมของการนำไปใช้งาน เพราะในสภาพแวดล้อมของโครงข่ายที่แตกต่างกัน โพรโตคอล TCP จะให้อัตราเร็วในการรับส่งข้อมูลที่แตกต่างกัน

ในปัจจุบันการสื่อสารผ่านดาวเทียมมีการใช้งานอย่างกว้างขวาง และมีงานประยุกต์ที่หลากหลายมากขึ้น การใช้อินเตอร์เนตผ่านดาวเทียมก็เป็นอีกงานประยุกต์หนึ่งที่กำลังได้รับความนิยม โพรโตคอล TCP/IP ที่มีอัลกอริทึมที่เหมาะสมกับสภาพแวดล้อมบนโครงข่ายดาวเทียมจึงมีความสำคัญ และจำเป็นต้องพัฒนาขึ้นเฉพาะ เพราะโพรโตคอล TCP ที่ใช้โดยทั่วไปได้พัฒนาขึ้นเพื่อตอบสนองต่อการส่งผ่านข้อมูลบนโครงข่ายภาคพื้นดินเป็นหลัก หากนำโพรโตคอลบนโครงข่ายภาคพื้นดินมาใช้บนโครงข่ายดาวเทียมจะมีผลทำให้อัตราเร็วในการรับส่งข้อมูลมีค่าต่ำกว่าโพรโตคอลที่ออกแบบมาเพื่อโครงข่ายดาวเทียมโดยเฉพาะ

งานวิจัยที่ผ่านมาได้มีการศึกษาและปรับปรุงแบบแผนควบคุมความคับคั่งของโพรโตคอล TCP ไว้มากมาย ยกตัวอย่างเช่น TCP-Tahoe, TCP-Reno, TCP-New Reno, TCP-Peach, TCP-Peach+, TCP-Vegas, TCP-Westwood, TCP-Westwood+ และ TCP-Jersey แต่โพรโตคอล TCP ดังกล่าว ส่วนมาก

จะถูกพัฒนาขึ้นมาจากสภาพแวดล้อมบนโครงข่ายภาคพื้นดิน ซึ่งมีสภาพแวดล้อมแตกต่างจากโครงข่ายดาวเทียม เช่น ค่าเวลาในการรับส่งหรือตัวแปร *RTT* (Round Trip Time) ของโครงข่ายดาวเทียมจะมีค่ามากกว่าโครงข่ายภาคพื้นดินมาก และค่าอัตราการส่งข้อมูลผิดพลาดหรือตัวแปร *BER* (Bit Error Rate) ซึ่งโครงข่ายดาวเทียมมีค่ามากกว่าโครงข่ายภาคพื้นดินเช่นกัน ค่าตัวแปรเหล่านี้ทำให้เกิดปัญหาในการใช้โพรโตคอล TCP/IP ผ่านดาวเทียม และเป็นผลที่มีสาเหตุมาจากแบบแผนควบคุมความคับคั่งในโพรโตคอล TCP โดยตรง ดังนั้นการใช้โพรโตคอล TCP/IP ผ่านดาวเทียมจึงต้องมีแบบแผนควบคุมความคับคั่งที่เหมาะสมกับสภาพแวดล้อมของโครงข่ายดาวเทียมด้วย

1.2 วัตถุประสงค์ของงานวิจัย

งานวิจัยนี้จึงมีจุดประสงค์เพื่อประเมินหาสมรรถนะของโพรโตคอล TCP ในแบบแผนควบคุมความคับคั่ง (Congestion Control Scheme) รุ่นต่างๆ ในสภาพแวดล้อมที่เป็นโครงข่ายภาคพื้นดิน และในสภาพแวดล้อมที่เป็นโครงข่ายดาวเทียม โดยจะใช้พารามิเตอร์ goodput, fairness และ friendliness เป็นหลักในการตัดสินสมรรถนะ แบบแผนควบคุมความคับคั่งที่นำมาพิจารณาสมรรถนะมี 3 แบบแผน คือ แบบแผนควบคุมความคับคั่ง TCP แบบมาตราฐาน (TCP-Newreno), TCP-Peach และ TCP-Vegas แล้วนำแบบแผนควบคุมความคับคั่งดังกล่าวมาเปรียบเทียบสมรรถนะเพื่อหาแบบแผนที่เหมาะสมกับการสื่อสารผ่านดาวเทียม และเพื่อศึกษาและเปรียบเทียบข้อดีและข้อเสียของแบบแผนควบคุมความคับคั่ง TCP รุ่นต่างๆ

1.3 ขอบเขตของงานวิจัย

1.3.1 ประเมินหาสมรรถนะแบบแผน TCP-Newreno, TCP-Peach และ TCP-Vegas

1.3.2 เปรียบเทียบสมรรถนะแบบแผน TCP-Newreno, TCP-Peach และ TCP-Vegas ในสภาพต่างๆ

- 1.3.2.1 สภาพแวดล้อมที่มีการติดต่อจุดเดียว (simple single-connection scenario)
- 1.3.2.2 สภาพแวดล้อมที่มีความคับคั่งจุดเดียว (single bottleneck scenario)
- 1.3.2.3 สภาพแวดล้อมที่มีความคับคั่งหลายจุด (multiple bottleneck scenario)
- 1.3.2.4 สภาพแวดล้อมของโครงข่ายดาวเทียม (satellite scenario)

1.3.3 วิเคราะห์ผลการทดลองเพื่อหาแบบแผนควบคุมความคับคั่งที่เหมาะสมกับการสื่อสารผ่านดาวเทียม

1.4 วิธีการดำเนินงานวิจัย

1.4.1 ศึกษาความรู้พื้นฐานของโพรโตคอล TCP และโพรโตคอล IP

1.4.2 ศึกษาแบบแผนควบคุมความคับคั่งTCP-Newreno, TCP-Peach และ TCP-Vegas

1.4.3 ศึกษาคุณสมบัติของการสื่อสารผ่านโครงข่ายดาวเทียม

1.4.4 วิเคราะห์หาปัญหาการใช้โพรโตคอล TCP/IP สื่อสารผ่านโครงข่ายดาวเทียม

1.4.5. ศึกษาการทำงานของ NS-2 Simulator

1.4.6 หาสมรรถนะของโพรโตคอล TCP-Newreno, TCP-Peach และ TCP-Vegas

1.4.7 เปรียบเทียบสมรรถนะโพรโตคอล TCP-Newreno, TCP-Peach และ TCP-Vegas ในสภาพแวดล้อมที่กำหนด

1.4.8 วิเคราะห์ผลการทดลองเพื่อหาแบบแผนควบคุมความคับคั่งที่เหมาะสมกับการสื่อสารผ่านดาวเทียม

1.4.9 สรุปผลการทดลอง

1.4.10 รวบรวมข้อมูลเพื่อเขียนวิทยานิพนธ์ฉบับสมบูรณ์

1.5 ประโยชน์ที่คาดว่าจะได้รับ

1.5.1 มีความรู้ความเข้าใจโพรโตคอล TCP/IP ซึ่งเป็นโพรโตคอลที่สำคัญในระบบอินเตอร์เนต

1.5.2 มีความรู้ความเข้าใจในระบบดาวเทียม และการสื่อสารด้วยโพรโตคอล TCP/IP ผ่านโครงข่ายดาวเทียม

1.5.3 สามารถแยกแยะข้อดีและข้อเสียของโพรโตคอล TCP แบบต่างๆ ได้เพื่อเป็นประโยชน์ในการนำไปประยุกต์ใช้งานต่อไป

1.5.4 สามารถหาโพรโตคอลที่เหมาะสมกับการสื่อสารผ่านดาวเทียมได้

1.5.5 เป็นแนวทางในการทำวิจัยการออกแบบโพรโตคอล TCP ต่อไป

1.6 ภาพรวมของวิทยานิพนธ์

เนื้อหาของวิทยานิพนธ์ฉบับนี้ แบ่งออกเป็น 5 บท บทที่ 1 จะเป็นบทนำ บทที่ 2 จะกล่าวถึงทฤษฎีพื้นฐานที่เกี่ยวข้อง อาทิเช่น ความรู้พื้นฐานของโพรโตคอล TCP หลักการสื่อสารผ่านโครงข่ายดาวเทียม ประเภทของดาวเทียม ความรู้พื้นฐานของแบบแผนควบคุมความคับคั่ง ปัญหาของแบบแผนควบคุมความคับคั่งที่เกิดขึ้นเมื่อใช้บนโครงข่ายดาวเทียม วิธีแก้ปัญหาที่ใช้อยู่ในปัจจุบันและอธิบาย

รายละเอียดของอัลกอริธึมของแบบแผนควบคุมความคับคั่ง TCP-Newreno TCP-Peach และ TCP-Vegas ส่วนในบทที่ 3 จะกล่าวถึงแนวคิดในการดำเนินการวิจัย เสนอวิธีการสร้างแบบจำลองโครงข่ายเพื่อทดลองหาสมรรถนะของแบบแผนควบคุมความคับคั่งแบบต่างๆ ส่วนในบทที่ 4 จะเป็นส่วนของผลการทดลองที่ได้จากแบบจำลองโครงข่ายจากบทที่ 3 และในบทที่ 5 จะเป็นส่วนสรุปและอภิปรายข้อเสนอแนะข้อคิดเห็นตลอดจนแนวทางในการพัฒนาวิธีการและอัลกอริทึมต่อไป

บทที่ 2

เอกสารและงานวิจัยที่เกี่ยวข้อง

2.1 ทฤษฎีและความรู้พื้นฐาน

2.1.1 ความรู้พื้นฐานเกี่ยวกับโพรโตคอล TCP [1]

2.1.1.1 TCP (Transmission Control Protocol)

โพรโตคอล TCP เป็นโพรโตคอลที่มีคุณสมบัติการส่งชุดข้อมูลเป็นไบต์ (byte stream), มีการสร้างเส้นทางการติดต่อแบบ connection-oriented และเป็นโพรโตคอลที่รับประกันการส่งข้อมูลถูกต้องจากต้นทางไปปลายทางโดยมีคุณสมบัติดังกล่าวมีรายละเอียดดังนี้

การส่งชุดข้อมูลเป็นไบต์

TCP เป็นโพรโตคอลที่เชื่อมต่ออยู่ระหว่าง session layer ขึ้นไปถึง application layer กับ network layer ลงมา เมื่อ application layer ส่งข้อมูลมาที่ TCP ชุดข้อมูลจะเป็นชุดละ 8 บิต (ไบต์) เสมอ TCP จะจัดข้อมูลที่ได้รับมาเป็นเซ็กเมนต์เพื่อส่งต่อไปยังผู้รับ โดยทั่วไปโพรโตคอล TCP จะให้ขนาดของแต่ละเซ็กเมนต์มีค่าเท่ากับ 1 KB

Connection-Oriented

ก่อนที่จะมีการรับส่งข้อมูลกันระหว่างผู้ส่งและผู้รับในชั้นการส่งข้อมูลบน TCP จะต้องมีการสร้างเส้นทางเชื่อมต่อกันก่อนเสมอ ดังนั้นการแลกเปลี่ยนข้อมูลบน TCP จะถูกส่งไปบนเส้นทางเดียวกัน ซึ่งจะต่างไปจากการแลกเปลี่ยนข้อมูลบนชั้น IP ที่ไม่จำเป็นต้องสร้างเส้นทางขึ้นมาก่อน (connectionless)

ความน่าเชื่อถือ (reliability)

การที่ TCP สามารถรับประกันการส่งข้อมูลให้ถึงปลายทางได้ TCP จะอาศัยกระบวนการดังต่อไปนี้

Checksums เซ็กเมนต์ TCP ทุกเซ็กเมนต์จะมีฟังก์ชัน checksum อยู่ ซึ่งทำหน้าที่ตรวจจับความเสียหาย (link error) ในการส่งข้อมูลทั้งส่วนหัวข้อมูล (header) และส่วนของตัวข้อมูล

กระบวนการตรวจจับความซ้ำซ้อนของชุดข้อมูล ในโครงข่ายการสวิทช์แบบแพ็กเกตอาจทำให้เกิดแพ็กเกตซ้ำได้ โดยมากจะเกิดขึ้นจากการกระบวนการส่งข้อมูลซ้ำ (retransmission) โดยที่แพ็กเกตชุดแรกอาจถูกประวิงเวลาและแพ็กเกตชุดที่สองถูกส่งเนื่องจากการขาด acknowledgement (ACK) ทำให้ผู้รับสามารถรับแพ็กเกตนี้ซ้ำกันได้ TCP จะคอยติดตามชุดข้อมูลที่ได้รับเพื่อที่จะยกเลิกชุดข้อมูลที่ได้รับไปแล้ว

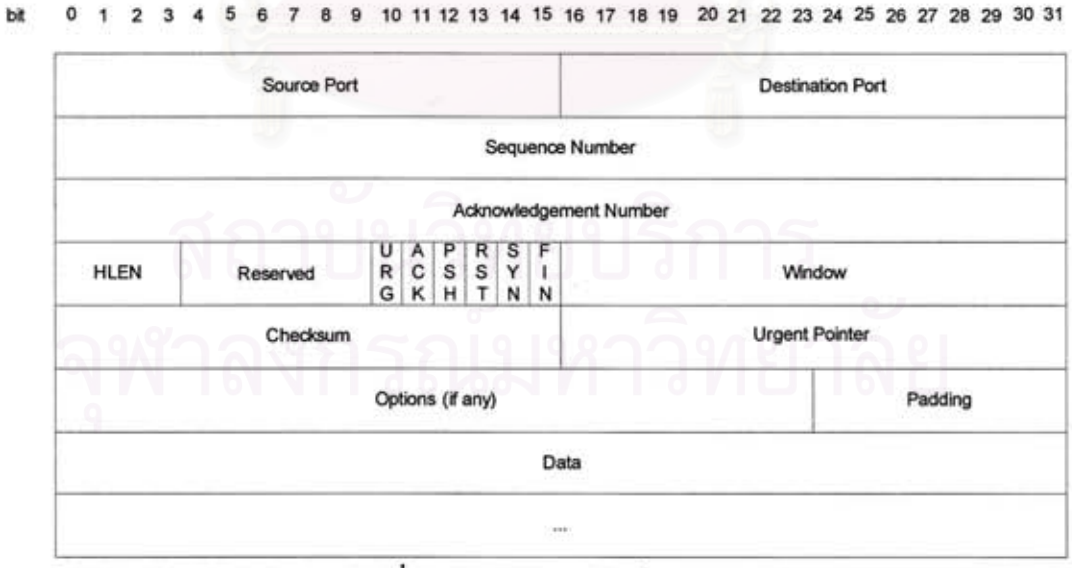
การส่งข้อมูลซ้ำ เป็นกระบวนการรับประกันการส่งข้อมูลบน TCP ซึ่งจะส่งแพ็กเกตที่สูญหายไปใหม่อีกครั้งหนึ่ง การส่งชุดข้อมูลเมื่อถึงผู้รับแล้ว ผู้รับจะส่ง positive acknowledgement กลับไปบอกผู้ส่งว่าการส่งชุดข้อมูลชุดนั้นสำเร็จ แต่หากการส่งชุดข้อมูลนั้นไม่สำเร็จ จะทราบได้จากการขาด positive acknowledgement หรือ การหมดคาบเวลา (timeout) และทำให้เกิดกระบวนการส่งข้อมูลซ้ำ

หมายเลขลำดับ ในโครงข่ายการสวิทช์แบบแพ็กเกต สามารถทำให้เกิดแพ็กเกตที่ผู้รับได้รับไม่เรียงหมายเลขได้ TCP จะทำหน้าที่จัดเรียงหมายเลขชุดข้อมูลใหม่ ก่อนที่จะส่งต่อไปที่ application layer

ตัวจับเวลา TCP สามารถจัดการกับตัวนับเวลาได้ทั้งแบบคงที่ (static) และไม่คงที่ (dynamic) TCP จะรอผู้รับตอบ acknowledgement กลับมาในช่วงเวลาที่กำหนด หากเกินช่วงเวลาที่กำหนด TCP จะเข้าสู่กระบวนการส่งข้อมูลซ้ำ

2.1.1.2 รูปแบบของหัวข้อมูล TCP

เซ็กเมนต์ TCP คือ หัวข้อมูล TCP รวมกับ TCP หนึ่งแพ็กเกต แสดงไว้ดังรูปที่ 1 ขนาดของหัวข้อมูลไม่รวมตัวเลือก (options) มีขนาดเท่ากับ 20 ไบต์



รูปที่ 2.1 รูปแบบของหัวข้อมูล TCP

พอร์ต (port)

ตัวเลข 16 บิต ระบุถึงโปรแกรมประยุกต์ที่เซ็กเมนต์ TCP ถูกส่งมาจากฝั่งผู้ส่ง ตัวเลขพอร์ต แบ่งได้เป็น 3 ช่วง คือพอร์ตสาธารณะคือพอร์ตหมายเลข 0 ถึง 1023 ส่วนพอร์ตลงทะเบียนคือพอร์ตหมายเลข 1024 ถึง 49151 และพอร์ตส่วนบุคคลคือพอร์ตหมายเลข 49152 ถึง 65535 การกำหนดหมายเลขพอร์ตเพื่อให้ TCP สามารถเชื่อมต่อกับโปรแกรมประยุกต์ใน application layer ได้อย่างถูกต้อง ตัวอย่างเช่น เทลเน็ตเซิฟเวอร์จะถูกกำหนดขึ้นให้เป็นพอร์ตสาธารณะหมายเลข 23 เมื่อรวมหมายเลขพอร์ตและ หมายเลข IP จะแสดงถึงการมีอยู่หนึ่งเดียว (unique) ของการส่งข้อมูลในโพรโตคอล TCP บนโลกอินเตอร์เนต

พอร์ตปลายทาง

ตัวเลข 16 บิต ระบุถึงโปรแกรมประยุกต์ทางฝั่งผู้รับ มักจะเป็นหมายเลขเดียวกันกับหมายเลขพอร์ตต้นทาง

หมายเลขลำดับ (Sequence Number)

ตัวเลข 32 บิต ระบุหมายเลขของแพ็กเกต เพื่อให้ผู้รับสามารถแยกแยะแพ็กเกตและเรียงแพ็กเกตเข้าด้วยกันได้ มีจำนวนหมายเลขเท่ากับ 2^{32} -1 หมายเลข และจะถูกกลับไปนับ 0 ใหม่ เมื่อกำหนดจนครบหมายเลขทั้งหมด

หมายเลข Acknowledgement

ตัวเลข 32 บิต ระบุถึงข้อมูลชุดต่อไปที่ผู้รับคาดว่าจะได้รับเป็นชุดต่อไป ดังนั้นหมายเลขนี้จะมากกว่าหมายเลขชุดข้อมูลที่ได้รับแล้ว หมายเลขนี้จะถูกใช้ก็ต่อเมื่อบิตควบคุม ACK ถูกเปิดใช้

ความยาวของหัวข้อมูล

ตัวเลข 4 บิตนี้ บอกความยาวของหัวข้อมูล TCP โดยทั่วไปในหัวข้อมูล TCP 1 เซ็กเมนต์จะมีขนาด 20 ไบต์ เมื่อไม่มีตัวเลือกแต่สามารถมีขนาดได้สูงสุด 60 ไบต์ ความยาวของหัวข้อมูลจำเป็นต้องมี เพราะในกรณีที่ใช้ตัวเลือก ถ้าไม่มีความยาวของหัวข้อมูลจะไม่สามารถรู้ได้ว่าใช้ตัวเลือกไปเท่าไร ในมาตราฐาน TCP จะเรียกความยาวของหัวข้อมูลว่า “data offset”

สำรอง (Reserved)

ตัวเลข 6 บิต นี้สำรองไว้ใช้ในอนาคต

บิตควบคุม

Urgent Pointer (URG) ถ้าบิตนี้ถูกเปิดใช้ ผู้รับจะแปลความหมายฟิลด์นี้

Acknowledgement (ACK) ถ้าบิตนี้ถูกเปิดใช้ จะมีกระบวนการ acknowledgement จะถูกใช้

Push Function (PSH). ถ้าบิตนี้ถูกเปิดใช้ ผู้รับจะส่งข้อมูลชุดนั้นไปที่โปรแกรมประยุกต์ทันที ตัวอย่างการใช้บิตนี้ คือ การส่ง Control-BREAK ที่โปรแกรมประยุกต์ต้องการทันที

Reset the Connection (RST) ถ้าบิตนี้ถูกเปิดใช้ จะเป็นสัญญาณที่บอกให้ปลายทางว่าต้นทางได้ยกเลิกการติดต่อ และให้คิวข้อมูลถูกนำกลับไปเริ่มต้นนับใหม่

Synchronize (SYN) ใช้ในกระบวนการเริ่มต้นของการติดต่อ ในช่วงเริ่มการสร้างเส้นทาง เพื่อบอกปลายทางถึงความพยายามที่ต้นทางจะทำการ "synchronize"

No More Data from Sender (FIN) ถ้าบิตนี้ถูกเปิดใช้ จะเป็นการบอกปลายทางว่า ต้นทางได้ส่งข้อมูลจนหมดแล้ว

วินโดว์ (window)

ตัวเลข 16 บิต ระบุการใช้การควบคุมการไหลข้อมูล (flow control) ในการบอกขนาดของวินโดว์ ตัวเลขนี้บอกถึงปริมาณข้อมูลที่ปลางทางยอมรับก่อนที่จะส่ง ACK มีขนาดสูงสุด 65,535 ไบต์ แต่หากใช้ร่วมกับตัวเลือกที่เรียกว่า “window scale” จะสามารถเพิ่มขนาดได้อีก

Checksum

ใช้ในการตรวจสอบความผิดพลาดในการส่งข้อมูล ได้ในระดับหนึ่ง

Urgent pointer

ตัวเลข 16 บิตนี้ จะบอกปลายทางเมื่อไบต์สุดท้ายของชุดข้อมูลเร่งด่วนในเซ็กเมนต์สิ้นสุดลง

ตัวเลือก (Options)

ใช้เป็นฟังก์ชันเพิ่มเติม ฟิลด์นี้จะมีขนาดไม่แน่นอน แต่จะมีขนาดได้ไม่เกิน 40 ไบต์ เนื่องจากขนาดของฟิลด์ความยาวหัวข้อมูลมี 4 บิต โดยทั่วไปจะเป็นขนาดเซ็กเมนต์สูงสุด (maximum segment size, MMS) ใช้ในการบอกต้นทางถึงปริมาณข้อมูลที่ปลายทางสามารถรับได้ ส่วนตัวเลือกอื่นๆ จะใช้ในเทคนิคการทำการควบคุมการไหลข้อมูลและการควบคุมความคับคั่ง

Padding

เนื่องจากตัวเลือกทำให้หัวข้อมูล TCP มีขนาดไม่แน่นอน ทำให้ต้องมีฟิลด์ที่บอกจุดจบของส่วนหัวข้อมูลประกอบไปด้วยศูนย์จำนวน 8 บิต ก่อนที่จะเข้าสู่ส่วนของตัวข้อมูล

ตัวข้อมูล (Data)

เป็นส่วนของข้อมูลที่รับมาจากโปรแกรมประยุกต์ส่งไปที่โปรแกรมประยุกต์ปลายทาง เมื่อรวมเข้ากับหัวข้อมูลจะเรียกว่าเซ็กเมนต์ ถ้าเป็นเซ็กเมนต์ acknowledgement จะไม่มีฟิลด์นี้

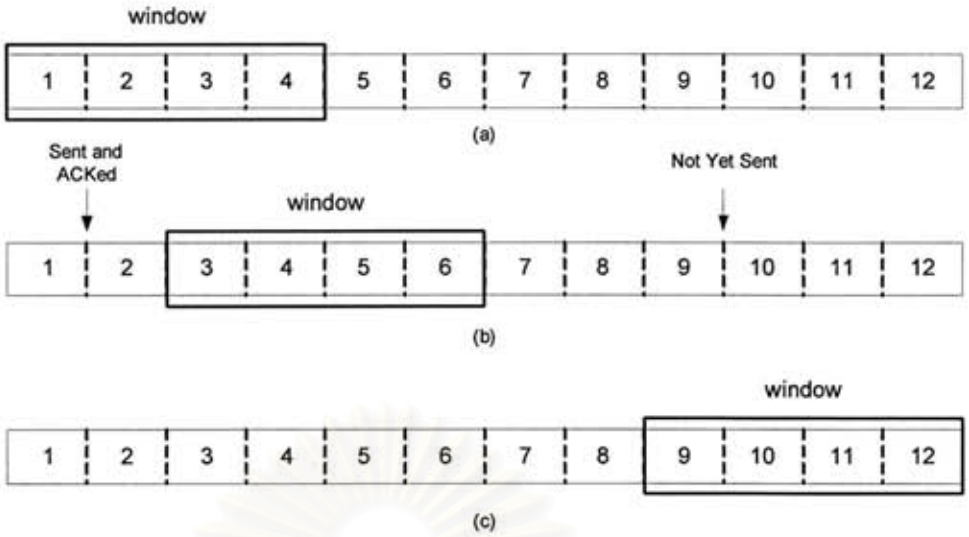
2.1.1.3 การควบคุมการไหลข้อมูล

การควบคุมการไหลข้อมูลเป็นเทคนิคที่มีจุดประสงค์หลักในการควบคุมอัตราการส่งข้อมูลจากต้นทางไปยังปลายทาง เพื่อให้ได้อัตราการส่งข้อมูลที่ความเร็วสูงและให้ได้สมรรถนะที่ดี และต้องยังสามารถป้องกันการรับข้อมูลเกินความสามารถของโครงข่ายได้ด้วย

การควบคุมการไหลข้อมูล จะไม่เหมือนกับ การควบคุมความคับคั่ง ที่จะมีหน้าที่หลักในการจัดการกับการโหลดเกินของอุปกรณ์สื่อกลาง เช่น IP เราต์เตอร์

การควบคุมการไหลข้อมูล จะใช้ฟิลด์วินโดว์ในการบอกขนาดของข้อมูลระหว่างการแลกเปลี่ยนข้อมูลระหว่างต้นทางและปลายทาง สำหรับใช้ในการปรับอัตราการส่งข้อมูล เรียกว่า "การเลื่อนวินโดว์" ดังแสดงในรูปที่ 2

จากรูปที่ 2.2 มีขนาดของวินโดว์เท่ากับ 4 ไบต์ จะเลื่อนจากซ้ายไปขวา เมื่อจำนวนไบต์ภายในวินโดว์ถูกส่งไปและได้ ACK กลับมา ขนาดของวินโดว์และอัตราการเพิ่มหรือลดกำลังอยู่ในระหว่างการวิจัย [4] (AIMD, Additive Increase Multiplication Decrease)



รูปที่ 2.2 เทคนิคการเลื่อนวินโดว์

2.1.1.4 พื้นฐานของแบบแผนควบคุมความคับคั่ง (Congestion Control)

แบบแผนควบคุมความคับคั่งและการจัดการทราฟฟิคบนอินเตอร์เนตกำลังเป็นที่วิจัยกันอย่างกว้างขว้างและมีหลายแบบแผนที่อยู่ระหว่างการทดลอง แต่มาตราฐานโดยทั่วไปของแบบแผนควบคุมความคับคั่งใน TCP อธิบายไว้ใน [2] มีรายละเอียดของกระบวนการดังต่อไปนี้

2.1.1.4.1 กระบวนการ Slow Start

slow start เป็นกระบวนการบน TCP ที่นำไปใช้ในการควบคุมอัตราการส่งข้อมูล บนฝั่งของผู้ส่งอาจเรียกได้ว่า "sender-based flow control" ซึ่งจะมีความสัมพันธ์โดยตรงกับอัตราการรับ acknowledgement ที่ผู้รับส่งกลับมา หรืออาจกล่าวได้ว่า slow start มีหน้าที่ในการประเมินหาแบนด์วิดท์ของช่องสัญญาณ เพราะก่อนทำการส่งข้อมูล ผู้ส่งไม่ทราบข้อมูลใดๆ เกี่ยวกับช่องสัญญาณเลย ดังนั้น slow start จึงเป็นกระบวนการที่ขาดไม่ได้ในการเริ่มส่งข้อมูล

เมื่อการติดต่อครั้งแรกเริ่มต้นขึ้นอัลกอริทึม slow start จะมีค่าเริ่มต้นของ congestion window (*cwnd*) เท่ากับ 1 เซ็กเมนต์ซึ่งก็คือค่า maximum segment size (MMS) ที่กำหนดขึ้นโดยผู้รับเป็นค่าเริ่มต้นระหว่างขั้นตอนการสร้างการติดต่อ (connection establishment phase) เมื่อมี ACK ตอบกลับมาจากผู้รับ ผู้ส่งจะเพิ่ม *cwnd* ขึ้นอีก 1 เซ็กเมนต์ในทุกครั้งที่ได้รับ ACK ตอบกลับมา ดังนั้นผู้ส่งสามารถมีขนาดของวินโดว์อย่างน้อยเท่ากับค่าต่ำสุดระหว่าง *cwnd* ที่กำหนดขึ้นโดยผู้ส่งกับ advertised

window ที่กำหนดขึ้นโดยผู้รับ และขนาดของวินโดว์ที่ส่งออกไปจริงเรียกว่า transmission window

จะเห็นได้ว่า slow start ไม่จำเป็นต้องส่งข้อมูลช้าๆ เสมอไป เมื่อโครงข่ายยังไม่เกิดความคับคั่ง หรือยังมีเวลาการตอบสนองของโครงข่ายที่ดี ยกตัวอย่างเช่น สามารถเพิ่มขนาดของวินโดว์ เป็น 2 เซ็กเมนต์ หลังจากได้ ACK จากการส่งในครั้งแรก และเพิ่มขนาดของวินโดว์ เป็น 4 เซ็กเมนต์ หลังจากได้ ACK จากการส่งในครั้งสอง และเพิ่มขนาดของวินโดว์ เป็น 8 เซ็กเมนต์ หลังจากได้ ACK ในครั้งสาม กล่าวคือมีอัตราการเพิ่มเป็นสองเท่า อย่างนี้ไปเรื่อยๆ จนถึงค่าสูงสุดที่ค่า advertised window หรือ จนเกิดความคับคั่งขึ้น

2.1.1.4.2 กระบวนการ Congestion Avoidance

การเกิดความคับคั่งในโครงข่ายเกิดได้จากการเปลี่ยนโครงข่ายจากโครงข่ายที่มีช่องสัญญาณที่มีความจุขนาดใหญ่ไปยังโครงข่ายที่มีช่องสัญญาณขนาดเล็กกว่า หรือเกิดจากสัญญาณอินพุตหลายสัญญาณเข้ามาในเราต์เตอร์และเอาท์พุตมีความจุต่ำกว่าสัญญาณอินพุตรวมกัน เป็นผลให้เกิดความคับคั่งได้เช่นกัน กระบวนการ congestion avoidance เป็นกระบวนการที่คอยจัดการเมื่อการส่งแพ็กเกตเกิดสูญหาย

การสูญหายของแพ็กเกตอาจเกิดขึ้นได้จากสองกรณี คือ เกิดขึ้นจากความคับคั่งของโครงข่ายจุดใดจุดหนึ่งในช่วงต้นทางถึงปลายทาง หรือเกิดจากความเสียหายของสัญญาณข้อมูลในระหว่างทาง (link error) ข้อสัณนิษฐานเบื้องต้นเมื่อเกิดการสูญหายของแพ็กเกต คือ ให้มีสาเหตุมาจากความคับคั่งในโครงข่าย เพราะความน่าจะเป็นในการเกิดความเสียหายของสัญญาณข้อมูลมีต่ำมาก จุดบ่งชี้การสูญหายของแพ็กเกตอยู่สองประการ คือ หมดคาบเวลาการรอ ACK ทางฝั่งผู้ส่งหรือผู้ส่งได้รับ duplicate ACK

congestion avoidance กับ slow start เป็นกระบวนการที่เป็นอิสระต่อกันและมีจุดประสงค์ต่างกัน แต่เมื่อเกิดความคับคั่งขึ้น โพรโตคอล TCP จะลดอัตราการส่งข้อมูลเข้าไปในโครงข่ายลง และให้ slow start เริ่มทำงานต่ออีกครั้ง แต่ในทางปฏิบัติกระบวนการทั้งสองจะถูกสร้างขึ้นบนอัลกอริทึมเดียวกัน

congestion avoidance และ slow start จะใช้ตัวแปร *cwnd* (congestion window) และ *ssthresh* (slow start threshold size) เป็นตัวแปรในการจัดการกับอัตราการส่งข้อมูล โดยมีอัลกอริทึมที่สร้างรวมกันดังนี้

- กำหนดค่าเริ่มต้นให้ *cwnd = 1* เซ็กเมนต์และค่า *ssthresh = 65,535* ไบต์ (เป็นค่าสูงสุดของขนาดวินโดว์)

- Transmission window จะมีค่าไม่เกินค่าต่ำสุดของ *cwnd* กับ advertised window

- เมื่อเกิดความคับคั่งขึ้น ครึ่งหนึ่งของขนาด transmission window จะให้เป็นค่า *ssthresh* ถ้าความคับคั่งถูกบ่งชี้ว่าเกิดจากการหมดคาบเวลาจะให้ *cwnd* มีค่าเท่ากับ 1 เซ็กเมนต์(เข้าสู่กระบวนการ slow start)

- เมื่อชุดข้อมูลชุดใหม่ที่ส่งไปได้รับ ACK ตอบกลับมา *cwnd* จะถูกเพิ่มขึ้น แต่จะเพิ่มขึ้นลักษณะอย่างไร จะขึ้นอยู่กับว่าขณะนั้นอยู่ที่กระบวนการใด (slow start หรือ congestion avoidance) ถ้าขณะนั้น *cwnd* มีค่าน้อยกว่าหรือเท่ากับค่า *ssthresh* จะอยู่ใน slow start แต่ถ้ามากกว่าจะอยู่ใน congestion avoidance ถ้าอยู่ใน slow start *cwnd* จะถูกเพิ่มขึ้นจนถึงค่า *ssthresh* (เป็นค่าใหม่ที่มีค่าเป็นครึ่งหนึ่งของขนาด transmission window) จากนั้น congestion avoidance จะเข้ามารับหน้าที่แทน

การเพิ่ม *cwnd* ใน congestion avoidance จะเพิ่มขึ้นเท่ากับ *1/cwnd* ทุกครั้งที่ได้รับ ACK ในหนึ่ง *RTT* (Round Trip Time) และไม่สนใจว่าในหนึ่ง *RTT* จะได้รับจำนวน ACK เท่าไร ทำให้ลักษณะการเพิ่มขึ้นของ *cwnd* จะเป็นเชิงเส้น ต่างจากการเพิ่มขึ้น *cwnd* ใน slow start ที่จะเพิ่ม *cwnd* ทุกครั้งตามจำนวน ACK ที่ได้รับในหนึ่ง *RTT* ทำให้ลักษณะการเพิ่ม *cwnd* เป็นแบบเอกซ์โพเนนเชียล

2.1.1.4.3 กระบวนการ Fast Retransmission

เมื่อผู้ส่งได้รับ duplicate ACK อาจมีสาเหตุมาจากการเรียงหมายเลขลำดับทางฝั่งผู้รับไม่สามารถเรียงกันได้ เวลาที่ผู้รับใช้ในการเรียงหมายเลขจะมีค่าไม่มาก ดังนั้นจึงสรุปได้ว่า หากผู้ส่งได้รับ duplicate ACK หนึ่งหรือสอง สาเหตุน่ามาจากการเรียงหมายเลขยังไม่สมบูรณ์เท่านั้น แต่หากได้รับ duplicate ACK มากกว่าสาม อาจสรุปได้ว่ามีสาเหตุมาจากการสูญหายของแพ็กเกต ดังนั้นหากผู้ส่งได้รับสาม duplicate ACK (จำนวนนี้มีค่าไม่แน่นอน แต่โดยมากแล้วจะให้มีค่าเท่ากับสาม) ผู้ส่งจะทำการส่งข้อมูลซ้ำทันที โดยไม่รอให้ถึงเวลาการหมดคาบเวลา

2.1.1.4.4 กระบวนการ Fast Recovery

เมื่อกระบวนการ fast retransmission เกิดขึ้น โดยมีข้อบ่งชี้จากการรับ duplicate ACK ทำให้ผู้ส่งรู้ว่าการส่งข้อมูลยังคงส่งไปที่ผู้รับได้ โดยไม่เกิดความคับคั่ง เพราะว่า duplicate ACK จะเกิดขึ้นได้เมื่อผู้รับได้รับเซ็กเมนต์ TCP ที่ส่งไป นั้นเป็นสิ่งยืนยันได้ว่าไม่มีการเกิดความคับคั่งในโครงข่าย ทำให้แทนที่จะลดอัตราการส่งข้อมูลและกลับไปที่กระบวนการ slow start ก็ทำเพียงเข้าสู่กระบวนการ congestion avoidance

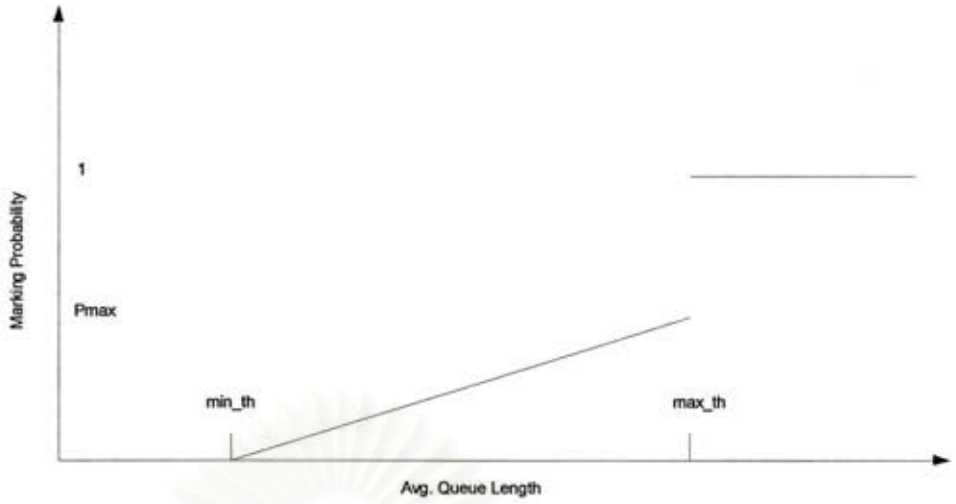
2.1.1.5 RED (Random Early Detection) และ ECN (Explicit Congestion Notification) [12]

อุปกรณ์เราต์เตอร์เป็นอุปกรณ์ที่สามารถกำหนดอัตราการส่งข้อมูลใน TCP ได้ ในแง่ของการจัดการคิว (active queue management, AQM) การจัดการคิวแบบ RED (Random Early Detection) เป็นแบบแผนการจัดการคิว ที่ใช้บนเราต์เตอร์ซึ่งจะคอยส่งแจ้งความน่าเป็นในการเกิดการละทิ้งแพ็กเกตบนเราต์เตอร์ก่อนที่จะเกิดการล้นของหน่วยความจำ (overflow) ค่า min_{th} และ max_{th} และความยาวคิวเฉลี่ย (average queue length, *avg*) ซึ่งคำนวณได้จากค่าเฉลี่ยการเคลื่อนที่ (moving average) ของความยาวคิวในขณะนั้นบนเราต์เตอร์ จะใช้ในการวัดหาความน่าจะเป็นในการเกิดความคับคั่ง

เราต์เตอร์จะละทิ้งแพ็กเกตด้วยความน่าจะเป็นเริ่มต้นที่กำหนดไว้ ตามสมการที่ 1 ถ้าค่า *avg* อยู่ระหว่างค่า min_{th} และ max_{th} ถ้าอยู่ต่ำกว่าค่า min_{th} จะไม่ละทิ้งแพ็กเกต แต่ถ้าสูงกว่าค่า max_{th} จะทำการละทิ้งแพ็กเกตทั้งหมด (ละทิ้งด้วยความน่าจะเป็นเท่ากับหนึ่ง) เมื่อเกิดการละทิ้งแพ็กเกต สัญญาณจะถูกส่งไปบอกผู้รับให้ส่ง duplicate ACK ไปบอกผู้ส่งให้ลดอัตราการส่งข้อมูลโดยตามแบบแผนควบคุมความคับคั่งที่อยู่บนผู้ส่ง

$$P = P_{\max} \frac{avg - \min_{th}}{\max_{th} - \min_{th}} \qquad \text{สมการที่ 1}$$

ECN (Explicit Congestion Notification) คือแบบแผนที่พัฒนาต่อจาก RED โดยแทนที่ ECN จะละทิ้งแพ็กเกตเหมือน RED แต่จะทำเครื่องหมายบอกความคับคั่งที่กำลังจะเกิดขึ้นแทน แสดงดังรูปที่ 2.3 โดยเราต์เตอร์จะใช้บิต CE ใน IP header และจะกำหนดบิตนี้ให้เป็นหนึ่งเมื่อค่า *avg* เกินค่าที่กำหนดไว้ ทำให้ผู้รับรู้สภาพของความคับคั่ง และจะส่งไปบอกผู้ส่งด้วย ACK ที่กำหนดบิต ECE (Explicit Congestion Echo) ในหัวข้อมูล TCP ให้มีค่าเป็นหนึ่ง



รูปที่ 2.3 การทำเครื่องหมายบอกความคับคั่งบนแพ็กเกต [12]

2.1.2 การสื่อสารผ่านโครงข่ายดาวเทียมด้วยโพรโตคอล TCP

การสื่อสารผ่านโครงข่ายดาวเทียมจะมีลักษณะของช่องสัญญาณอย่างไรขึ้นอยู่กับหลายปัจจัย ยกตัวอย่างเช่น ประเภทของวงโคจรของดาวเทียมที่ใช้ในการสื่อสารซึ่งมีผลต่อเวลาไปและกลับ *RTT* ตัวแปรนี้มีผลต่อระยะเวลาที่อยู่ในกระบวนการ slow start นอกจากนั้นสภาพอากาศของแต่ละประเทศก็มีผลเช่นกัน ในกรณีที่มีฝนตกระหว่างการสื่อสารดาวเทียมจะทำให้ค่ากำลังสัญญาณทางฝั่งขารับต่ำลง และยังมีค่าความผิดพลาดบิตสูงขึ้นด้วย ซึ่งค่าตัวแปรเหล่านี้มีผลต่อการสื่อสารผ่านดาวเทียมด้วยโพรโตคอล TCP ซึ่งจะทำให้กระบวนการ congestion avoidance คิดว่าการเกิดความผิดพลาดบิตเป็นการเกิดความคับคั่งขึ้นในระบบ จนมีผลทำให้กระบวนการ congestion avoidance ลดขนาดของ *cwnd* ลงครึ่งหนึ่ง ส่งผลให้ goodput ของระบบมีค่าลดลงไปด้วย

2.1.2.1 ประเภทวงโคจรของดาวเทียม

ประเภทวงโคจรของดาวเทียมสามารถแบ่งตามความสูงของวงโคจรจากพื้นโลกได้ โดยมีรายละเอียดดังนี้

2.1.2.1.1 Low-Earth Orbit (LEO) โคจรสูงจากพื้นโลกประมาณ 700 - 2,000 กิโลเมตรและใช้เวลาเดินทางของสัญญาณไปกลับ (*RTT*) ประมาณ 0.05 วินาที และมีคาบการโคจรรอบโลกประมาณ 100 – 127 นาที

2.1.2.1.2 Medium-Earth Orbit (MEO) โคจรสูงจากพื้นโลกประมาณ 10,000 กิโลเมตรและใช้เวลาเดินทางของสัญญาณไปกลับ (*RTT*) ประมาณ 0.25 วินาที

2.1.2.1.3 Geosynchronous-Earth Orbit (GEO) โคจรสูงจากพื้นโลกประมาณ 36,000 กิโลเมตรและใช้เวลาเดินทางของสัญญาณไปกลับ (*RTT*) ประมาณ 0.55 วินาที และมีคาบการโคจรรอบโลกประมาณ 24 ชั่วโมงซึ่งหากมองจากพื้นโลกจะเห็นดาวเทียมอยู่กับที่ (มีความเร็วสัมพัทธ์เป็นศูนย์เมื่อเทียบกับโลก)

2.1.2.2 คุณสมบัติของช่องสัญญาณในระบบดาวเทียมกับผลกระทบในโพรโตคอล TCP [6],[8],[9]

ช่องสัญญาณของการสื่อสารในระบบดาวเทียมจะมีค่า delay-bandwidth product สูง ค่า delay-bandwidth product คือ ปริมาณข้อมูลที่ส่งไปยังผู้รับได้โดยไม่ต้องมี ACK ตอบกลับมายังผู้ส่ง หรือกล่าวอีกมุมมองได้ว่า หากมีท่อเชื่อมต่อจากผู้ส่งไปยังผู้รับ และการส่งข้อมูลจะส่งผ่านท่อไป delay-bandwidth product จะเป็นปริมาณข้อมูลทั้งหมดที่สามารถบรรจุอยู่ในท่อนั้นได้ ค่า delay-bandwidth product หาค่าได้จาก *RTT*BW/2* เมื่อ *BW* คือ แบนด์วิดท์ของช่องสัญญาณ ในระบบดาวเทียมจะมีค่า *RTT* และ *BW* มีค่าสูง ทำให้ค่า delay-bandwidth product มีค่าสูงกว่าระบบในภาคพื้นดิน

ใน slow start เวลาที่ใช้อยู่ในกระบวนการนี้หาค่าได้จากสมการที่ 2 และเมื่อนำไปคำนวณในระบบดาวเทียมจะได้เวลาที่อยู่ใน slow start แสดงไว้ในตารางที่ 1

$$t_{slowstart} = RTT * (1 + \log(\frac{BW * RTT}{l})) \qquad \text{สมการที่ 2}$$

l คือ ค่าความยาวเฉลี่ยของแพ็กเกต

ตารางที่ 2.1 ระยะเวลาช่วง Slow Start สำหรับดาวเทียมชนิด LEO, MEO และ GEO [10]

ชนิดของดาวเทียม (Satellite Type)	RTT (Round Trip Time)	$t_{SlowStart}$ (BW = 1 Mb/sec)	$t_{SlowStart}$ (BW = 10 Mb/sec)	$t_{SlowStart}$ (BW = 155 Mb/sec)
LEO (Low Earth Orbit)	0.05 วินาที	0.18 วินาที	0.35 วินาที	0.55 วินาที
MEO (Medium Earth Orbit)	0.25 วินาที	1.49 วินาที	2.32 วินาที	3.31 วินาที
GEO (Geostationary Earth Orbit)	0.55 วินาที	3.91 วินาที	5.73 วินาที	7.91 วินาที

จากตารางที่ 2.1 จะเห็นได้ว่าในระบบดาวเทียมจะมีช่วงเวลาที่อยู่ใน slow start นาน ซึ่งเป็นผลเสียต่อการใช้งานประยุกต์ที่ใช้ TCP มีหลายครั้งที่การส่งข้อมูลเสร็จสิ้นภายในช่วง slow start ทำให้ผู้ใช้ไม่มีโอกาสที่จะใช้ความเร็วสูงสุด ยกตัวอย่างเช่น ใน HTTP (อยู่ใน application layer) มีลักษณะการใช้งานที่ต้องสร้างการติดต่อ TCP ใหม่อยู่เสมอ (ทุกครั้งที่มีการเรียกข้อมูลบน webpage) ทำให้สมรรถนะการใช้งานอินเตอร์เนตผ่านระบบดาวเทียมไม่มีประสิทธิภาพและไม่สามารถใช้ทรัพยากรได้อย่างเต็มที่ ด้วยเหตุนี้จึงได้มีงานวิจัยที่ได้เสนอวิธีการแก้ปัญหาดังจะกล่าวในหัวข้อถัดไป

ช่องสัญญาณของการสื่อสารในระบบดาวเทียมมีค่าความผิดพลาดบิต *BER* (bit error rate) สูงกว่าระบบภาคพื้นดินเช่นกัน ค่า *BER* สูงๆจะทำให้เกิดปัญหาใน slow start และใน congestion avoidance เพราะเมื่อแพ็กเกตเกิดเสียหายระหว่างทาง จะทำให้ transmission window มีค่าลดลง และยิ่งมีค่า BER สูง ยิ่ง ทำให้ transmission window ลงลงมากด้วย และอีกประเด็นที่สำคัญ คือ ผู้ส่งไม่รู้ว่าการสูญหายของแพ็กเกตเกิดจากสาเหตุใด แต่จะสมมุติให้เกิดจากความคับคั่งในโครงข่าย ทั้งที่เกิดจากความเสียหายของแพ็กเกตระหว่างทาง ทำให้ต้องเข้าสู่อัลกอริทึม slow start และ congestion avoidance เพื่อลด transmission window ที่เข้าสู่โครงข่าย แต่จริงๆแล้วถ้าเกิดความเสียหายของแพ็กเกต จะไม่จำเป็นต้องลด transmission window ทำให้เสียสมรรถนะการทำงานไปโดยไม่จำเป็น

2.2 แบบแผนควบคุมความคับคั่งรุ่นต่างๆ

แบบแผนควบคุมความคับคั่งในโพรโตคอล TCP มีเสนอไว้มากมาย แต่ที่จะนำมาประเมินหาสมรรถนะในงานวิจัยนี้ จะใช้แบบแผนควบคุมความคับคั่ง TCP-Newreno, TCP-Peach และ TCP-Vegas โดยมีรายละเอียดดังนี้

2.2.1 TCP-Newreno [2]

TCP-Newreno เป็นพื้นฐานมาจากแบบแผนควบคุมความคับคั่ง TCP-Reno สำหรับการออกแบบอัลกอริธึมของกระบวน fast recovery ได้อธิบายไว้ที่ RFC2981 ผู้ส่งข้อมูลบนโพรโตคอล TCP จะทำกระบวนการ fast retransmission ก็ต่อเมื่อเกิดการหมดคาบเวลาหรือผู้ส่งได้รับ duplicate acknowledge จำนวน 3 ครั้ง การที่เกิดกระบวนการส่งข้อมูลซ้ำ เมื่อแพ็กเกตหายไปเพียงหนึ่งแพ็กเกต อาจต้องส่งแพ็กเกตหลายแพ็กเกตไปซ้ำก็ได้ ในกรณีนี้ถ้ามีการใช้ตัวเลือก SACK (Selective Acknowledge) จะทำให้ผู้ส่งรู้ข้อมูลว่าแพ็กเกตไหนบ้างที่เกิดการสูญหายจริงๆ ทำให้ผู้ส่งส่งแพ็กเกตที่สูญหายเท่านั้นในระหว่างกระบวนการ fast recovery แต่หากไม่สามารถเลือกตัวเลือก SACK ได้ จะทำให้ผู้ส่งมีข้อมูลไม่เพียงพอที่จะตัดสินใจว่าให้ทำการส่งข้อมูลซ้ำหรือไม่ ในช่วงกระบวนการ fast recovery จากการที่ได้รับ 3 duplicate acknowledgement ผู้ส่งสามารถตัดสินใจได้ว่าเป็นการสูญหายของแพ็กเกต และจะส่งข้อมูลที่สูญหายกลับไป หลังจากเวลานี้ผู้ส่งอาจจะได้รับ duplicate acknowledgement อีกเพราะแพ็กเกตที่ส่งออกไปอาจจะกำลังอยู่โครงข่ายระหว่างทาง

ในกรณีของการสูญหายของแพ็กเกตหลายแพ็กเกตบนวินโดว์เดียวกัน ผู้ส่งจะได้รับ acknowledgement สำหรับแพ็กเกตที่สูญหายซึ่งจะถูกส่งข้อมูลซ้ำในช่วงเริ่มแรกของกระบวนการ fast retransmission ส่วนในกรณีของการสูญหายของแพ็กเกตเพียงแพ็กเกตเดียว แพ็กเกต acknowledgement ที่ได้รับจะบ่งชี้ให้ส่งแพ็กเกตทุกแพ็กเกตก่อนจะเข้าสู่กระบวนการ fast retransmission อย่างไรก็ตามถ้ามีแพ็กเกตสูญหายไปหลายแพ็กเกต แพ็กเกต acknowledgement ที่ได้รับจะบ่งชี้ให้ส่งแพ็กเกตบางแพ็กเกตแต่ไม่ครบทุกแพ็กเกตก่อนจะเข้าสู่กระบวนการ fast retransmission เราเรียกการ acknowledge อย่างนี้ว่า partial acknowledge

ในกระบวนการ fast recovery ผู้ส่งจะตอบสนองต่อ partial acknowledge โดยตัดสินให้แพ็กเกตที่อยู่ในลำดับเดียวกันสูญหายและทำการส่งแพ็กเกตที่สูญหายนี้อีกครั้งหนึ่ง

TCP-Newreno เป็นแบบแผนควบคุมความคับคั่งที่พัฒนากระบวนการ fast recovery มาจาก TCP-Reno เริ่มต้นเมื่อได้รับ 3 duplicate acknowledge และจะจบลงเมื่อเกิดการหมดคาบเวลาหรือได้รับหมายเลขแพ็กเกต acknowledge ที่สูงขึ้นซึ่งครอบคลุมหมายเลขที่ทำการ retransmit

2.2.1.1 อัลกอริธึมของ TCP-Newreno มีรายละเอียดดังต่อไปนี้

2.2.1.1.1 3 duplicate acknowledge

เมื่อผู้ส่งได้รับ 3 duplicate acknowledge และผู้ส่งไม่ได้อยู่ในกระบวนการ fast recovery ให้ตรวจสอบค่า acknowledge สะสมว่ามีค่ามากกว่าหมายเลขลำดับเริ่มต้นของการส่งหรือไม่ ถ้าตรงตามเงื่อนไขให้ไปข้อ ก. ถ้าไม่ตรงตามเงื่อนไขให้ไปข้อ ข.

ก. ใช้กระบวนการ fast retransmission

ให้กำหนดค่า *ssthresh* ให้มีค่าตามสมการที่ 3

$$ssthresh = \max(\frac{FlightSize)}{2,2 * SMSS}) \qquad \text{สมการที่3}$$

FlightSize คือ ค่าจำนวนของข้อมูลที่ถูกส่งไปแต่ยังไม่ได้ acknowledge กลับมา

SMSS (Sender Maximum Segment Size) คือ ขนาดสูงสุดของเซ็กเมนต์ทางฝั่งผู้ส่ง

บันทึกค่าสูงสุดของหมายเลขลำดับเริ่มต้นของการส่งและไปข้อที่ 2.2.1.1.2

ข. ไม่ใช้กระบวนการ fast retransmission

ห้ามเข้าสู่กระบวนการ fast retransmission และกระบวนการ fast recovery โดยเฉพาะห้ามเปลี่ยนแปลงค่า *ssthresh* และห้ามไปข้อ 2.2.1.1.2 คือห้ามมีการส่งข้อมูลซ้ำแพ็กเกตที่สูญหาย และห้ามกระทำข้อ 2.2.1.1.3

2.2.1.1.2 เข้าสู่กระบวนการ fast retransmission

ให้ทำการส่งข้อมูลซ้ำแพ็กเกตที่สูญหายและกำหนดค่า *cwnd* ให้มีตามสมการที่4

$$cwnd = ssthresh + 3 * SMSS \qquad \text{สมการที่4}$$

การเพิ่มขึ้นของ *cwnd* เป็นจำนวน 3 เซ็กเมนต์จะไปอยู่ในโครงข่ายและอยู่ที่ buffer ของผู้รับ

2.2.1.1.3 fast recovery

สำหรับ duplicate acknowledge ที่เพิ่มเข้ามาในระหว่างกระบวนการ fast recovery ค่า *cwnd* จะ มีค่าเพิ่มหนึ่งเซ็กเมนต์ ค่าที่เพิ่มขึ้นนี้จะไปอยู่ในโครงข่าย

2.2.1.1.4 เมื่อได้รับ acknowledge ของข้อมูลชุดใหม่ acknowledge นี้จะเป็นของการส่งข้อมูลซ้ำ ในข้อ 2.1.1.2 หรือ เป็นของการส่งข้อมูลซ้ำ ในขั้นต่อมา

ก. Full Acknowledgement

ถ้า ACK ที่ได้รับ acknowledge ทุกหมายเลขข้อมูลและรวมถึงหมายเลขเริ่มต้นของการส่ง แล้ว ACK นี้จะ acknowledge ทุกเซ็กเมนต์กลางทางที่ส่งไประหว่างการส่งของเซ็กเมนต์ที่สูญหายและการรับของ 3 duplicate ACK ให้กำหนดค่าของ *cwnd* ตามสมการที่3 หรือ ตามสมการที่5

$$cwnd = \min(ssthresh, FlightSize, SMSS) \qquad \text{สมการที่5}$$

หมายเหตุ:

FlightSize ในสมการที่4 หมายถึง ค่าจำนวนของข้อมูลที่ถูกส่งไปแต่ยังไม่ได้ acknowledge กลับมาเมื่อออกจากกระบวนการ fast recovery

FlightSize ในสมการที่3 หมายถึง ค่าจำนวนของข้อมูลที่ถูกส่งไปแต่ยังไม่ได้ acknowledge กลับมาเมื่อเข้าสู่กระบวนการ fast recovery

ถ้าค่าของ *cwnd* เป็นไปตามสมการที่3 ในกรณีนี้จะมีการวัดจำนวนของข้อมูลเพื่อหลีกเลี่ยงความคับคั่งที่สามารถเกิดขึ้นได้ โดยจำนวนของข้อมูลที่อยู่ในโครงข่ายจะต้องมีค่าน้อยกว่าค่า *cwnd* มากๆ กระบวนการพื้นฐานคือการจำกัดจำนวนของแพ็กเกตที่สามารถถูกส่งเพื่อตอบสนองต่อการ acknowledge เพียงหนึ่งเดียว ซึ่งเป็นที่รู้จักในตัวแปร *maxburst_* ใน NS simulator ขั้นต่อไปคือออกจากกระบวนการ fast recovery

ข. Partial Acknowledgement

ถ้า ACK ที่ได้รับไม่ acknowledge ทุกหมายเลขข้อมูลและรวมถึงหมายเลขเริ่มต้นของการส่ง แล้ว ACK นี้จะเรียกว่า partial acknowledge ในกรณีนี้จะส่งข้อมูลซ้ำเซ็กเมนต์แรกที่ยังไม่มีการ acknowledge การลด *cwnd* ที่ส่งออกให้น้อยลงเท่ากับจำนวนของข้อมูลชุดใหม่ที่ acknowledge โดย ACK สะสม ถ้า partial ACK acknowledge อย่างน้อยหนึ่ง *SMSS* ของข้อมูลชุดใหม่ แล้วให้เพิ่มไบต์ *SMSS* ไว้หลัง *cwnd* ในข้อ 2.2.1.1.3 การเพิ่มขึ้นของ *cwnd* เพื่อสะท้อนถึงเซ็กเมนต์ที่เพิ่มขึ้นมาที่เหลืออยู่ในโครงข่าย ส่งเซ็กเมนต์ชุดใหม่ถ้ามีการนิยามไว้ในตัวแปร *cwnd* การลดลงของวินโดว์บางส่วนนี้พยายามจะประกันว่า เมื่อจบกระบวนการ fast recovery ค่า *ssthresh* จะยังคงอยู่ในโครงข่าย และห้ามออกจากกระบวนการ fast recovery ถ้ามี duplicate ACK ให้ไปทำข้อ 2.2.1.1.3

สำหรับ partial ACK แรกที่มาถึงกระหว่างกระบวนการ fast recovery จะไปตั้งค่าตัวจับเวลาการส่งข้อมูลซ้ำใหม่

2.2.1.1.5 การหมดคาบเวลาการส่งข้อมูลซ้ำ

หลังจากหมดคาบเวลาของการส่งข้อมูลซ้ำเกิดขึ้น ให้บันทึกค่าหมายเลขลำดับสูงสุดในค่าหมายเลขลำดับเริ่มต้นของการส่งและให้ออกจากกระบวนการ fast recovery

ในข้อ 2.2.1.1.1 ได้ระบุถึงการตรวจสอบ ACK สะสมมีค่าครอบคลุมค่า *recover* เพราะ ACK บรรจุหมายเลขลำดับที่ผู้ส่งคาดว่าจะได้รับค่าต่อไป ค่าตัวแปร *ack_number* มีค่าครอบคลุมมากกว่าตัวแปร *recover* เมื่อเป็นไปตามสมการที่6

$$ack\_number - 1 > recover \qquad \text{สมการที่6}$$

ตัวอย่างเช่น จะมีอย่างน้อยหนึ่งไบต์ของข้อมูลเป็น acknowledge นอกเหนือไปจากไบต์สูงสุดที่ยังอยู่ในโครงข่ายเมื่อเข้าสู่กระบวนการ fast retransmit ครั้งสุดท้าย ในข้อ 2.2.1.1.5 ค่า *cwnd* มีค่าลดลงหลังจากได้รับ partial ACK แต่บางครั้งค่า *cwnd* จะมีค่าเพิ่มขึ้นเมื่อได้รับ partial ACK การเพิ่มหรือลดของค่า *cwnd* จะขึ้นอยู่กับ รูป

แบบเดิมของการสูญหายของแพ็กเกต partial ACK อาจ acknowledge วินโดว์ของข้อมูล ในกรณีนี้ ถ้า *cwnd* มีค่าไม่ลดลง ผู้ส่งสามารถส่ง วินโดว์ของข้อมูลไปกลับได้

ส่วนในกรณีที่ไม่สามารถเลือกตัวเลือก SACK ได้ ผู้ส่งจะได้รับข้อมูลที่จำกัดเมื่อประเด็นในเรื่องของการ recovery จากการสูญหายของแพ็กเกตหลายแพ็กเกตจากวินโดว์ของข้อมูลเป็นสิ่งสำคัญ ดังนั้นวิธีที่ดีที่สุดคือการใช้ตัวเลือก SACK

2.2.1.1.6 การตั้งค่าตัวจับเวลาใหม่เพื่อตอบสนองต่อ partial acknowledge

ความแตกต่างในเรื่องของ partial ACK ที่ระบุไว้ในข้อ 2.1.1 เมื่อมีการตั้งค่าตัวจับเวลาใหม่หลังจาก partial acknowledgement อัลกอริธึมในข้อ 2.1.1.5 บอกว่าให้ทำการตั้งค่าตัวจับเวลาใหม่ก็ต่อเมื่อได้รับ partial ACK แรกเท่านั้น ในกรณีนี้ ถ้ามีแพ็กเกตจำนวนมากสูญหายไปจากวินโดว์ของข้อมูล จะมีผลทำให้ตัวจับเวลาทางฝั่งผู้ส่งเกิดการหมดคาบเวลาและทำให้เข้าสู่กระบวนการ slow start อีกครั้ง เราเรียกว่า impatient variant ของกระบวนการ Newreno และ impatient variant ที่กล่าวมานี้จะไม่เหมือนกันกับอัลกอริธึมใน RFC2988 ในเรื่องการตั้งเวลาใหม่หลังจากการส่งแพ็กเกตทุกแพ็กเกตหรือส่งข้อมูลซ้ำ แต่ถ้าการตั้งเวลาใหม่ทำทุกครั้งหลังการได้รับ partial ACK เราจะเรียกว่า Slow-but-Steady variant ของ TCP-Newreno ในกรณีนี้สำหรับ วินโดว์ที่มีการสูญหายของแพ็กเกตเป็นจำนวนมาก ผู้ส่งจะทำการส่งข้อมูลซ้ำแพ็กเกตได้อย่างมากหนึ่งแพ็กเกตต่อหนึ่งเวลาไปกลับ

เมื่อมีแพ็กเกตจำนวนหนึ่ง (N แพ็กเกต) ได้เกิดการสูญหายไปจากวินโดว์ หาก N นั้นมีค่ามาก slow-but-steady variant สามารถอยู่ในกระบวนการ fast recovery ได้สำหรับ N เท่าของเวลาไปกลับ ทำการส่งข้อมูลซ้ำแพ็กเกตที่สูญหายในแต่ละเวลาไปกลับ สำหรับรูปแบบนี้ impatient variant จะให้การ recovery ได้เร็วกว่าและมีสมรรถนะที่ดีกว่า การทดลองการจำลองด้วยสคริปท์ "ns test-suite-newreno.tcl slow1" ใน NS Simulator แสดงให้เห็นว่า impatient variant มีสมรรถนะที่ดีกว่า slow-but-steady variant impatient variant มีความสำคัญสำหรับการติดต่อ TCP ด้วย *cwnd* ที่มีค่าสูงดังแสดงให้เห็นในการจำลองด้วยสคริปท์ "ns test-suite-newreno.tcl impatient4" ใน NS Simulator

มีบางรูปแบบที่ slow-but-steady variant ให้สมรรถนะที่ดีกว่า impatient variant ซึ่งจะเกิดในกรณีที่ มีแพ็กเกตจำนวนไม่มากเกิดสูญหาย เวลาการหมด

คาบเวลาที่ตั้งค่าไว้ มีค่าเล็กพอที่ตัวจับเวลาการส่งข้อมูลซ้ำจะหมดเวลา และสมรรถนะจะดีขึ้นเมื่อไม่มีการหมดคาบเวลาเกิดขึ้น ในการจำลองด้วยสคริปท์ "ns test-suite-newreno.tcl impatient2" และ "ns test-suite-newreno.tcl slow2" ใน NS Simulator จะแสดงในรูปแบบที่กล่าวไว้

slow-but-steady variant สามารถให้ goodput ที่ดีกว่า impatient variant โดยหลีกเลี่ยงการส่งข้อมูลซ้ำที่ไม่จำเป็น ดังแสดงในสคริปท์ "ns test-suite-newreno.tcl impatient3" และ "ns test-suite-newreno.tcl slow3" ใน NS Simulator slow-but-steady variant มีความสามารถที่จะจัดการกับ delay variation ในระบบได้ ในขณะที่ delay spike จะสามารถบังคับให้เกิดการหมดคาบเวลาและเข้าสู่กระบวนการ go-back-N recovery

จากที่กล่าวมา impatient variant และ slow-but-steady variant ยังไม่ใช่ variant ที่มีสมรรถนะดีที่สุด แต่โดยรวมแล้ว impatient variant จะดีกว่า เพราะสมรรถนะที่ไม่ดีของ slow-but-steady variant สำหรับการติดต่อ TCP ด้วยค่า *cwnd* ที่มีค่าสูง

สำหรับการเพิ่มสมรรถนะให้ดีขึ้นโดยมีหลักการให้การ recover จากการสูญหายของแพ็กเกตให้เร็วเหมือนในกระบวนการ slow start ในขณะที่ให้ทำการตั้งค่าตัวจับเวลาใหม่หลังได้รับ partial ACK แต่ก็ยังมีผลต่อสมรรถนะหากไม่สามารถเลือกตัวเลือก SACK ได้

2.2.1.1.7 การส่งข้อมูลซ้ำหลังจากได้รับ partial acknowledgement

Variant ที่เป็นไปได้ที่ตอบสนองต่อ partial acknowledgement จะต้องทำการส่งข้อมูลซ้ำมากกว่าหนึ่งแพ็กเกตหลังจากได้แต่ละ partial acknowledgement และทำการตั้งค่าตัวจับเวลาใหม่ หลังจากทำการส่งข้อมูลซ้ำในแต่ละครั้ง อัลกอริธึมที่กล่าวไว้จะทำการส่งข้อมูลซ้ำออกไปเพียงแพ็กเกตเดียวหลังจากได้แต่ละ partial acknowledgement นี้เป็นอีกทางเลือกที่ดีที่จะหลีกเลี่ยงการส่งข้อมูลซ้ำแพ็กเกตโดยไม่จำเป็น variant ที่ทำให้การ recover ทำได้เร็วขึ้นในกรณีที่มีแพ็กเกตหลายแพ็กเกตเกิดการสูญหาย โดยให้มีความเร็วเทียบเคียงได้กับกระบวนการ slow start ดังนั้นจึงทำการส่งข้อมูลซ้ำแพ็กเกตทีละ 2 แพ็กเกต หลังจากได้รับแต่ละ partial

ACK หากทำตามนี้การ recover N แพ็กเกตที่สูญหายไปจะใช้เวลาน้อยกว่า N เท่าของเวลาไปกลับ

อย่างไรก็ตาม การตอบสนองต่อ partial acknowledgement ที่กล่าวไว้ใน RFC2582 แม้ว่าทั้งสองจะส่งเพียงแพ็กเกตเดียวต่อหนึ่ง partial acknowledgement ก็ตาม ในข้อ 2.1.1.5 กล่าวไว้ว่าผู้ส่งจะตอบสนองต่อ partial acknowledgement โดยการลดขนาดของ *cwnd* เท่ากับขนาดของข้อมูลชุดใหม่ที่ acknowledge โดยเพิ่มเข้าไปที่ไบต์หลังของ *SMSS* ถ้า partial acknowledgement acknowledge อย่างน้อย *SMSS* ไบต์ของข้อมูลชุดใหม่และจะส่งเซ็กเมนต์ใหม่ไปถ้าระบุไว้ตามตัวแปร *cwnd* ตัวใหม่ ดังนั้นจะมีเพียงหนึ่งแพ็กเกตที่ส่งไปก่อนหน้านี้ถูกส่งข้อมูลซ้ำไปต่อการตอบสนองต่อแต่ละ partial acknowledgement แพ็กเกตใหม่ที่เพิ่มเข้ามาก็จะถูกส่งออกไปด้วยเหมือนกันขึ้นอยู่กับจำนวนชุด acknowledge ของข้อมูลชุดใหม่โดย partial acknowledgement ในทางกลับกัน variant ของ TCP-Newreno ก็จะกำหนดค่า *cwnd* ให้มีค่าเท่ากับ *ssthresh* เมื่อได้รับ partial acknowledgement นอกจากนั้นยังมีการติดตามแพ็กเกตที่หลงเหลืออยู่ในโครงข่ายหลังจากได้รับ partial acknowledgement ทำให้สมรรถนะเป็นที่ยอมรับและ variant ในการ recover เมื่อเกิดแพ็กเกตสูญหายหลายแพ็กเกตทำเวลาได้ดีอีกด้วย

2.2.1.1.8 การหลีกเลี่ยงกระบวนการ fast retransmission

ในหัวข้อนี้จะกล่าวถึงการดัดแปลงตัวแปร *recover* และพูดถึงวิธีการแยกความชัดเจนระหว่างการส่งข้อมูลซ้ำแพ็กเกตที่สูญหายกับ 3 duplicate acknowledgement จากการส่งข้อมูลซ้ำที่ไม่จำเป็นของ 3 แพ็กเกต

ในการณีที่ไม่สามารถเลือกตัวเลือก SACK ได้ duplicate acknowledgement จะไม่บรรจุข้อมูลที่เป็นการระบุข้อมูลของแพ็กเกตหรือแพ็กเกตทางฝั่งผู้รับที่ส่ง duplicate acknowledgement ในกรณีนี้ ผู้ส่งจะไม่สามารถแยกระหว่าง duplicate acknowledge ที่เป็นผลจากการสูญหายหรือ duplicate acknowledge ที่เกิดจากล่าช้าของแพ็กเกตกับ duplicate acknowledge ที่เกิดจากผู้ส่งส่งข้อมูลซ้ำแพ็กเกตที่ไม่จำเป็นที่ผู้รับได้รับแพ็กเกตชุดนี้ไว้เรียบร้อยแล้ว เพราะเหตุนี้ทำให้กระบวนการ fast retransmission และกระบวนการ fast recovery ใน TCP-Reno การสูญหายเซ็กเมนต์

หลายเซ็กแมนต์ในวินโดว์ชุดเดียวกัน บางครั้งจะทำให้เกิดกระบวนการ fast retransmit โดยไม่จำเป็น ส่งผลให้เกิดการลดลงของตัวแปร *cwnd*

กระบวนการ fast retransmission และกระบวนการ fast recovery ใน TCP-Reno ปัญหาทางด้านสมรรถนะมีสาเหตุมาจากกระบวนการ fast retransmit ของหลายแพ็กเกต แต่ก็ไม่เป็นปัญหาใหญ่เมื่อเปรียบเทียบกับ TCP-Tahoe ที่ไม่มีกระบวนการ fast recovery อย่างไรก็ตามการส่งข้อมูลซ้ำแพ็กเกตที่ไม่จำเป็นอาจเกิดขึ้นได้เช่นกัน หากไม่มีการแก้ไขอัลกอริธึมโดยเฉพาะ เช่น การใช้ตัวแปร *recover*

ใน RFC2582 นิยามให้ variant มาตราฐานของ TCP-Newreno กำหนดให้ไม่ให้ใช้ตัวแปร *recover* และไม่ให้ตรวจสอบ ถ้า duplicate acknowledge ครอบคลุมถึงตัวแปร *recover* ก่อนที่จะใช้กระบวนการ fast retransmit ด้วย variant มาตราฐาน ปัญหาของกระบวนการ fast retransmit ของหลายแพ็กเกตจากเพียงวินโดว์ชุดเดียวสามารถเกิดขึ้นหลังจากหมดคาบเวลาการส่งข้อมูลซ้ำหรือเกิดขึ้นในรูปแบบของการเรียงลำดับหมายเลขแพ็กเกต ซึ่งจะให้สมรรถนะเหมือนกับ less careful variant และ careful variant ของแบบแผนควบคุมความคับคั่ง TCP-Newreno ใน RFC2582 ซึ่งสามารถขจัดปัญหาของกระบวนการ fast retransmit หลายแพ็กเกตได้ อัลกอริธึมนี้นะใช้ตัวแปร *recover* โดยค่าเริ่มต้นคือค่าหมายเลขลำดับของการส่งเริ่มต้น หลังจากแต่ละหมดคาบเวลาการส่งข้อมูลซ้ำค่าหมายเลขลำดับสูงสุดจะถูกบันทึกให้เป็นค่าตัวแปร *recover*

ถ้าหลังจากหมดคาบเวลาการส่งข้อมูลซ้ำผู้ส่งจะส่ง 3 แพ็กเกตติดกันที่ได้รับแล้วทางฝั่งผู้รับ แล้วผู้ส่งจะได้รับ 3 duplicate acknowledgement ที่ไม่ครอบคลุมค่าตัวแปร *recover* ในกรณีนี้ duplicate acknowledge จะไม่ได้เป็องนเครื่องบ่งชี้การเกิดความคับคั่งใหม่ที่เกิดขึ้น มันเป็นเพียงเครื่องบ่งชี้ที่ผู้ส่งส่งข้อมูลซ้ำแพ็กเกตที่ไม่จำเป็นอย่างน้อย 3 แพ็กเกต

อย่างไรก็ตาม เมื่อแพ็กเกตที่เกิดจากส่งข้อมูลซ้ำสูญหายเอง ผู้ส่งก็ยังคงได้รับ 3 duplicate acknowledgement ที่ไม่ครอบคลุมค่าตัวแปร *recover* ในกรณีนี้ ผู้ส่งต้องหยุดกระบวนการถ้าอยู่ในช่วงเริ่มต้นของกระบวนการ fast retransmit สำหรับผู้ส่งที่สร้างด้วยอัลกอริธึมในข้อ 2.2.1.1.1 ถึงข้อ 2.2.1.1.5 ผู้ส่งจะไม่รู้ว่าเกิดการสูญหายของแพ็กเกตจาก duplicate acknowledgement และตัวจับเวลาในการส่งข้อมูลซ้ำจะเป็นวิธีการสำรองสำหรับการสูญหายของแพ็กเกต

มีวิธีการมากมายที่อยู่บนพื้นฐานของ timestamp หรืออยู่บนพื้นฐานของจำนวนของ acknowledgement สะสม ซึ่งจะทำให้ผู้ส่งสามารถแยกแยะระหว่าง 3 duplicate acknowledgement ตามกระบวนการส่งข้อมูลซ้ำแพ็กเกตที่สูญหายและ 3 duplicate acknowledgement จากการส่งข้อมูลซ้ำที่ไม่จำเป็นของ 3 แพ็กเกต ผู้ส่งอาจจะใช้วิธีการตัดสินใจว่าจะให้ใช้กระบวนการ fast retransmit ในบางกรณี แม้ 3 duplicate acknowledgement จะมีค่าไม่ครอบคลุมค่าตัวแปร *recover* ก็ตาม

ยกตัวอย่างเช่น เมื่อ 3 duplicate acknowledgement มีสาเหตุมาจากการส่งข้อมูลซ้ำที่ไม่จำเป็นของ 3 แพ็กเกต ซึ่งเหมือนกับการใช้ acknowledge สะสมโดยอย่างน้อย 4 เซ็กเมนต์ ในลักษณะที่คล้ายกัน วิธีการที่อยู่บนพื้นฐานของ timestamp ก็ใช้ความจริงที่ว่าเมื่อมีพื้นที่ว่างอยู่ในหมายเลขลำดับ timestamp จะอยู่ใน duplicate acknowledgement ซึ่งเป็น timestamp ของแพ็กเกตล่าสุดที่อยู่ในช่องของ acknowledgement สะสม ถ้า timestamp ถูกใช้ และผู้ส่งบันทึก timestamp ของเซ็กเมนต์ acknowledge สุดท้ายไว้ แล้ว timestamp ที่ถูกส่งโดย duplicate acknowledgement จะสามารถใช้แยกแยะระหว่างการส่งข้อมูลซ้ำแพ็กเกตที่สูญหายและ 3 duplicate acknowledgement จากการส่งข้อมูลซ้ำของ 3 แพ็กเกตที่ไม่จำเป็น วิธีการดังกล่าวได้แสดงไว้ใน NS Simulator ด้วยคำสั่ง "./test-all-newreno"

ก. วิธีการ ACK

ถ้าวิธีการที่อยู่บนพื้นฐานของ ACK ถูกใช้ แล้วผู้ส่งจะบันทึกค่าของ acknowledgement สะสมก่อนหน้านี้ให้เป็นตัวแปร *prev_highest_ack* และบันทึก acknowledgement สะสมหลังสุดให้เป็นตัวแปร *highest_ack* กระบวนการต่อไปนี้ให้กระทำถ้าข้อ 2.2.1.1.1 ล้มเหลวก่อนที่จะทำข้อ 2.2.1.1.1 ข.

ถ้าช่อง acknowledge สะสม ไม่ครอบคลุมค่าตัวแปร recover ให้ตรวจสอบว่า ค่า *cwnd* มีค่ามากกว่าไบต์ *SMSS* และความต่างระหว่างค่า *highest_ack* และค่า *prev_highest_ack* อยู่ที่ *4*SMSS* ไบต์ ถ้าจริง duplicate acknowledgement บ่งชี้เซ็กเมนต์ที่สูญหาย (ทำในข้อ 2.2.1.1.1 ก.) นอกจากนี้ duplicate acknowledgement คล้ายกับการส่งข้อมูลซ้ำโดยไม่จำเป็น (ทำในข้อ 2.2.1.1.1 ข.)

การตรวจสอบค่า *cwnd* สนับสนุนเพื่อป้องกันเข้าสู่กระบวนการ fast retransmit ทันที หลังจากหมดคาบเวลาการส่งข้อมูลซ้ำคล้ายกับตัวแปร *exitFastReturns* ใน NS Simulator โดยใช้คำสั่ง "./test-all-newreno newreno_rto_loss_ack" และ "./test-all-newreno newreno_rto_dup_ack"

ถ้ามีหลาย ACK สูญหาย ผู้ส่งสามารถเห็นการกระโดดใน acknowledgement สะสมมากกว่า 3 เซ็กเมนต์ และวิธีการอาจล้มเหลวได้ ใน NS Simulator ใช้คำสั่ง "./test-all-newreno newreno_rto_loss_ackf" ใน RFC2581 แนะนำให้ผู้รับควรส่ง duplicate acknowledgement สำหรับทุกๆแพ็กเกตที่เรียงลำดับไม่ถูกต้อง เช่นแพ็กเกตที่รับมาระหว่างกระบวนการ fast recovery วิธีการ ACK นี้ดูเหมือนจะล้มเหลวถ้าผู้รับไม่ทำตามคำแนะนำ เพราะหมายเลข ACK ที่สูญหายที่มีค่าน้อยกว่ามีความจำเป็นต่อการกระโดดใน ACK สะสม

ข. วิธีการ timestamp

ถ้าวิธีการนี้ถูกใช้ ผู้ส่งจะบันทึก timestamp ของเซ็กเมนต์ acknowledge สุดท้าย ย่อหน้าที่สองของข้อ 2.2.1.1.1 ให้แทนที่ด้วยกระบวนการดังต่อไปนี้

ถ้าช่อง acknowledge สะสมไม่ครอบคลุมตัวแปร recover ให้ตรวจสอบว่าถ้าการส่ง timestamp ใน non-duplicate acknowledgement มีค่าเท่ากับ timestamp ที่บันทึกไว้ ถ้าเป็นจริง duplicate acknowledgement จะบ่งชี้ถึงเซ็กเมนต์ที่สูญหาย (ให้ทำข้อ 2.2.1.1.1 ก.) หากไม่จริงให้ duplicate acknowledgement มีผลเหมือนกับการส่งข้อมูลซ้ำที่ไม่จำเป็น (ให้ทำข้อ 2.2.1.1.1 ข.)

ตัวอย่างของการใช้วิธีการ timestamp ใน NS Simulator ด้วยคำสั่ง "./test-allnewreno newreno_rto_loss_tsh" และ "./test-all-newreno newreno_rto_dup_tsh" วิธีการ timestamp ทำงานได้อย่างถูกต้อง ทั้งสองรูปแบบคือเมื่อผู้รับส่ง timestamp เหมือนที่ระบุไว้ใน RFC1323 และโดยดัดแปลงตัวเอง อย่างไรก็ตาม ถ้าผู้รับส่ง

timestamp โดยไม่มีกฎเกณฑ์ วิธีการนี้อาจล้มเหลวได้ หรือถ้าการหมดคาบเวลาไม่ได้เกิดขึ้นจริงและ ACK ที่กลับมาไม่ได้มาจากเซ็กเมนต์ที่เกิดจากส่งข้อมูลซ้ำก็สามารถทำให้วิธีการนี้ล้มเหลวได้เช่นกัน ซึ่งสามารถป้องกันได้ด้วยอัลกอริธึมการตรวจสอบ (detection algorithm) ใน RFC3522

2.2.1.2 การสร้างกระบวนการทางฝั่งผู้รับ

ใน RFC2581 ระบุไว้ว่าการเรียงลำดับเซ็กเมนต์ที่ไม่ถูกต้องต้องทำการ acknowledge ทันที เพื่อเร่งการ *recover* เมื่อเกิดการสูญหายของแพ็กเกต บางผู้รับห้ามส่ง ACK ทันที เมื่อได้ส่ง partial acknowledgement แต่จะรอให้ตัวจับเวลาของ delay acknowledgeเกิดการหมดคาบเวลาแทน ซึ่งจะทำให้เกิดข้อจำกัดที่รุนแรงต่อสมรรถนะของ TCP-Newreno โดยทำให้เกิดความล่าช้าของการรับ partial acknowledge ดังนั้นทางฝั่งผู้รับต้องส่ง acknowledge ทันทีไปยังผู้ส่งสำหรับเซ็กเมนต์ที่เรียงลำดับไม่ถูกต้อง แม้เซ็ก-เมนต์นั้นจะอยู่ในบัฟเฟอร์ก็ตาม

2.2.1.3 การสร้างกระบวนการทางฝั่งผู้ส่ง

ในข้อ 2.2.1.1.5 กระบวนการในขั้นตอนนี้ควรมีการวัดเพื่อหลีกเลี่ยงชุดข้อมูลมาติดๆกันเมื่อออกจากกระบวนการ fast recovery ในกรณีจำนวนของข้อมูลชุดใหม่ที่ผู้ส่งเลือกที่จะส่งออกไปตามค่าตัวแปร *cwnd* ที่มีค่ามากขึ้น เหตุการณ์สามารถเกิดขึ้นได้ในแบบแผนควบคุมความคับคั่ง TCP-Newreno เมื่อ ACK เกิดสูญหายหรือพิจารณาให้เป็นวินโดว์ของการปรับข้อมูลเพียงอย่างเดียว ดังนั้นจึงเป็นสาเหตุให้ผู้ส่งประเมินหาจำนวนเซ็กเมนต์ชุดใหม่ที่ถูกส่งระหว่างกระบวนการ recovery โดยเฉพาะอย่างยิ่ง ข้อมูลที่มาติดๆกันสามารถเกิดขึ้นได้เมื่อตัวแปร *FlightSize* มีค่าน้อยกว่าตัวแปร *cwnd* มากๆ เมื่อออกจากกระบวนการ fast recovery หนึ่งในกระบวนการที่สามารถหลีกเลี่ยงข้อมูลที่มาติดๆกันเมื่อออกจากกระบวนการ fast recovery คือต้องจำกัดจำนวนแพ็กเกตที่ถูกส่งไปเพื่อตอบสนองต่อหนึ่ง ACK อีกวิธีการหนึ่งที่สามารถหลีกเลี่ยงชุดข้อมูลที่มาติดๆกันได้คือกระบวนการ rate-based pacing หรือการกำหนดค่าตัวแปร *ssthresh* ให้มีค่าเท่ากับผลลัพธ์ของตัวแปร *cwnd* และตั้งค่าใหม่ให้กับตัวแปร *cwnd* ให้มีค่าเท่ากับตัวแปร *FlightSize*

ในกระบวนการสร้างอาจจำเป็นต้องใช้ flag ที่แยกออกมาต่างหาก เพื่อที่จะระบุให้ได้ว่า ณ ขณะนี้อยู่ในกระบวนการ fast recovery หรือไม่ การใช้ค่าของตัวนับเวลาของ duplicate acknowledgement เพื่อจุดประสงค์นี้ไม่น่าเชื่อถือเพราะมันสามารถเป็นการตั้งค่าใหม่ของวินโดว์และการเรียงลำดับไม่ถูกต้องของ duplicate acknowledgement เมื่อไม่อยู่ในกระบวนการ fast recovery ค่าของตัวแปร recover ควรแทนที่ด้วยตัวแปร *snd_una* ซึ่งจะทำให้ เมื่อมีข้อมูลจำนวนมากถูกส่งไปและ acknowledge หมายเลขลำดับจะไม่บ่งชี้ผิดค่าอันจะทำให้เข้าสู่กระบวนการ fast recovery โดยไม่จำเป็น

มันเป็นสิ่งสำคัญสำหรับผู้ส่งที่ต้องตอบสนองต่อการรับ duplicate acknowledgement อย่างถูกต้องเมื่อผู้ส่งไม่ได้อยู่ในกระบวนการ fast recovery อีกแล้ว (เพราะมีสาเหตุมาจากหมดคาบเวลาการส่งข้อมูลซ้ำ) กระบวนการการจำกัดการส่งซึ่งอธิบายไว้ใน RFC3042 จะอธิบายถึงการตองสนองที่เป็นไปได้ต่อ duplicate acknowledgement อันแรก หรือ duplicate acknowledgement อันที่สอง เมื่อได้รับ 3 duplicate acknowledgement ช่อง acknowledgement สะสมจะไม่ครอบคลุมค่าตัวแปร *recover* และกระบวนการ fast recovery จะไม่มีส่วนเกี่ยวข้อง ซึ่งเป็นสิ่งสำคัญที่ผู้ส่งจะไม่กระทำกระบวนการในข้อ 2.2.1.1.3 และ ข้อ 2.2.1.1.4 นอกจากนี้ผู้ส่งควรจะสามารถกำจัดการการหมดคาบเวลาที่ไม่เป็นจริงได้ สิ่งนี้เป็นส่างที่ควรนำมาพิจารณาเป็นพิเศษเพราะการสร้างแบบแผนควบคุมความคับคั่ง TCP-Newreno หลายครั้งจะเกิดข้อผิดพลาดนี้ขึ้นเสมอ รวมไปถึงการจำลองใน NS Simulator ด้วย ซึ่งใน NS Simulator มีการแก้ข้อผิดพลาดนี้ด้วยตัวแปร *exitFastReturns*

2.2.1.4 ความแตกต่างระหว่างแบบแผนควบคุมความคับคั่ง TCP-Reno และ TCP-Newreno

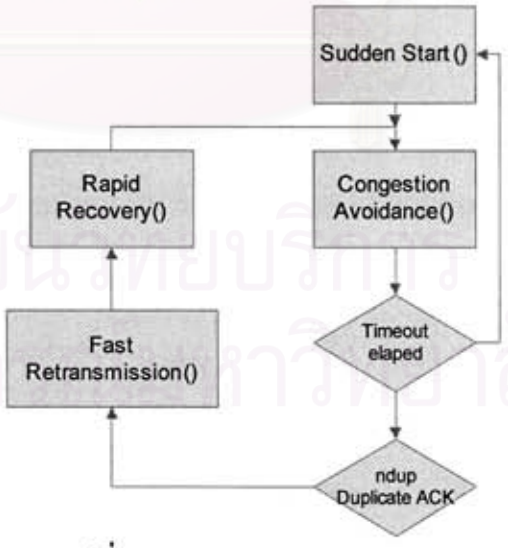
TCP-Newreno มีพื้นฐานมาจาก TCP-Reno โดยได้พัฒนากระบวนการ fast retransmit และกระบวนการ fast recovery ให้มีประสิทธิภาพดีขึ้นในหลายรูปแบบของการติดต่อ TCP TCP-Reno จะมีสมรรถนะที่ไม่ดีเมื่อมีแพ็กเกตหลายแพ็กเกตเกิดการสูญหายไปจากวินโดว์และได้แสดงว่า TCP-Newreno มีสมรรถนะที่ดีในการติดต่อ TCP ในรูปแบบต่างๆ แต่ก็มีในบางรูปแบบที่ TCP-Rreno มีสมรรถนะที่ดีกว่า TCP-Newreno คือเมื่อการส่งข้อมูลระหว่างผู้ส่งกับผู้รับไม่มีแพ็กเกตสูยหายเลยแต่อาจมีการเรียงลำดับไม่ถูกต้อง ดังนั้นผู้ส่งอาจได้รับ 3 duplicate acknowledgement ซึ่งจะทำให้เข้าสู่กระบวนการ fast retransmit หรือ fast recovery ด้วยแบบแผนควบคุมความคับคั่ง TCP-Reno จะพิจารณาให้เป็นการส่ง

ข้อมูลที่ไม่จำเป็นของการส่งแพ็กเกตเพียงแพ็กเกตเดียว แต่เมื่อพิจารณาในแบบแผนควบคุมความคับคั่ง TCP-Newreno ก็ยังพิจารณาให้เป็นการส่งข้อมูลที่ไม่จำเป็นเหมือนกันแต่จะเป็นการส่งของทั้ง วินโดว์

ในขณะที่ TCP-reno มีสมรรถนะที่ดีกว่า TCP-Newreno เมื่อมีการเรียงลำดับข้อมูลที่ไม่ถูกต้อง อย่างไรก็ตาม TCP-Newreno ก็มีสมรรถนะที่ดีกว่ามากในกรณีที่เกิดการสูญหายของแพ็กเกตหลายแพ็กเกตและมีน้ำหนักมากกว่าข้อดีของ TCP-Reno (การเลือกตัวเลือก SACK จะมีสมรรถนะที่ดีในทั้งสองรูปแบบที่กล่าวมา) สรุปได้ว่าเมื่อไม่สามารถเลือกตัวเลือก SACK ได้ ให้เลือกกระบวนการ fast retransmit และ fast recovery ของ TCP-Newreno ปัจจุบันในการสื่อสารผ่านอินเตอร์เน็ทแบบแผนควบคุมความคับคั่งที่ใช้กันอย่างแพร่หลายมากที่สุดคือ TCP-Newreno

2.2.2 TCP-Peach [10]

TCP-Peach ถูกออกแบบมาสำหรับโครงข่ายดาวเทียม TCP-Peach มีกระบวนการย่อย 4 กระบวนการ คือ sudden start, congestion avoidance, fast retransmission และ rapid recovery โดยกระบวนการ congestion avoidance และ fast retransmission ยังเหมือนอัลกอริทึม TCP-RENO แต่ได้ปรับปรุง slow start เป็น sudden start และ fast recovery เป็น rapid recovery มีอัลกอริทึมแสดงไว้ในรูปที่ 2.4 และพื้นฐานกระบวนการจะใช้เซ็กเมนต์จำลองเป็นหลัก (dummy segment) ซึ่งเป็นเซ็กเมนต์ที่ไม่มีข้อมูลจริง โดยมีรายละเอียดดังนี้



รูปที่ 2.4 แบบแผนของ TCP-Peach [10]

2.2.2.1 Dummy Segment

Dummy segment เป็นเซ็กเมนต์ที่มีความสำคัญต่ำในชั้น IP และ เป็นเซ็กเมนต์ที่ไม่มีข้อมูลบรรจุอยู่ภายใน ถูกสร้างขึ้นโดยผู้ส่ง ทำหน้าที่ประเมินหาทรัพยากรในโครงข่าย ถ้าในโครงข่ายเกิดความคับคั่งขึ้น แพ็กเกตในชั้น IP ที่บรรจุเซ็กเมนต์จำลองจะถูกขจัดออกไปเป็นลำดับแรก การขจัดเซ็กเมนต์จำลองจะไม่ทำให้ throughput ทางฝั่งผู้รับลดลง เพราะ เซ็กเมนข้อมูลต์ไม่ได้ถูกขจัดออกไป ถ้าในโครงข่ายไม่มีความคับคั่งเซ็กเมนต์จำลองสามารถส่งไปถึงผู้รับ ผู้รับสามารถแยกเซ็กเมนต์ข้อมูลและเซ็กเมนต์จำลองออกจากกันได้ เพราะ ผู้ส่งได้กำหนด TCP header ในบิตที่ยังไม่ได้ใช้งาน 1-6 บิตให้แยกเซ็กเมนต์ข้อมูลและเซ็กเมนต์จำลองในส่วนของ ACK ก็เช่นเดียวกันผู้รับก็จะกำหนดบิตที่ยังไม่ได้ใช้งาน 1-6 บิตให้แยกเซ็กเมนต์ข้อมูลและเซ็กเมนต์จำลองเมื่อผู้ส่งได้ ACK ของเซ็กเมนต์จำลองตอบกลับมา ผู้ส่งจะสามารถรู้ได้ว่าในโครงข่ายยังมีทรัพยากรเหลือที่ยังไม่ได้นำไปใช้ ทำให้ผู้ส่งสามารถเพิ่มอัตราการส่งข้อมูลขึ้นได้

ACK ของเซ็กเมนต์จำลองจะถูกส่งในช่วง sudden start และ rapid recovery และจะรับในช่วง congestion avoidance ทำให้ในอัลกอริทึม congestion avoidance จำเป็นต้องแก้ไขเล็กน้อย

ตัวแปร *wdsn* ถูกนิยามขึ้น และจะมีค่าเป็นศูนย์เมื่อเริ่มต้นการติดต่อด้วย TCP นำไปใช้เมื่อผู้ส่งได้ ACK ของเซ็กเมนต์จำลองผู้ส่งจะตรวจค่า wdsn ดังนี้

- ถ้า $wdsn = 0$ ผู้ส่งจะเพิ่มค่า *cwnd* อีก 1 เซ็กเมนต์
- ถ้า $wdsn \neq 0$ ผู้ส่งจะลดค่า *wdsn* ทีละ 1 เซ็กเมนต์และให้ค่า *cwnd* มีค่าเท่าเดิม

2.2.2.2 Sudden Start

Sudden Start ถูกออกแบบให้แทนที่ slow start เดิม Sudden Start จะเริ่มต้นด้วยการส่ง *cwnd* ขนาด 1 เซ็กเมนต์หลังจากนั้น จะส่งด้วย *cwnd* ที่มีขนาดเท่ากับ *rwnd-1* (*rwnd* คือ advertised window) ทุกคาบเวลาที่มีค่าตามสมการที่ 3

$$\tau = RTT / rwnd \qquad \text{สมการที่ 3}$$

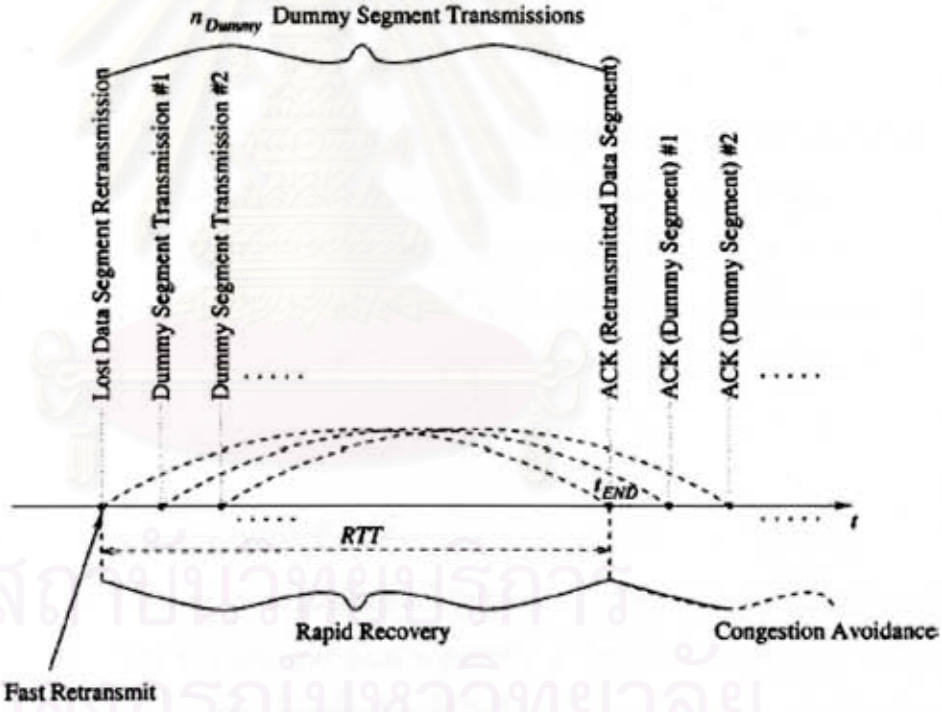
ผลที่ได้จะทำให้ หลังจากเวลาในการส่งผ่านไปหนึ่ง *RTT* จะทำให้ *cwnd* เพิ่มขึ้นอย่างรวดเร็ว ผู้รับสามารถประมาณหาค่า *RTT* ได้ในช่วงสร้างการติดต่อ

2.2.2.3 Rapid Recovery

Rapid Recovery ถูกออกแบบมาให้แทนที่ fast recovery มีจุดประสงค์เพื่อแก้ปัญหาของแพ็กเกตที่เสียหายระหว่างทาง ที่กล่าวไว้ในหัวข้อย่อยที่ 2.3.2.2

จากรูปที่ 2.3 เมื่อแพ็กเกตเกิดสูญหายและสามารถตรวจสอบได้โดย *ndup* (duplicate ACK) กระบวนการ fast recovery จะถูกนำมาใช้ หลังจากนั้น จะเข้าสู่กระบวนการ rapid recovery และสิ้นสุดเมื่อถึงเวลา t_{end} ประมาณ 1 *RTT* หลังจากทำการส่งข้อมูลซ้ำ แสดงไว้ในรูปที่ 2.5 เมื่อได้รับ ACK ของเซ็กเมนต์ข้อมูล กระบวนการจะเข้าสู่ congestion avoidance

Rapid recovery ยังคงใช้สมมุติฐานเดิมเดียวกันกับ fast recovery ว่า เมื่อเกิดการสูญหายของแพ็กเกต จะมีสาเหตุมาจากความคับคั่งของโครงข่าย เพราะ TCP ไม่สามารถรู้สาเหตุที่แท้จริงได้ ทำให้ผู้ส่งลด *cwnd* ลงครึ่งหนึ่งเหมือนกับ TCP-Reno ถ้า congestion window ขณะนั้นมีค่า $cwnd_0$ ดังนั้นในระหว่าง rapid recovery จะส่งข้อมูลด้วย $cwnd_0/2$



รูปที่ 2.5 กระบวนการใน Rapid Recovery [10]

ในการประเมินค่าแบนด์วิดท์ของช่องสัญญาณ ผู้ส่งจะส่งเซ็กเมนต์จำลองจำนวน n_{dummy} เมื่อได้รับ ACK ของ เซ็กเมนต์ข้อมูลแล้ว ACK ของเซ็กเมนต์จำลองจะตามมา ซึ่งเกิดขึ้นในกระบวนการ congestion avoidance แสดงไว้ในรูปที่ 2.5 ถ้าการสูญหายของแพ็กเกตเกิดจากความคับคั่งของโครงข่าย โครงข่ายสามารถบรรจุ congestion window ได้เท่ากับ $cwnd_0$ ต่อหนึ่ง *RTT* rapid recovery จะส่งเซ็กเมนต์ข้อมูลเท่ากับ $cwnd_0/2$ และส่งเซ็กเมนต์จำลอง เท่ากับ $cwnd_0/2$ เมื่อ ACK ของเซ็กเมนต์จำลองตัวแรกกลับมา ผู้ส่งจะยังคงไม่เพิ่ม *cwnd* เพราะการได้รับ ACK ของเซ็กเมนต์จำลองไม่สามารถแปลความได้ว่า การสูญหายของแพ็กเกตจะมีสาเหตุมาจากความเสียหายของแพ็กเกตระหว่างทาง แต่จะกำหนดค่า $wdsn = cwnd_0/2$ และหลังจากได้รับ ACK ของเซ็กเมนต์จำลองเท่ากับ $cwnd_0/2$ แล้ว ผู้ส่งจะเพิ่มค่า *cwnd* ทีละหนึ่งเซ็กเมนต์ทุกครั้งที่ได้รับ ACK ของเซ็กเมนต์จำลอง

ขั้นต่อไปให้ n_{dummy} มีค่าเท่ากับ *cwnd* ถ้า ACK ของเซ็กเมนต์จำลองกลับมาทุกเซ็กเมนต์ ดังนั้น congestion window จะมีค่า *cwnd* เท่ากับ $cwnd_0$ (เท่าค่าเดิมก่อนเกิดการสูญหายของแพ็กเกต)

2.2.3 TCP-Vegas [13]

TCP-Vegas ถูกออกแบบมาเพื่อเพิ่มค่า throughput และลดการสูญหายของแพ็กเกต โดยปรับปรุงกระบวนการ 3 กระบวนการ อันดับแรกคือกระบวนการส่งข้อมูลซ้ำ โดยพัฒนาหาเวลาการสูญหายของแพ็กเกตให้แม่นยำมากขึ้น อันดับสองคือกระบวนการ congestion avoidance ให้สามารถหลีกเลี่ยงความคับคั่ง โดยปรับปรุงอัตราการส่งข้อมูลเข้าสู่โครงข่าย และอันดับสุดท้ายคือกระบวนการ slow start โดยพยายามจะหลีกเลี่ยงการสูญหายของแพ็กเกตในขณะที่ประเมินหาแบนด์วิดท์

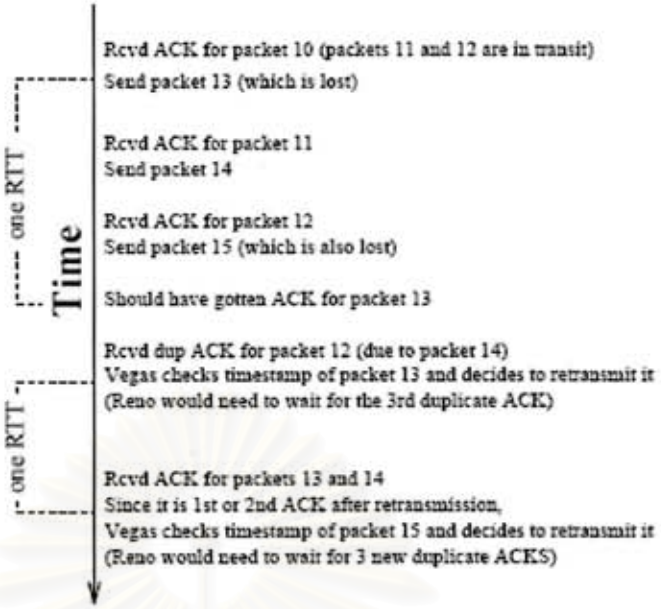
2.2.3.1 การปรับปรุงกระบวนการส่งข้อมูลซ้ำ

ใน TCP-Newreno การส่งข้อมูลซ้ำแพ็กเกตจะเกิดจากสองสาเหตุคือ เมื่อเกิดการหมดคาบเวลาหรือได้รับ n duplicated ACK (โดยปกติ n จะมีค่าเท่ากับ 3) การที่ TCP-Newreno จะส่ง duplicated ACK ก็ต่อเมื่อ ผู้รับได้รับข้อมูลที่ไม่ถูกต้อง ตัวอย่างเช่นเมื่อผู้รับได้รับแพ็กเกตหมายเลข 2 แต่แพ็กเกตหมายเลขเกิดสูญหายไป ผู้รับจึงได้รับแพ็กเกตต่อไปเป็นหมายเลข 4 แทน ทำให้ผู้รับต้องส่ง duplicated ACK สำหรับหมายเลข 2 ออก หมายถึงได้รับแพ็กเกตหมายเลข 2 แล้วให้ส่งแพ็กเกตหมายเลข 3 มา เมื่อผู้รับได้แพ็กเกตหมายเลข 5

ก็จะส่ง duplicated ACK ออกไปอีกเป็น duplicated ACK ที่สอง จนครบสาม duplicated ACK ผู้ส่งจึงส่งข้อมูลซ้ำแพ็กเกตหมายเลข 3 กลับไปใหม่ เมื่อคำนวณเวลาแล้วในบางโครงข่ายเวลาที่ใช้ไปทั้งหมดอาจมีค่ามากกว่าเวลาการหมดคาบเวลาซึ่งโดยปกติจะมีค่าประมาณ 300 ms TCP-Vegas จึงออกแบบมาเพื่อลดเวลาในการส่งข้อมูลซ้ำให้น้อยลงโดยทำการบันทึกเวลาทุกครั้งที่มีการส่งแพ็กเกตออกไปและเมื่อได้รับ ACK TCP-Vegas จะทำการคำนวณค่า RTT และจะตัดสินใจให้มีการส่งข้อมูลซ้ำแพ็กเกต แสดงตัวอย่างในรูปที่ 2.6 โดยการตัดสินจะมีสถานการณ์สองสถานการณ์ดังต่อไปนี้

2.2.3.1.1 เมื่อได้รับ duplicated ACK TCP-Vegas จะทำการตรวจสอบระหว่างเวลาปัจจุบันกับเวลาที่บันทึกไว้กับแพ็กเกตที่เกี่ยวข้องว่ามีค่ามากกว่าเวลาการหมดคาบเวลาหรือไม่ หากเป็นจริง TCP-Vegas ก็จะส่งข้อมูลซ้ำแพ็กเกตนั้นออกไปโดยไม่รอให้ได้รับ n duplicated ACK

2.2.3.1.2 เมื่อผู้ส่งได้รับ ACK ที่หนึ่งและที่สองหลังการได้ทำการส่งข้อมูลซ้ำ TCP-Vegas ได้ตรวจสอบค่าช่วงเวลาระหว่าง ACK ที่หนึ่งกับ ACK ที่สอง หากค่าช่วงเวลาที่คำนวณได้มีค่ามากกว่าช่วงการหมดคาบเวลา TCP-Vegas จะทำการส่งข้อมูลซ้ำแพ็กเกตนั้นทันที โดยไม่รอให้ได้รับ duplicated ACK



รูปที่ 2.6 ตัวอย่างของกระบวนการส่งข้อมูลซ้ำใน TCP-Vegas[13]

2.2.3.2 การปรับปรุงกระบวนการ congestion avoidance

ใน TCP-Newreno จะพิจารณาการสูญหายของแพ็กเกตให้มีสาเหตุมาจากการความคับคั่งของโครงข่ายเพียงอย่างเดียว ซึ่งแท้จริงแล้วอาจเกิดจากความเสียหายของแพ็กเกตเองก้ได้ ซึ่งถ้าเกิดจากความเสียหายของแพ็กเกตเองแล้ว กระบวนการไม่จำเป็นต้องลดค่า *cwnd* TCP-Vegas ได้ออกแบบให้สามารถหาสาเหตุที่แท้จริงของการสูญหายของแพ็กเกตได้โดยอาศัยการคำนวณค่าความแตกต่างระหว่างค่า throughput จากการประเมิน (Expected Throughput) กับค่า throughput ที่วัดได้จริง โดยค่า Expected Throughput คำนวณได้ตามสมการที่ 4 และให้ค่าความต่าง (*Diff*) มีค่าตามสมการที่ 5

$$Expected = WindowSize / BassRTT \qquad \text{สมการที่4}$$

$$Diff = Expected - Actual \qquad \text{สมการที่5}$$

โดยกำหนดให้ *WindowSize* คือ ค่าขนาดของ *cwnd* ที่ส่งล่าสุด และค่า *BassRTT* คือ ค่า *RTT* ที่มีค่าน้อยที่สุดที่สามารถวัดได้

โดยนิยามค่า threshold ขึ้นมาสองค่าคือค่า α เป็นขอบล่าง และ β เป็นขอบบน มีหลักการอยู่ว่า หากค่า *Diff* มีค่าน้อยกว่า α ให้เพิ่มค่า *cwnd* หากมีค่ามากกว่า β ให้ลดค่า *cwnd* และหากอยู่ระหว่างค่า α และ β ให้คงค่า *cwnd* ไว้

2.2.3.3 การปรับปรุงกระบวนการ slow start

ในกระบวนการ slow start ของ TCP-Newreno จะมีการเพิ่มค่า *cwnd* เป็นสองเท่าทุกๆ RTT ที่ได้รับ ACK การเพิ่มขึ้นของค่า *cwnd* จะเพิ่มขึ้นเรื่อยๆจนทำให้โครงข่ายเกิดความคับคั่งหรือถึงค่า *ssthresh* จึงจะลดค่า *cwnd* ลงครึ่งหนึ่ง TCP-Vegas จะปรับปรุงโดยการประเมินหาแบนด์วิดท์ที่เหลือของระบบเพื่อให้ในช่วงสุดท้ายของกระบวนการ slow start ไม่เพิ่มค่า *cwnd* จนทำให้เกิดความคับคั่งขึ้น การปรับปรุงกระบวนการ slow start จะอาศัยหลักการการประเมินแบนด์วิดท์เหมือนกับในกระบวนการ congestion avoidance ที่กล่าวไว้ในหัวข้อ 2.3.2 โดยมีการดัดแปลงบางส่วน การเพิ่มของ *cwnd* จะเพิ่มแบบ exponential ทุกๆที่คาบเวลาไปกลับ แต่ในระหว่างที่ค่า *cwnd* มีค่าคงที่ จะมีการเปรียบเทียบค่าตัวแปร *Expected* กับค่า *Actual* หากค่าตัวแปร *Expected* มีค่ามากกว่าตัวแปร *Actual* เท่ากับ threshold γ จะให้การเพิ่มค่า *cwnd* เป็นแบบเชิงเส้น

2.2.3 อัลกอริธึมของ TCP-Vegas

อัลกอริธึมของ TCP-Vegas แสดงด้วยตารางที่ 2.2

ตารางที่ 2.2 โปรแกรมภาษาซีของ TCP-Vegas

```
#ifndef lint
static const char rcsid[] =
"@(#)   $Header:   /nfs/jade/vint/CVSROOT/ns-2/tcp/tcp-vegas.cc,v   1.35
2000/12/14 05:05:21 xuanc Exp $ (NCSU/IBM)";
#endif

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

#include "ip.h"
#include "tcp.h"
#include "flags.h"

#define MIN(x, y) ((x)<(y) ? (x) : (y))


static class VegasTcpClass : public TclClass {
public:
      VegasTcpClass() : TclClass("Agent/TCP/Vegas") {}
      TclObject* create(int, const char*const*) {
            return (new VegasTcpAgent());
      }
} class_vegas;


VegasTcpAgent::VegasTcpAgent() : TcpAgent()
{
      v_sendtime_ = NULL;
      v_transmits_ = NULL;
}

VegasTcpAgent::~VegasTcpAgent()
{
      if (v_sendtime_)
            delete []v_sendtime_;
      if (v_transmits_)
            delete []v_transmits_;
}

void
VegasTcpAgent::delay_bind_init_all()
{
      delay_bind_init_one("v_alpha_");
      delay_bind_init_one("v_beta_");
      delay_bind_init_one("v_gamma_");
      delay_bind_init_one("v_rtt_");
      TcpAgent::delay_bind_init_all();
        reset();
}

int
VegasTcpAgent::delay_bind_dispatch(const   char   *varName,   const   char
*localName,
                                   TclObject *tracer)
```

```
{
      /* init vegas var */
        if   (delay_bind(varName,   localName,   "v_alpha_",   &v_alpha_,
tracer))
            return TCL_OK;
        if    (delay_bind(varName,    localName,    "v_beta_",    &v_beta_,
tracer))
            return TCL_OK;
        if   (delay_bind(varName,   localName,   "v_gamma_",   &v_gamma_,
tracer))
            return TCL_OK;
        if (delay_bind(varName, localName, "v_rtt_", &v_rtt_, tracer))
            return TCL_OK;
        return     TcpAgent::delay_bind_dispatch(varName,     localName,
tracer);
}

void
VegasTcpAgent::reset()
{
      t_cwnd_changed_ = 0.;
      firstrecv_ = -1.0;
      v_slowstart_ = 2;
      v_sa_ = 0;
      v_sd_ = 0;
      v_timeout_ = 1000.;
      v_worried_ = 0;
      v_begseq_ = 0;
      v_begtime_ = 0.;
      v_cntRTT_ = 0; v_sumRTT_ = 0.;
      v_baseRTT_ = 1000000000.;
      v_incr_ = 0;
      v_inc_flag_ = 1;

      TcpAgent::reset();
}

void
VegasTcpAgent::recv_newack_helper(Packet *pkt)
{
      newack(pkt);
#if 0
      // like TcpAgent::recv_newack_helper, but without this
      if ( !hdr_flags::access(pkt)->ecnecho() || !ecn_ ) {
            opencwnd();
      }
#endif
      /* if the connection is done, call finish() */
      if ((highest_ack_ >= curseq_-1) && !closed_) {
            closed_ = 1;
            finish();
      }
}

void
VegasTcpAgent::recv(Packet *pkt, Handler *)
{
```

```
        double currentTime = vegastime();
        hdr_tcp *tcph = hdr_tcp::access(pkt);
        hdr_flags *flagh = hdr_flags::access(pkt);

#if 0
        if (pkt->type_ != PT_ACK) {
                Tcl::instance().evalf("%s error \"recieved non-ack\"",
                                      name());
                Packet::free(pkt);
                return;
        }
#endif /* 0 */
        ++nackpack_;

        if(firstrecv_<0) { // init vegas rtt vars
                firstrecv_ = currentTime;
                v_baseRTT_ = v_rtt_ = firstrecv_;
                v_sa_   = v_rtt_ * 8.;
                v_sd_   = v_rtt_;
                v_timeout_ = ((v_sa_/4.)+v_sd_)/2.;
        }

        if (flagh->ecnecho())
                ecn(tcph->seqno());
        if (tcph->seqno() > last_ack_) {
                if (last_ack_ == 0 && delay_growth_) {
                        cwnd_ = initial_วินโดว์();
                }
                /* check if cwnd has been inflated */
                if(dupacks_ > numdupacks_ &&  cwnd_ > v_newcwnd_) {
                        cwnd_ = v_newcwnd_;
                        // vegas ssthresh is used only during slow-start
                        ssthresh_ = 2;
                }
                int oldack = last_ack_;

                recv_newack_helper(pkt);

                /*
                 * begin of once per-rtt actions
                 *    1. update path fine-grained rtt and baseRTT
                 *      2. decide what to do with cwnd_, inc/dec/unchanged
                 *         based on delta=expect - actual.
                 */
                if(tcph->seqno() >= v_begseq_) {
                        double rtt;
                        if(v_cntRTT_ > 0)
                                rtt = v_sumRTT_ / v_cntRTT_;
                        else
                                rtt = currentTime - v_begtime_;

                        v_sumRTT_ = 0.0;
                        v_cntRTT_ = 0;

                        // calc # of packets in transit
                        int rttLen = t_seqno_ - v_begseq_;
```

```
                /*
                 * decide should we incr/decr cwnd_ by how much
                 */
                if(rtt>0) {
                        /* if there's only one pkt in transit, update
                         * baseRTT
                         */
                        if(rtt<v_baseRTT_ || rttLen<=1)
                                v_baseRTT_ = rtt;

                        double expect;    // in pkt/sec
                        // actual = (# in transit)/(current rtt)
                        v_actual_ = double(rttLen)/rtt;
                        // expect = (current วินโดว์ size)/baseRTT
                        expect = double(t_seqno_-last_ack_)/v_baseRTT_;

                        // calc actual and expect thruput diff, delta
                        int                             delta=int((expect-
v_actual_)*v_baseRTT_+0.5);
                        if(cwnd_ < ssthresh_) { // slow-start
                                // adj cwnd every other rtt
                                v_inc_flag_ = !v_inc_flag_;
                                if(!v_inc_flag_)
                                        v_incr_ = 0;
                                else {
                                    if(delta > v_gamma_) {
                                      // slow-down a bit to ensure
                                      // the net is not so congested
                                      ssthresh_ = 2;
                                      cwnd_-=(cwnd_/8);
                                      if(cwnd_<2)
                                            cwnd_ = 2.;
                                      v_incr_ = 0;
                                    } else
                                      v_incr_ = 1;
                                }
                        } else { // congestion avoidance
                                if(delta>v_beta_) {
                                        /*
                                         * slow down a bit, retrack
                                         * back to prev. rtt's cwnd
                                         * and dont incr in the nxt rtt
                                         */
                                        --cwnd_;
                                        if(cwnd_<2) cwnd_ = 2;
                                        v_incr_ = 0;
                                } else if(delta<v_alpha_)
                                        // delta<alpha, faster....
                                        v_incr_ = 1/cwnd_;
                                else // current rate is cool.
                                        v_incr_ = 0;
                        }
                } // end of if(rtt > 0)

                // tag the next packet
```

```
        v_begseq_ = t_seqno_;
        v_begtime_ = currentTime;
    } // end of once per-rtt section

    /* since we set how much to incr only once per rtt,
     * need to check if we surpass ssthresh during slow-start
     * before the rtt is over.
     */
    if(v_incr_ == 1 && cwnd_ >= ssthresh_)
        v_incr_ = 0;

    /*
     * incr cwnd unless we havent been able to keep up with it
     */
    if(v_incr_>0 && (cwnd_-(t_seqno_-last_ack_))<=2)
        cwnd_ = cwnd_+v_incr_;

    // Add to make Vegas obey maximum congestion วินโดว์ variable.
    if (maxcwnd_ && (int(cwnd_) > maxcwnd_)) {
        cwnd_ = maxcwnd_;
    }

    /*
     * See if we need to update the fine grained timeout value,
     * v_timeout_
     */

    // reset v_sendtime for acked pkts and incr v_transmits_
    double sendTime = v_sendtime_[tcph->seqno()%v_maxwnd_];
    int transmits = v_transmits_[tcph->seqno()% v_maxwnd_];
    int range = tcph->seqno() - oldack;
    for(int k=((oldack+1) %v_maxwnd_); \
        k<=(tcph->seqno()%v_maxwnd_) && range >0 ; \
        k=((++k) % v_maxwnd_), range--) {
        v_sendtime_[k] = -1.0;
        v_transmits_[k] = 0;
    }

    if((sendTime !=0.) && (transmits==1)) {
         // update fine-grained timeout value, v_timeout_.
        double rtt, n;
        rtt = currentTime - sendTime;
        v_sumRTT_ += rtt;
        ++v_cntRTT_;
        if(rtt>0) {
            v_rtt_ = rtt;
            if(v_rtt_ < v_baseRTT_)
                v_baseRTT_ = v_rtt_;
            n = v_rtt_ - v_sa_/8;
            v_sa_ += n;
            n = n<0 ? -n : n;
            n -= v_sd_ / 4;
            v_sd_ += n;
            v_timeout_ = ((v_sa_/4)+v_sd_)/2;
            v_timeout_ += (v_timeout_/16);
        }
```

```
            }

            /*
             * check the 1st or 2nd acks after dup ack received
             */
            if(v_worried_>0) {
                  /*
                   * check if any pkt has been timeout. if so,
                   * retx it. no need to change cwnd since we
                   * already did.
                   */
                  --v_worried_;
                  int expired=vegas_expire(pkt);
                  if(expired>=0) {
                        dupacks_ = numdupacks_;
                        output(expired, TCP_REASON_DUPACK);
                  } else
                        v_worried_ = 0;
            }
      } else if (tcph->seqno() == last_ack_)  {
            /* check if a timeout should happen */
            ++dupacks_;
            int expired=vegas_expire(pkt);
            if (expired>=0 || dupacks_ == numdupacks_) {
                  double      sendTime=v_sendtime_[(last_ack_+1)      %
v_maxwnd_];
                  int       transmits=v_transmits_[(last_ack_+1)      %
v_maxwnd_];
                        /*  The  line  below,  for  "bug_fix_"  true,
avoids
                        * problems with multiple fast retransmits after
                  * a retransmit timeout.
                        */
                  if ( !bug_fix_ || (highest_ack_ > recover_) || \
                      ( last_cwnd_action_ != CWND_ACTION_TIMEOUT)) {
                        int win = วินโดว์();
                        last_cwnd_action_ = CWND_ACTION_DUPACK;
                        recover_ = maxseq_;
                        /* check for timeout after recv a new ack */
                        v_worried_ = MIN(2, t_seqno_ - last_ack_ );

                        /* v_rto expon. backoff */
                        if(transmits > 1)
                              v_timeout_ *=2.;
                        else
                              v_timeout_ += (v_timeout_/8.);
                        /*
                         * if cwnd hasnt changed since the pkt was sent
                         * we need to decr it.
                         */
                        if(t_cwnd_changed_ < sendTime ) {
                              if(win<=3)
                                    win=2;
                              else if(transmits > 1)
                                    win >>=1;
                              else
```

```
                                        win -= (win>>2);

                                        // record cwnd_
                                        v_newcwnd_ = double(win);
                                        // inflate cwnd_
                                        cwnd_ = v_newcwnd_ + dupacks_;
                                        t_cwnd_changed_ = currentTime;
                                }

                                // update coarser grained rto
                                reset_rtx_timer(1);
                                if(expired>=0)
                                        output(expired, TCP_REASON_DUPACK);
                                else
                                        output(last_ack_ + 1, TCP_REASON_DUPACK);

                                if(transmits==1)
                                        dupacks_ = numdupacks_;
                                }
                        } else if (dupacks_ > numdupacks_)
                                ++cwnd_;
                }
                Packet::free(pkt);

#if 0
                if (trace_)
                        plot();
#endif /* 0 */

                /*
                 * Try to send more data
                 */
                if (dupacks_ == 0 || dupacks_ > numdupacks_ - 1)
                        send_much(0, 0, maxburst_);
}

void
VegasTcpAgent::timeout(int tno)
{
        if (tno == TCP_TIMER_RTX) {
                if (highest_ack_ == maxseq_ && !slow_start_restart_) {
                        /*
                         * TCP option:
                         * If no outstanding data, then don't do anything.
                         *
                         * Note:  in the USC implementation,
                         * slow_start_restart_ == 0.
                         * I don't know what the U. Arizona implementation
                         * defaults to.
                         */
                        return;
                };
                dupacks_ = 0;
                recover_ = maxseq_;
                last_cwnd_action_ = CWND_ACTION_TIMEOUT;
                reset_rtx_timer(0);
                ++nrexmit_;
```

```
            slowdown(CLOSE_CWND_RESTART|CLOSE_SSTHRESH_HALF);
            cwnd_ = double(v_slowstart_);
            v_newcwnd_ = 0;
            t_cwnd_changed_ = vegastime();
            send_much(0, TCP_REASON_TIMEOUT);
      } else {
            /* delayed-sent timer, with random overhead to avoid
             * phase effect. */
            send_much(1, TCP_REASON_TIMEOUT);
      };
}

void
VegasTcpAgent::output(int seqno, int reason)
{
      Packet* p = allocpkt();
      hdr_tcp *tcph = hdr_tcp::access(p);
      double now = Scheduler::instance().clock();
      tcph->seqno() = seqno;
      tcph->ts() = now;
      tcph->reason() = reason;

      /* if this is the 1st pkt, setup senttime[] and transmits[]
       * I alloc mem here, instrad of in the constructor, to cover
       * cases which วินโดws get set by each different tcp flows */
      if (seqno==0) {
            v_maxwnd_ = int(wnd_);
            if (v_sendtime_)
                  delete []v_sendtime_;
            if (v_transmits_)
                        delete []v_transmits_;
            v_sendtime_ = new double[v_maxwnd_];
            v_transmits_ = new int[v_maxwnd_];
            for(int i=0;i<v_maxwnd_;i++) {
                  v_sendtime_[i] = -1.;
                  v_transmits_[i] = 0;
            }
      }

      // record a find grained send time and # of transmits
      int index = seqno % v_maxwnd_;
      v_sendtime_[index] = vegastime();
      ++v_transmits_[index];

      /* support ndatabytes_ in output - Lloyd Wood 14 March 2000 */
      int bytes = hdr_cmn::access(p)->size();
      ndatabytes_ += bytes;
      ndatapack_++; // Added this - Debojyoti 12th Oct 2000
      send(p, 0);
      if (seqno == curseq_ && seqno > maxseq_)
            idle();  // Tell application I have sent everything so far

      if (seqno > maxseq_) {
            maxseq_ = seqno;
            if (!rtt_active_) {
                  rtt_active_ = 1;
```

```
                if (seqno > rtt_seq_) {
                        rtt_seq_ = seqno;
                        rtt_ts_ = now;
                }
        }
    } else {
        ++nrexmitpack_;
                nrexmitbytes_ += bytes;
    }

    if (!(rtx_timer_.status() == TIMER_PENDING))
        /* No timer pending.  Schedule one. */
        set_rtx_timer();
}

/*
 * return -1 if the oldest sent pkt has not been timeout (based on
 * fine grained timer).
 */
int
VegasTcpAgent::vegas_expire(Packet* pkt)
{
    hdr_tcp *tcph = hdr_tcp::access(pkt);
    double    elapse    =    vegastime()    -    v_sendtime_[(tcph-
>seqno()+1)%v_maxwnd_];
    if (elapse >= v_timeout_) {
        return(tcph->seqno()+1);
    }
    return(-1);
}
```

บทที่ 3

การดำเนินการวิจัย

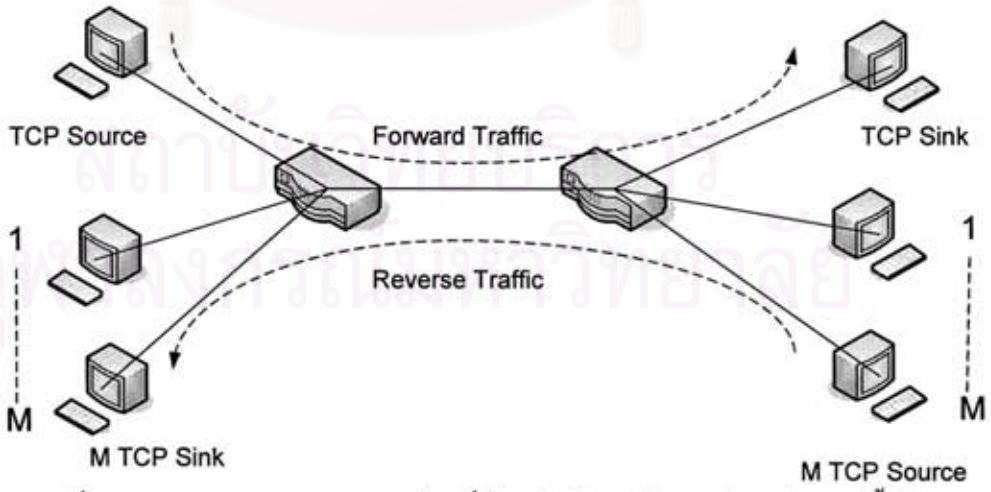
3.1 แบบจำลองโครงข่าย

การออกแบบจำลองของโครงข่ายเพื่อทดสอบสมรรถนะของแบบแผนควบคุมความความคับคั่งแบบต่างๆ ได้ออกแบบให้มีสภาพแวดล้อมที่แตกต่างกัน เพื่อให้ครอบคลุมรูปแบบการติดต่อของ TCP ให้มากที่สุดและเหมือนในโครงข่ายจริงให้มากที่สุด นอกจากนั้นยังได้ออกแบบโครงข่ายทั้งโครงข่ายภาคพื้นดินและโครงข่ายดาวเทียมเพื่อเปรียบเทียบแบบแผนควบคุมความคับคั่งแบบต่างว่ามีสมรรถนะอย่างไร เมื่อนำไปใช้งานบนดาวเทียมเทียบกับบนภาคพื้นดิน โดยมีรายละเอียดดังต่อไปนี้

3.1.1 สภาพแวดล้อมที่มีการติอต่อจุดเดียว (single TCP-connection scenario)

3.1.1.1 สภาพแวดล้อมที่มีการติอต่อจุดเดียวบนโครงข่ายภาคพื้นดิน (single TCP-connection scenario of Terrestrial Network)

แบบจำลองของสภาพแวดล้อมที่มีการติอต่อจุดเดียวบนโครงข่ายภาคพื้นดินแสดงไว้ในรูปที่ 3.1 จากรูปเราจะพิจารณาแบบแผนควบคุมความคับคั่งเฉพาะขาไป (Forward Traffice) เท่านั้น แต่จะมีการจำลองให้มีการส่งข้อมูล TCP ขากลับ (Reverse Traffic) ด้วย โดยจะกำหนดให้การส่งข้อมูลขาไป ให้ใช้แบบแผนควบคุมความคับคั่งที่จะนำมาพิจารณาในวิทยานิพนธ์คือ TCP-Newreno TCP-Peach และ TCP-Vegas ตามลำดับ ส่วนการส่งข้อมูลขากลับให้ใช้แบบแผนควบคุมความคับคั่ง TCP-Newreno เท่านั้น การเลือก

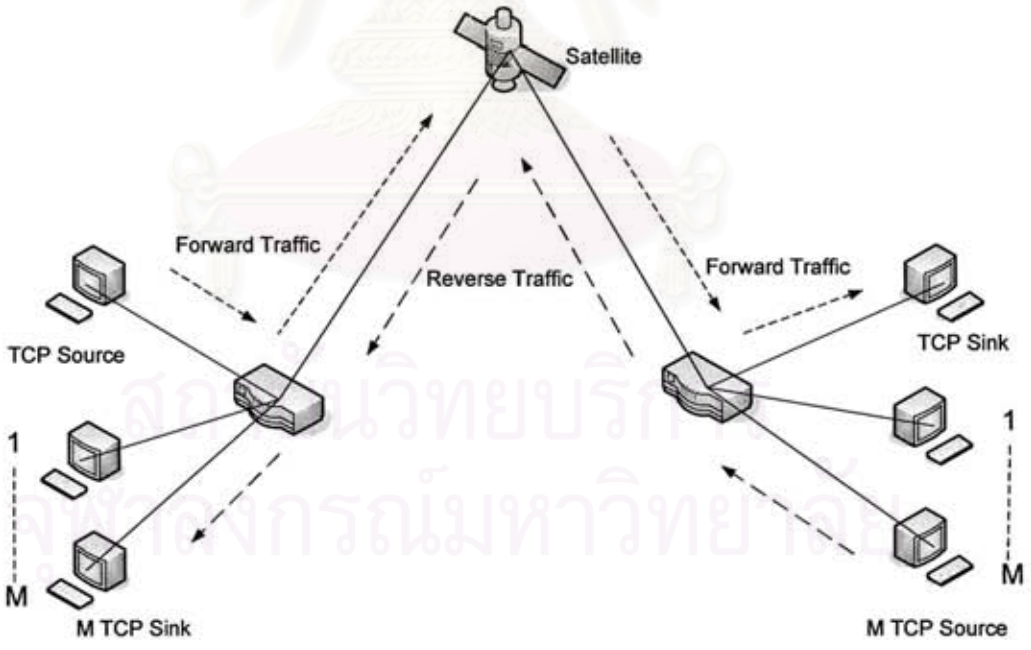


รูปที่ 3.1 แบบจำลองของสภาพแวดล้อมที่มีการติอต่อจุดเดียวบนโครงข่ายภาคพื้นดิน

TCP-Newreno มาพิจารณาเพราะ TCP-Newreno เป็นแบบแผนควบคุมความคับคั่งที่นิยมใช้ในเครือข่ายอินเตอร์เน็ทมากที่สุด ส่วนการเลือก TCP-Peach และ TCP-Vegas เพราะเป็นแบบแผนที่ใช้งานได้ดีในสภาพแวดล้อมที่มีค่าเวลาไปกลับสูงและมีค่า BER สูง ในการจำลองทุกแบบจำลองจะใช้ค่า แบนด์วิดธ์ในลิงค์ใช้สาย 1Mbps แบนด์วิดธ์ในลิงค์ไร้สาย 10 Mbps ค่า BER ของลิงค์ใช้สายเท่ากับ 10^{-4} ค่า BER ของลิงค์ไร้สายเท่ากับ 10^{-2} และขนาดของบัฟเฟอร์เท่ากับ 1 kB

3.1.1.2 สภาพแวดล้อมที่มีการติอต่อจุดเดียวบนโครงข่ายดาวเทียม (single TCP-connection scenario of Satellite Network)

แบบจำลองของสภาพแวดล้อมที่มีการติอต่อจุดเดียวบนโครงข่ายดาวเทียมแสดงไว้ในรูปที่ 3.2 จากรูปจะมีการเราจะพิจารณาแบบแผนควบคุมความคับคั่งเฉพาะขาไป (Forward Traffice) เท่านั้น แต่จะมีการจำลองให้มีการส่งข้อมูล TCP ขากลับ (Reverse Traffic) ด้วย โดยจะกำหนดให้การส่งข้อมูลขาไป ให้ใช้แบบแผนควบคุมความคับคั่งที่จะนำมาพิจารณาในวิทยานิพนธ์คือ TCP-Newreno TCP-Peach และ TCP-Vegas ตามลำดับ ส่วนการส่งข้อมูลขากลับให้ใช้แบบแผนควบคุมความคับคั่ง TCP-Newreno เท่านั้น

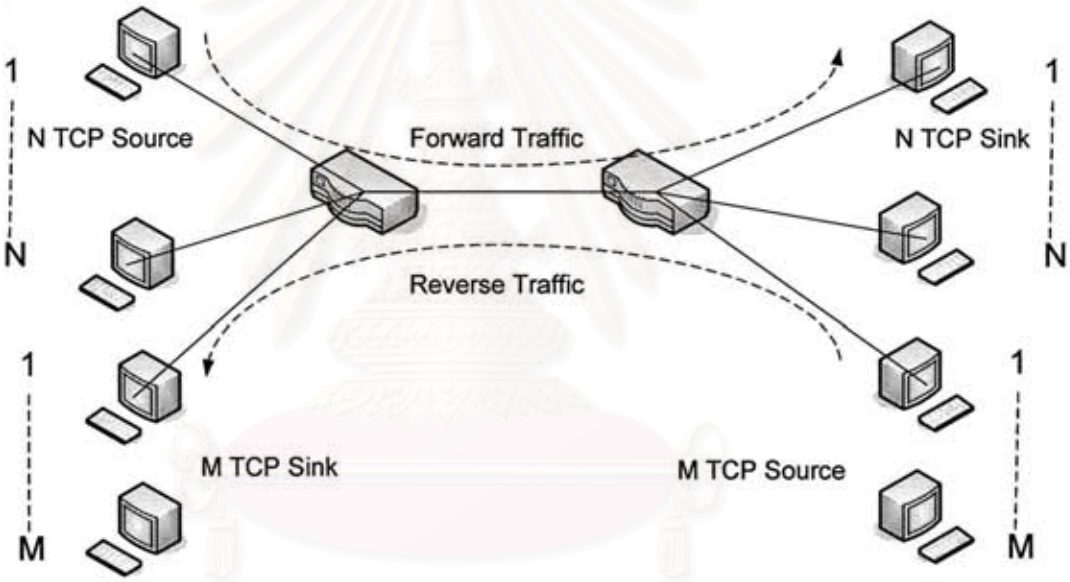


รูปที่ 3.2 แบบจำลองของสภาพแวดล้อมที่มีการติอต่อจุดเดียวบนโครงข่ายดาวเทียม

3.1.2 สภาพแวดล้อมที่มีความคับคั่งจุดเดียว (single bottleneck scenario)

3.1.2.1 สภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายภาคพื้นดิน (single bottleneck scenario of Terrestrial Network)

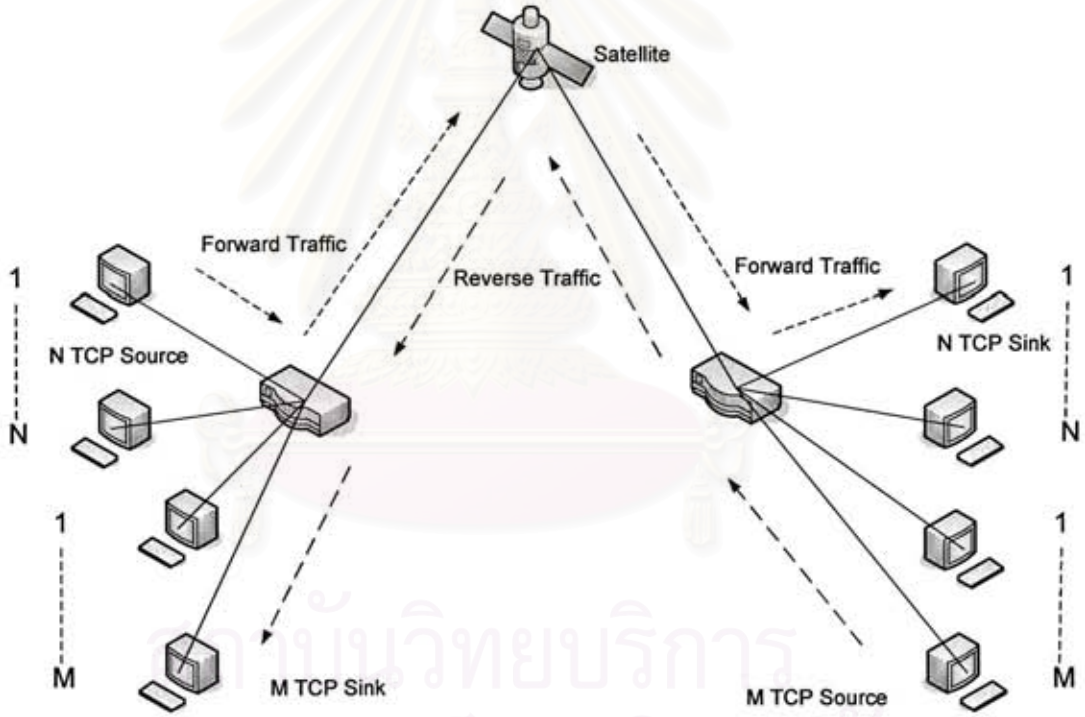
แบบจำลองของสภาพแวดล้อมที่มีการติอต่อจุดเดียวบนโครงข่ายภาคพื้นดินแสดงไว้ในรูปที่ 3.3 จากรูปจะมีการเราจะพิจารณาแบบแผนควบคุมความคับคั่งเฉพาะขาไป (Forward Traffice) เท่านั้น ซึ่งมีจำนวนเท่ากับ N ค่า N ที่ใช้ในการจำลองมีค่าเท่ากับ 10 หมายถึงมีผู้ส่ง TCP 10 โนดทำการส่งแพ็กเกตพร้อมกัน ส่วนการจำลองให้มีการส่งข้อมูล TCP ขากลับ (Reverse Traffic) ด้วย และให้มีจำนวนเท่ากับ M ค่า M ที่ใช้ในการจำลองมีค่าเท่ากับ 10 เช่นกัน โดยจะกำหนดให้การส่งข้อมูลขาไป ให้ใช้แบบแผนควบคุมความคับคั่งที่จะนำมาพิจารณาในวิทยานิพนธ์คือ TCP-Newreno TCP-Peach และ TCP-Vegas ตามลำดับ ส่วนการส่งข้อมูลขากลับให้ใช้แบบแผนควบคุมความคับคั่ง TCP-Newreno เท่านั้น



รูปที่ 3.3 แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายภาคพื้นดิน

3.1.2.2 สภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายดาวเทียม (single bottleneck scenario of Satellite Network)

แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายดาวเทียม แสดงไว้ในรูปที่ 3.4 จากรูปจะมีการเราจะพิจารณาหาสมรรถนะแบบแผนควบคุมความคับคั่งเฉพาะขาไป (Forward Traffice) เท่านั้น ซึ่งมีจำนวนเท่ากับ N ค่า N ที่ใช้ในการจำลองมีค่าเท่ากับ 10 หมายถึงมีผู้ส่ง TCP 10 โนดทำการส่งแพ็กเกตพร้อมกัน ส่วนการจำลองให้มีการส่งข้อมูล TCP ขากลับ (Reverse Traffic) ด้วย และให้มีจำนวนเท่ากับ M ค่า M ที่ใช้ในการจำลองมีค่าเท่ากับ 10 เช่นกัน โดยจะกำหนดให้การส่งข้อมูลขาไป ให้ใช้แบบแผนควบคุมความคับคั่งที่จะนำมาพิจารณาในวิทยานิพนธ์คือ TCP-Newreno TCP-Peach และ TCP-Vegas ตามลำดับ ส่วนการส่งข้อมูลขากลับให้ใช้แบบแผนควบคุมความคับคั่ง TCP-Newreno เท่านั้น

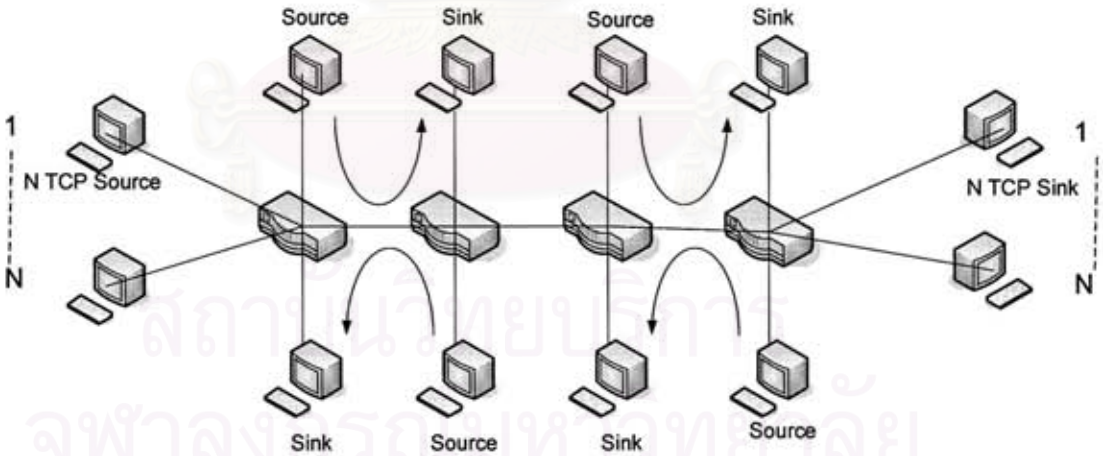


รูปที่ 3.4 แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายดาวเทียม

3.1.3 สภาพแวดล้อมที่มีความคับคั่งหลายจุด (multiple bottleneck scenario)

3.1.3.1 สภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายภาคพื้นดิน (multiple bottleneck scenario of Terrestrial Network)

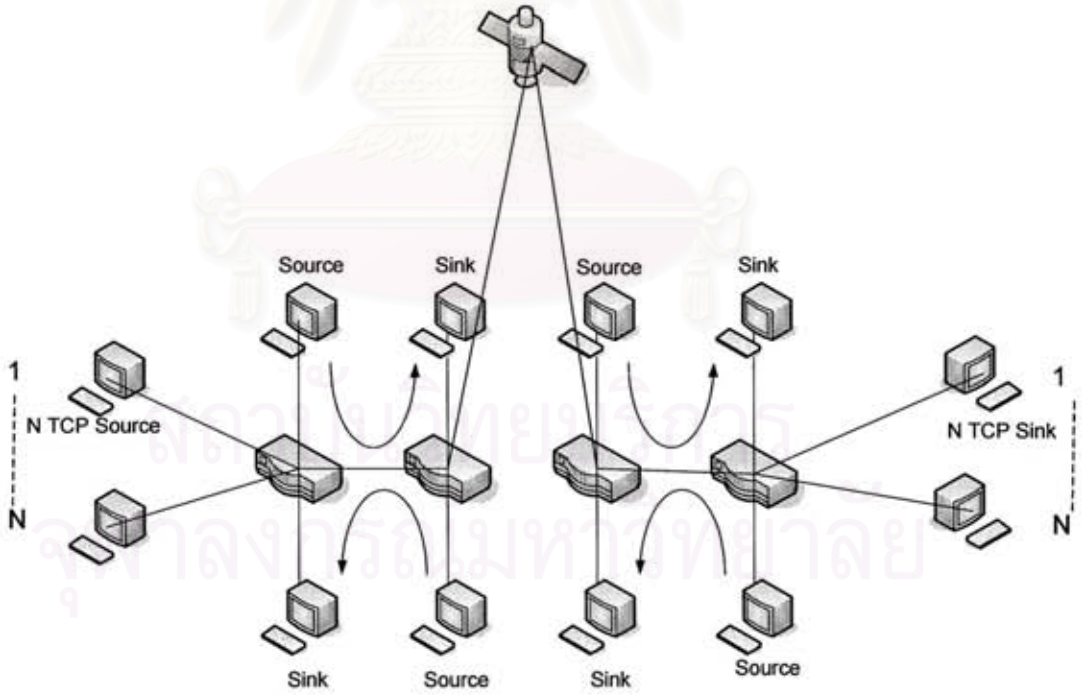
แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายภาคพื้นดิน แสดงไว้ในรูปที่ 3.5 สภาพแวดล้อมลักษณะนี้มีเหมือนกับการใช้งานจริงมากที่สุดบนโครงข่ายภาคพื้นดิน จากรูปจะมีการเราจะพิจารณาหาสมรรถนะแบบแผนควบคุมความคับคั่งเฉพาะขาไป (Forward Traffice) เท่านั้น และจะไม่พิจารณาหาสมรรถนะแบบแผนควบคุมความคับคั่งขาไปที่เกิดขึ้นระหว่างทาง ซึ่งมีจำนวนเท่ากับ N ค่า N ที่ใช้ในการจำลองมีค่าเท่ากับ 10 หมายถึงมีผู้ส่ง TCP 10 โนดทำการส่งแพ็กเกตพร้อมกัน ในสภาพแวดล้อมนี้จะกำหนดให้ระหว่างมีจุดที่ทำให้เกิดความคับคั่งได้หลายจุด กำหนดให้มี 3 จุดที่เกิดความคับคั่งขึ้นได้ และในแต่ละจุดจะมีทราฟฟิคทั้งในส่วนของการส่งข้อมูลขาไปและส่วนการจำลองให้มีการส่งข้อมูล TCP ขากลับ ซึ่งกำหนดให้ในแต่ละจุดมีจำนวนผู้ส่งทั้งขาไปและขากลับมีค่าเท่ากับ 10 ดังนั้นในแต่ละจุดจะมีผู้ส่ง TCP รวมเท่ากับ 20 และให้มีจำนวนรวมทั้ง 3 จุดความคับคั่งเท่ากับ 60 โดยจะกำหนดให้การส่งข้อมูลขาไป ให้ใช้แบบแผนควบคุมความคับคั่งที่จะนำมาพิจารณาในวิทยานิพนธ์คือ TCP-Newreno TCP-Peach และ TCP-Vegas ตามลำดับ ส่วนการส่งข้อมูลที่อยู่ระหว่างจุดที่สามารถเกิดความคับคั่งได้นั้นให้ใช้แบบแผนควบคุมความคับคั่ง TCP-Newreno เท่านั้น ทั้งขาไปและขากลับ



รูปที่ 3.5 แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายภาคพื้นดิน

1.3.2 สภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายดาวเทียม (multiple bottleneck scenario of Satellite Network)

แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายดาวเทียม แสดงไว้ในรูปที่ 3.6 สภาพแวดล้อมลักษณะนี้มีเหมือนกับการใช้งานจริงบนโครงข่ายดาวเทียมมากที่สุด จากรูปจะมีการเราจะพิจารณาหาสมรรถนะแบบแผนควบคุมความคับคั่งเฉพาะขาไป (Forward Traffice) เท่านั้น และจะไม่พิจารณาหาสมรรถนะแบบแผนควบคุมความคับคั่งขาไปที่เกิดขึ้นระหว่างทาง ซึ่งมีจำนวนเท่ากับ N ค่า N ที่ใช้ในการจำลองมีค่าเท่ากับ 10 หมายถึงมีผู้ส่ง TCP 10 โนดทำการส่งแพ็กเกตพร้อมกัน ในสภาพแวดล้อมนี้จะกำหนดให้ระหว่างมีจุดที่ทำให้เกิดความคับคั่งได้หลายจุด กำหนดให้มี 3 จุดที่เกิดความคับคั่งขึ้นได้ และในแต่ละจุดจะมีทราฟฟิคทั้งในส่วนของการส่งข้อมูลขาไปและส่วนการจำลองให้มีการส่งข้อมูล TCP ขากลับ ซึ่งกำหนดให้ในแต่ละจุดมีจำนวนผู้ส่งทั้งขาไปและขากลับม่ค่าเท่ากับ 10 ดังนั้นในแต่ละจุดจะมีผู้ส่ง TCP รวมเท่ากับ 20 และให้มีจำนวนรวมทั้ง 3 จุดความคับคั่งเท่ากับ 60 โดยจะกำหนดให้การส่งข้อมูลขาไป ให้ใช้แบบแผนควบคุมความคับคั่งที่จะนำมาพิจารณาในวิทยานิพนธ์คือ TCP-Newreno TCP-Peach และ TCP-Vegas ตามลำดับ ส่วนการส่งข้อมูลที่อยู่ระหว่างจุดที่สามารถเกิดความคับคั่งได้นั้นให้ใช้แบบแผนควบคุมความคับคั่ง TCP-Newreno เท่านั้น ทั้งขาไปและขากลับ



รูปที่ 3.6 แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายดาวเทียม

3.2 พารามิเตอร์ที่ใช้ในการประเมินสมรรถนะ

การประเมินหาสมรรถนะจะใช้พารามิเตอร์ goodput, fairness และ friendliness ในโครงข่ายดาวเทียม ด้วยเงื่อนไขการเกิดความคับคั่งและไม่เกิดความคับคั่ง โดยใช้ NS-2 Simulator

3.2.1 สมรรถนะ goodput [11],[12]

goodput คือ ปริมาณข้อมูลที่แท้จริงที่ส่งเข้าสู่โครงข่าย แสดงไว้ในสมการที่ 6 ซึ่งแสดงถึงสมรรถนะของโครงข่าย แบบแผน TCP ที่ดี ต้องส่งข้อมูลให้มาก แต่ในขณะเดียวกันต้องเข้ากันได้กับ TCP แบบอื่นๆ ในการใช้ทรัพยากรและแบนด์วิดท์ของโครงข่ายร่วมกัน

$$Goodput = \frac{sent\_data - retransmission\_data}{transfer\_time} \qquad \text{สมการที่ 6}$$

3.2.2 Fairness ของ TCP [12]

Fairness คือ ความสามารถในการทำงานร่วมกันของแบบแผน TCP เดียวกัน (intra-protocol) ในการวัดสมรรถนะ Fairness จะใช้ Fairness Index Function ดังนิยามไว้ในสมการที่ 7

$$F(x) = \frac{(\sum x_i)^2}{n(\sum x_i^2)} \qquad \text{สมการที่ 7}$$

x_i คือ throughput ของการติดต่อ TCP ลำดับที่ i

n คือ จำนวนการติดต่อ TCP

$F(x)$ จะมีค่าตั้งแต่ $1/n$ ถึง 1 หากมีการจัดสรรการใช้แบนด์วิดท์แบบสมบูรณ์ Fairness Index Function จะมีค่าเท่ากับ 1 และหากมีเพียงการติดต่อเดียวที่ใช้แบนด์วิดท์ทั้งหมดจะมีค่า Fairness Index Function เท่ากับ $1/n$

3.2.3 Friendliness ของ TCP [12]

Friendliness คือ ความสามารถในการทำงานร่วมกันของแบบแผน TCP ที่ต่างกัน (inter-protocol) ในการวัดสมรรถนะ Friendliness จะทำโดยการวัดค่า Throughput ของแต่ละแบบแผน TCP ที่อยู่บนโครงข่ายเดียวกัน (ที่ต้องการใช้ช่องสัญญาณเดียวกัน)

บทที่ 4

ผลการวิเคราะห์ข้อมูล

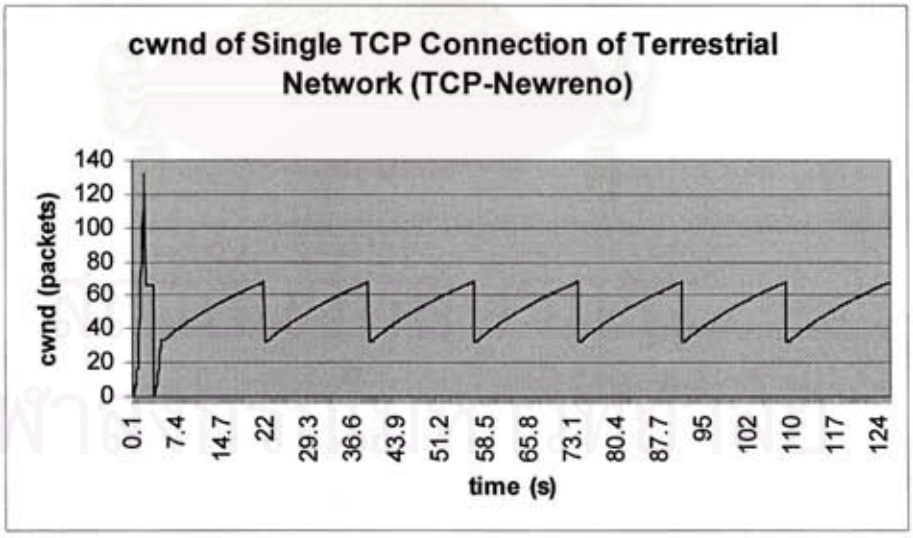
4.1 ผลการวัดค่า goodput

ผลการวิเคราะห์ข้อมูลของโครงข่ายแบบต่างๆที่กล่าวไว้ในบทที่ 3 เป็นดังต่อไปนี้

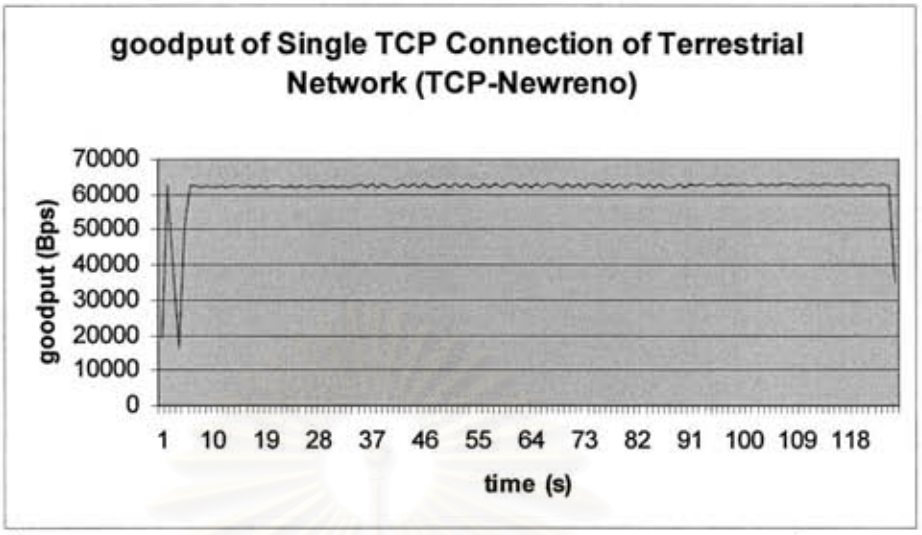
4.1.1 สภาพแวดล้อมที่มีการติดต่อจุดเดียว (single TCP-connection scenario)

4.1.1.1 สภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายภาคพื้นดิน (single TCP-connection scenario of Terrestrial Network)

จากรูปที่ 4.1 แสดงค่า *cwnd* ของ TCP-Newreno ของแบบจำลองสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายภาคพื้นดิน สังเกตุที่เวลาก่อนวินาทีที่ 5.2 จะอยู่ในกระบวนการ slow start และในวินาทีที่ 2.4 จะเกิดสูญหายของแพ็กเกตขึ้นทำให้กระบวนการ slow start ลดค่า *cwnd* ลงครึ่งหนึ่ง และยังเกิดการสูญหายของแพ็กเกตอีกครั้งที่วินาทีที่ 3.5 สาเหตุมาจาก timeout ทำให้ค่า goodput ในช่วงเวลาดังกล่าวมีค่าลดลงมากตามแสดงในรูปที่ 4.2 ส่วนกระบวนการ congestion avoidance จะอยู่ในช่วงวินาทีตั้งแต่ 5.2 ขึ้นไป จะมีลักษณะเป็นรูปแบบซ้ำๆกัน โดยการเพิ่มของค่า *cwnd* จะเพิ่มขึ้นเกือบจะเป็นเชิงเส้น ทำให้ค่า goodput ในช่วงนี้ค่อนข้างมีค่าคงที่ จนเกิดความคับคั่งขึ้นจึงทำให้ต้องลดค่า *cwnd* ลงมาครึ่งหนึ่ง



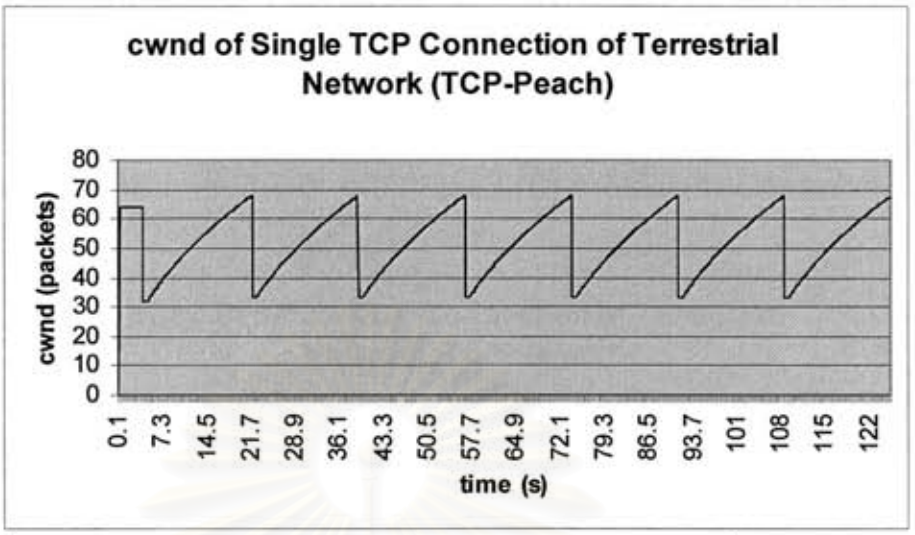
รูปที่ 4.1 ค่า *cwnd* ของ TCP-Newreno ของแบบจำลองของสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายภาคพื้นดิน



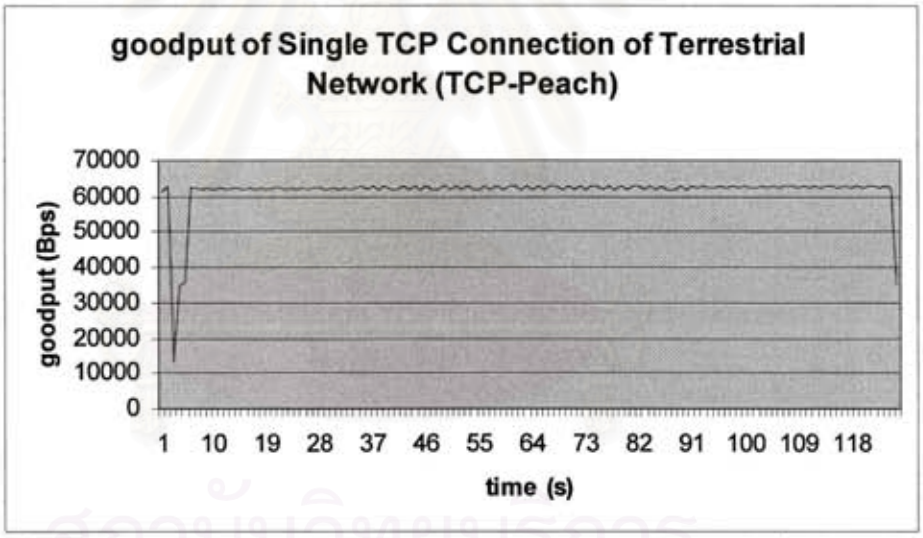
รูปที่ 4.2 ค่า goodput ของ TCP-Newreno ของ แบบจำลองของสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายภาคพื้นดิน

จากรูปที่ 4.3 แสดงค่า *cwnd* ของ TCP-Peach ของแบบจำลองสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายภาคพื้นดิน TCP-Peach จะเริ่มต้นกระบวนการ sudden start รายละเอียดกล่าวอยู่ในบทที่ 2 ในวินาทีที่ 0 ถึง 4 จะอยู่ในกระบวนการ sudden start ทำให้ค่า *cwnd* มีค่าสูงในช่วงนี้ ทำให้ค่า goodput มีค่าสูงเช่นกัน ดังแสดงไว้ในรูปที่ 4.4 จน dummy segment เริ่มเกิดการสูญหายในวินาทีที่ 4 ขึ้นไป กระบวนการจึงเริ่มเข้าสู่ congestion avoidance ซึ่งเป็นกระบวนการที่มีอัลกอริธึมเหมือนกับ congestion avoidance ใน TCP-Newreno

จากรูปที่ 4.5 แสดงค่า *cwnd* ของ TCP-Vegas ของแบบจำลองสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายภาคพื้นดิน TCP-Vegas จะเริ่มต้นกระบวนการ sudden start รายละเอียดกล่าวอยู่ในบทที่ 2 ในวินาทีที่ 0 ถึง 4 จะอยู่ในกระบวนการ sudden start ทำให้ค่า *cwnd* มีค่าสูงในช่วงนี้ ทำให้ค่า goodput มีค่าสูงเช่นกัน จน dummy segment เริ่มเกิดการสูญหาย กระบวนการจึงเริ่มเข้าสู่ congestion avoidance ซึ่งเป็นกระบวนการที่มีอัลกอริธึมเหมือนกับ congestion avoidant ใน TCP-Newreno

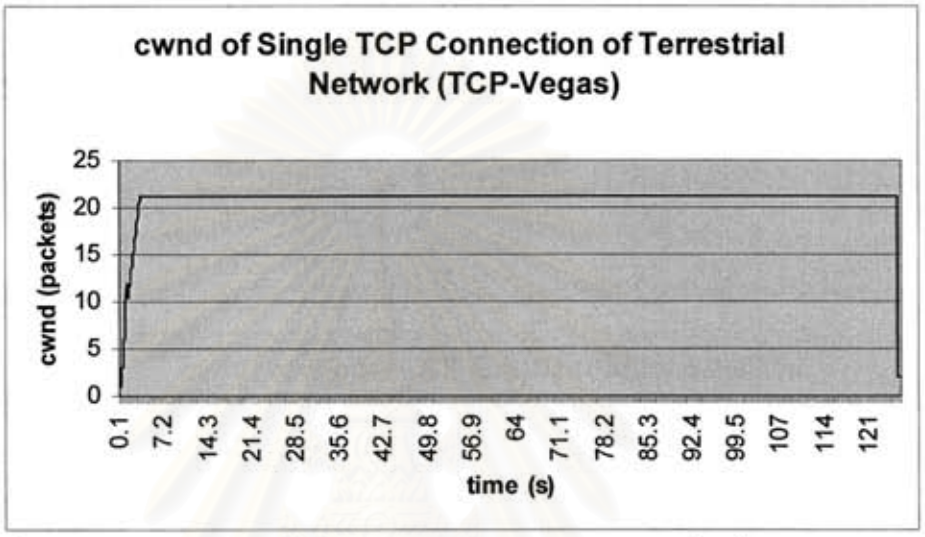


รูปที่ 4.3 ค่า *cwnd* ของ TCP-Peach ของ แบบจำลองของสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายภาคพื้นดิน

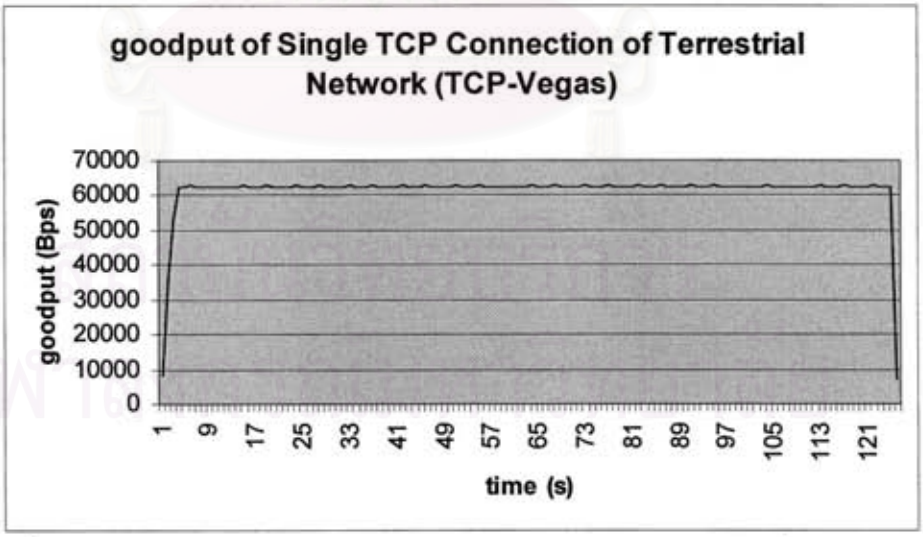


รูปที่ 4.4 ค่า goodput ของ TCP-Peach ของ แบบจำลองของสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายภาคพื้นดิน

จากรูปที่4.5 แสดงค่า *cwnd* ของ TCP-Vegas ของ แบบจำลองของสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายภาคพื้นดิน สังเกตว่าค่า *cwnd* ในกระบวนการ congestion avoidance จะมีค่าคงที่ตลอดเพราะค่า *Diff* อยู่ในช่วง threshold α และ β ตลอดกระบวนการ ทำให้ค่า goodput มีค่าคงที่ตลอดกระบวนการเช่นกัน



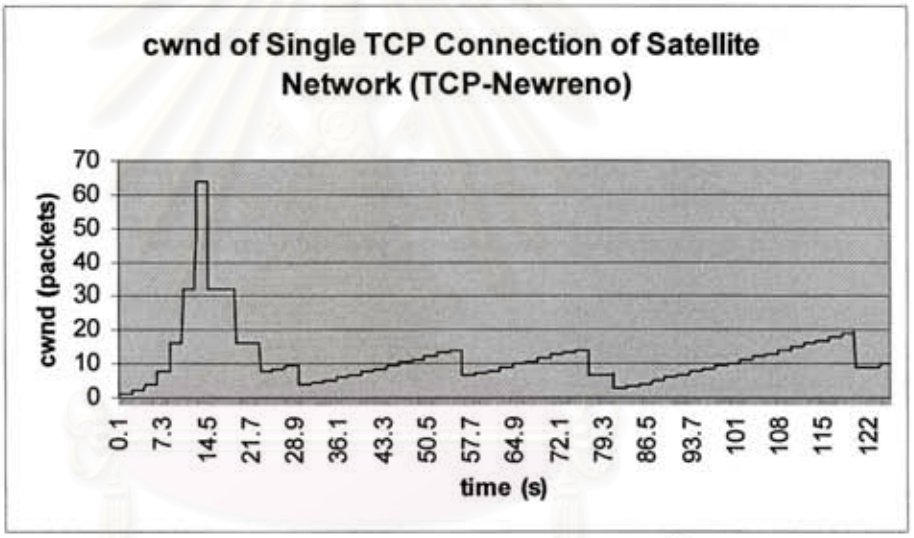
รูปที่ 4.5 ค่า *cwnd* ของ TCP-Vegas ของ แบบจำลองของสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายภาคพื้นดิน



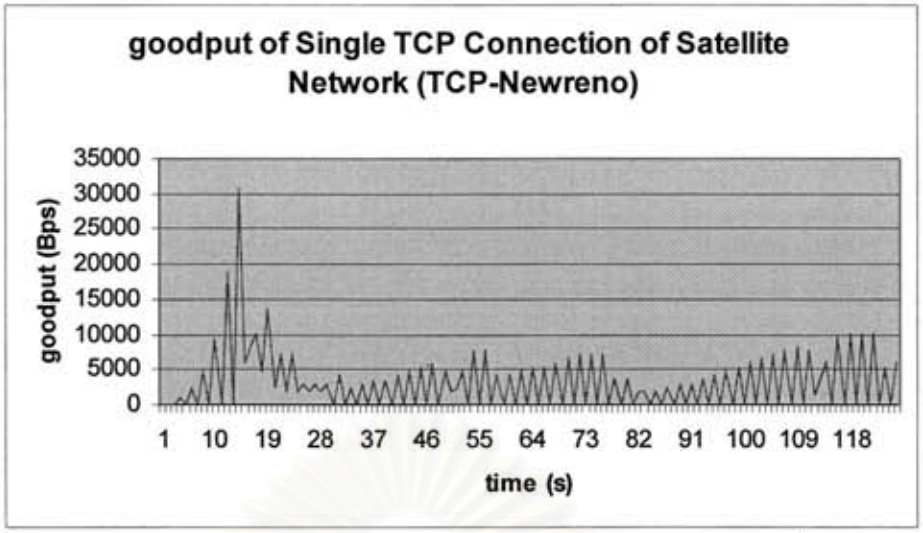
รูปที่ 4.6 ค่า goodput ของ TCP-Vegas ของ แบบจำลองของสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายภาคพื้นดิน

4.1.1.2 สภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายดาวเทียม (single TCP-connection scenario of Satellite Network)

จากรูปที่ 4.7 แสดงค่า *cwnd* ของ TCP-Newreno ของแบบจำลองสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายดาวเทียม สังเกตุที่เวลาก่อนวินาทีที่ 13.2 จะอยู่ในกระบวนการ slow start ส่วนกระบวนการ congestion avoidance จะอยู่ในช่วงวินาทีตั้งแต่ 13.2 ขึ้นไป จากรูปที่ 4.8 goodput ของระบบจะมีค่าต่ำมากเมื่อเปรียบเทียบกับแบบจำลองภาคพื้นดิน เพราะมีสาเหตุมาจาก ค่า *RTT* ที่มีค่ามาก ทำให้ระยะเวลาที่จะไปถึงความเร็วสูงใช้เวลามาก นอกจากนั้นในโครงข่าวดาวเทียมยังมีค่า *BER* สูง จึงทำให้ในกระบวนการ congestion avoidance มีการลดค่า *cwnd* อยู่เสมอ ส่งผลให้ค่า goodput มีค่าลดลงด้วย

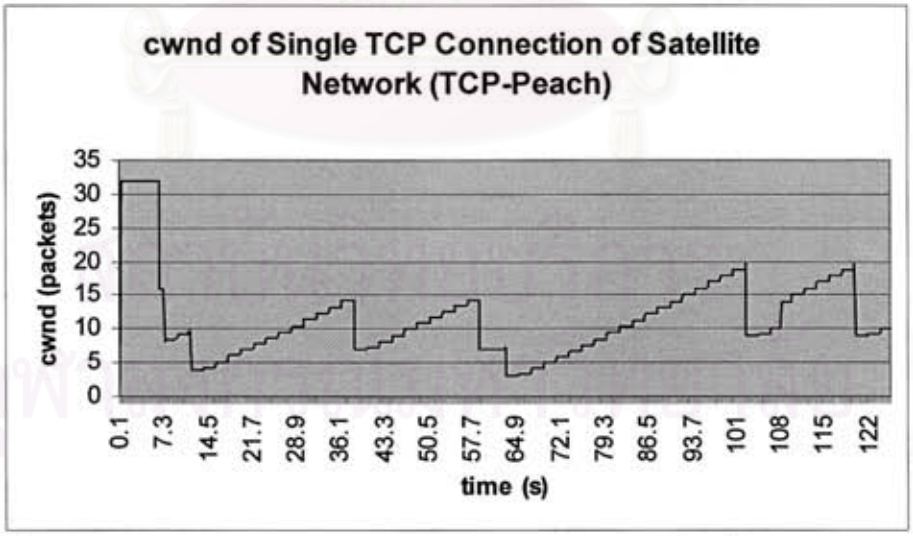


รูปที่ 4.7 ค่า *cwnd* ของ TCP-Newreno แบบจำลองของสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายดาวเทียม

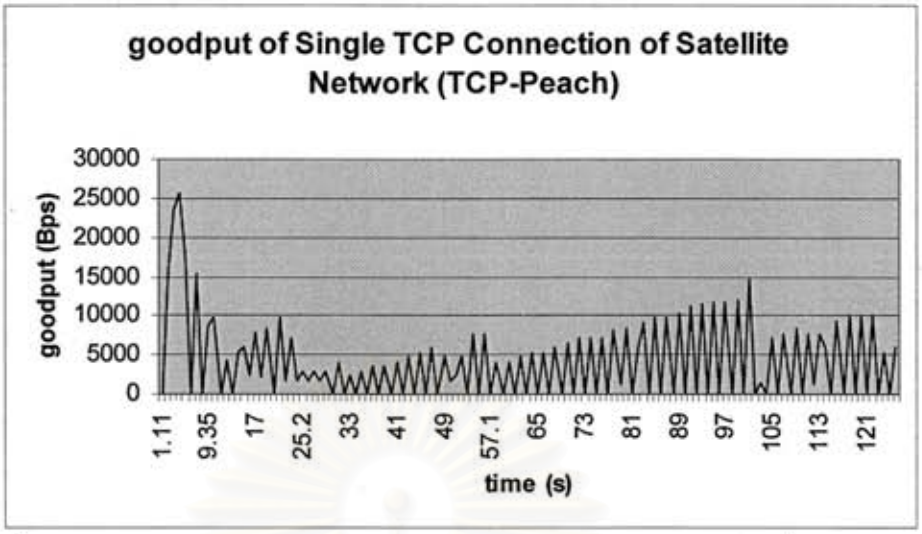


รูปที่ 4.8 ค่า goodput ของ TCP-Newreno ของ แบบจำลองของสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายดาวเทียม

จากรูปที่ 4.9 แสดงค่า *cwnd* ของ TCP-Peach ของแบบจำลองสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายดาวเทียม สังเกตุที่เวลาก่อนวินาทีที่ 5.5 จะอยู่ในกระบวนการ sudden start ส่วนกระบวนการ congestion avoidance จะอยู่ในช่วงวินาทีตั้งแต่ 14.7 ขึ้นไป ในโครงข่ายดาวเทียมค่า goodput จะมีค่าต่ำ ซึ่งมีคำอธิบายคล้ายกับย่อหน้าข้างต้น เมื่อคำนวณค่าเฉลี่ยของ goodput ของ TCP-Peach จะมีค่าดีกว่า TCP-Newreno เท่ากับ 24.53%

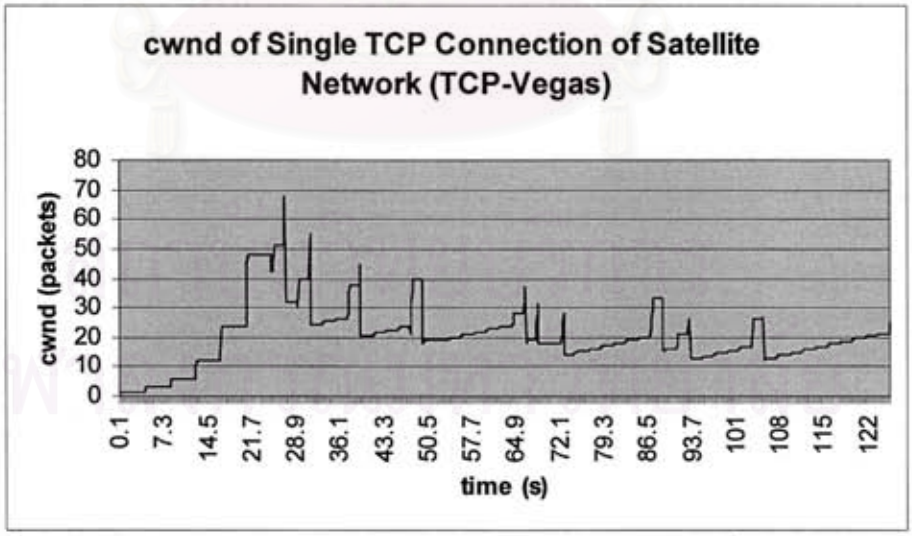


รูปที่ 4.9 ค่า *cwnd* ของ TCP-Peach ของ แบบจำลองของสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายดาวเทียม

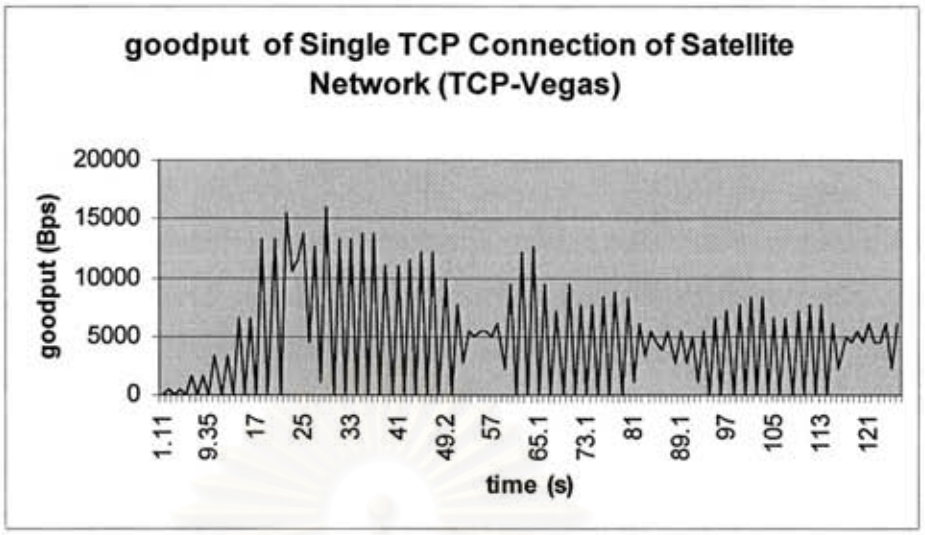


รูปที่ 4.10 ค่า goodput ของ TCP-Peach ของ แบบจำลองของสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายดาวเทียม

จากรูปที่ 4.11 แสดงค่า *cwnd* ของ TCP-Vegas ของ แบบจำลองของสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายดาวเทียม ช่วงวินาทีที่ 0 ถึง 21.8 จะอยู่ในกระบวนการ slow start ซึ่งจะทำให้ค่าในช่วงนี้มี goodput ต่ำ เพราะใช้เวลาในการอยู่ในกระบวนการ slow start นาน และในกระบวนการ congestion avoidance มีการประเมินหาแบนด์วิดท์ของช่องสัญญาณด้วยค่าตัวแปร *Diff* TCP-Vegas จึงมีการลดค่า *cwnd* ต่อต่อเมื่อเกิดการคับคั่งของช่องสัญญาณเกิดขึ้นจริง

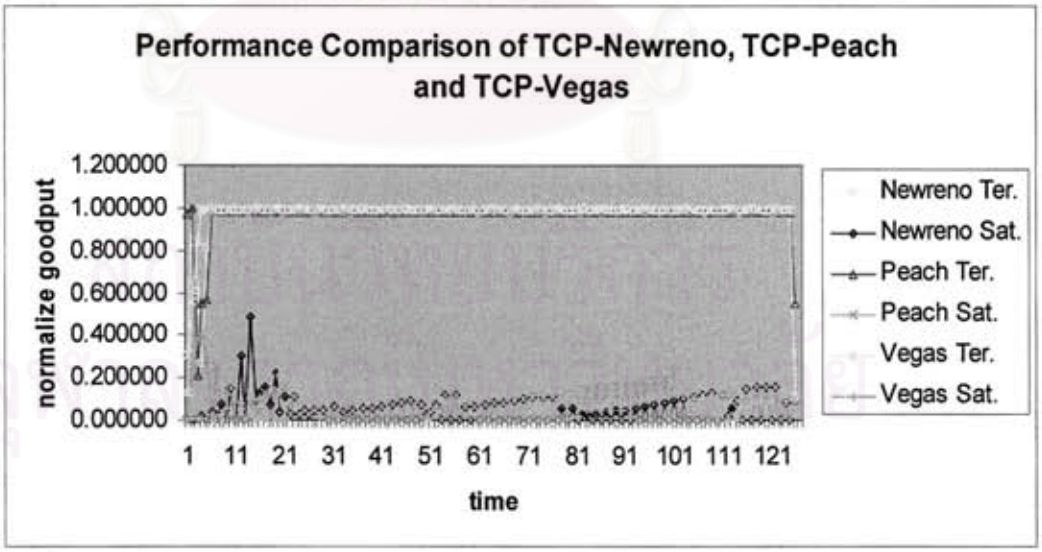


รูปที่ 4.11 ค่า *cwnd* ของ TCP-Vegas ของแบบจำลองของสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายดาวเทียม



รูปที่ 4.12 ค่า goodput ของ TCP-Vegas ของ แบบจำลองของสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายดาวเทียม

จากรูปที่ 4.13 เปรียบเทียบสมรรถนะระหว่าง TCP-Newreno TCP-Peach และ TCP-Vegas สรุปได้ว่าในสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายภาคพื้นดิน สมรรถนะของ TCP ทั้งสามรุ่นนี้จะมีสมรรถนะไม่แตกต่างกันมากนัก เมื่อพิจารณาในบนโครงข่ายดาวเทียม TCP-Vegas จะมี goodput เฉลี่ยมีค่ามากกว่า TCP-Newreno เท่ากับ 30.41% และ TCP-Peach มีค่า goodput เฉลี่ยมากกว่า TCP-Newreno เท่ากับ 24.53%

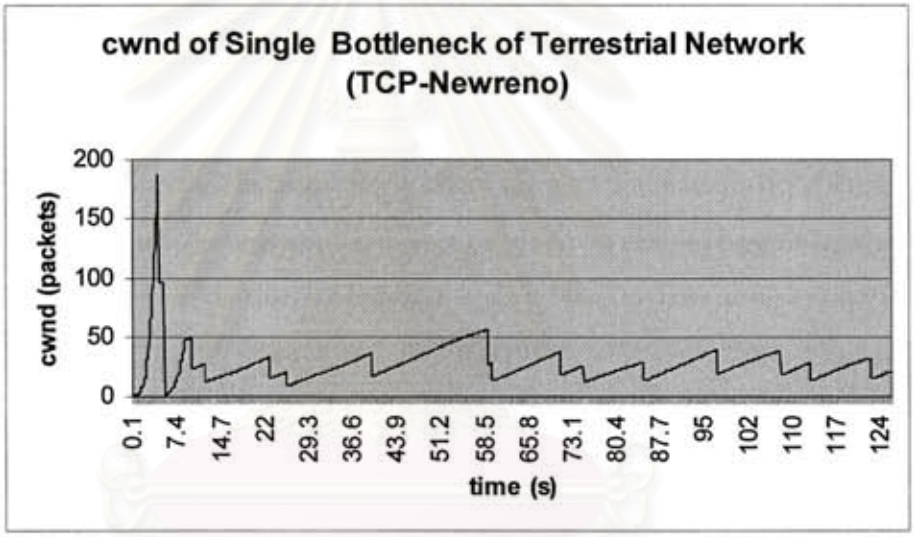


รูปที่ 4.13 เปรียบเทียบค่า goodput ระหว่าง TCP-Newreno TCP-Peach และ TCP-Vegas ของแบบจำลองของสภาพแวดล้อมที่มีการติดต่อจุดเดียว

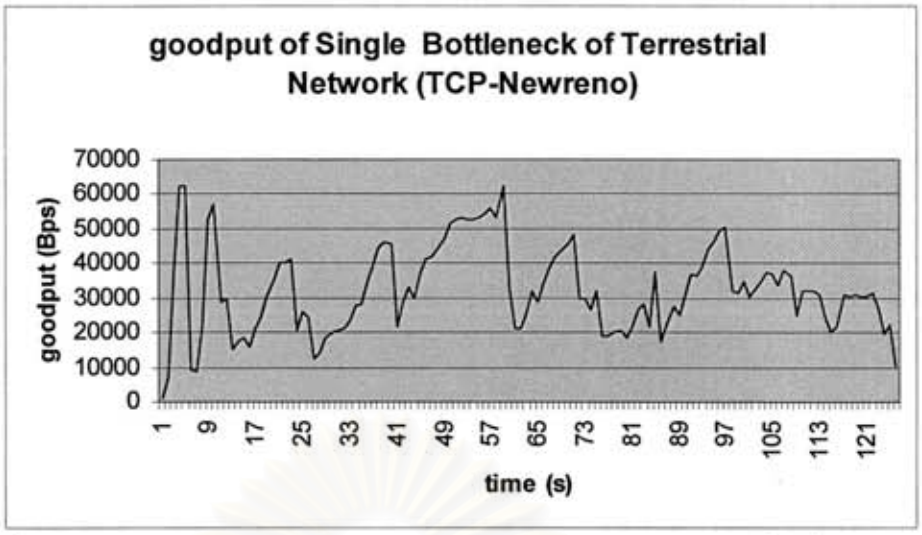
4.1.2 สภาพแวดล้อมที่มีความคับคั่งจุดเดียว (single bottleneck scenario)

4.1.2.1 สภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายภาคพื้นดิน (single bottleneck scenario of Terrestrial Network)

ในสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายภาคพื้นดินจะมีค่า cwnd ที่ไม่เป็นรูปแบบเหมือนกับสภาพแวดล้อมที่มีการติดต่อจุดเดียวบนโครงข่ายภาคพื้นดิน เพราะมีผู้ส่ง TCP หลายผู้ส่งร่วมกันใช้ช่องสัญญาณเดียวกันทำให้เกิดความคับคั่งขึ้น ณ ขณะใดก็ได้ ทำให้ค่า cwnd มีค่าลดลงเมื่อเกิดความคับคั่งขึ้นหรือ ลดลงเมื่อเกิดการสูญหายของแพ็กเกต จากรูปที่ 4.14 จะเห็นได้ว่าค่า cwnd ของ TCP-Newreno มีค่าลดลงที่ใด ณ จุดนั้นจะเกิดการสูญหายของแพ็กเกตสูญหายของแพ็กเกตหรือเกิดความคับคั่งของช่องสัญญาณเกิดขึ้น

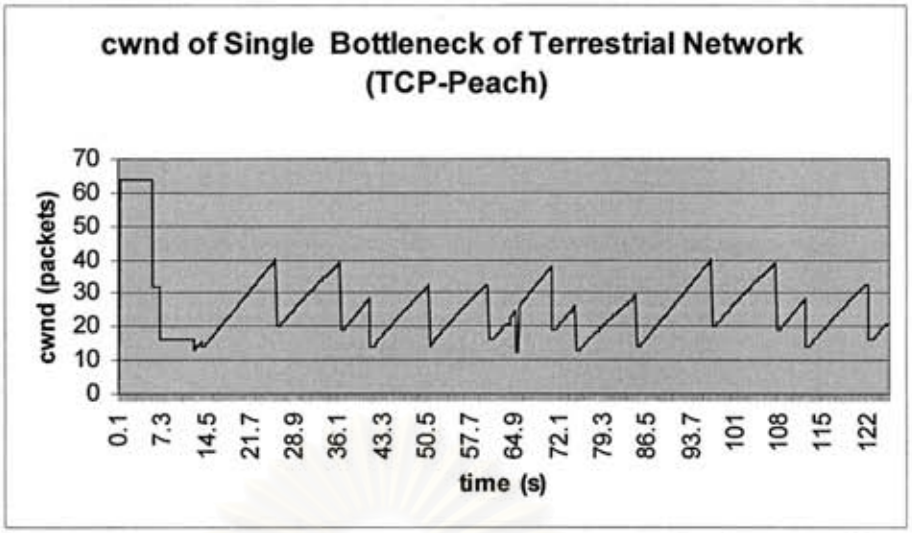


รูปที่ 4.14 ค่า *cwnd* ของ TCP-Newreno ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายภาคพื้นดิน

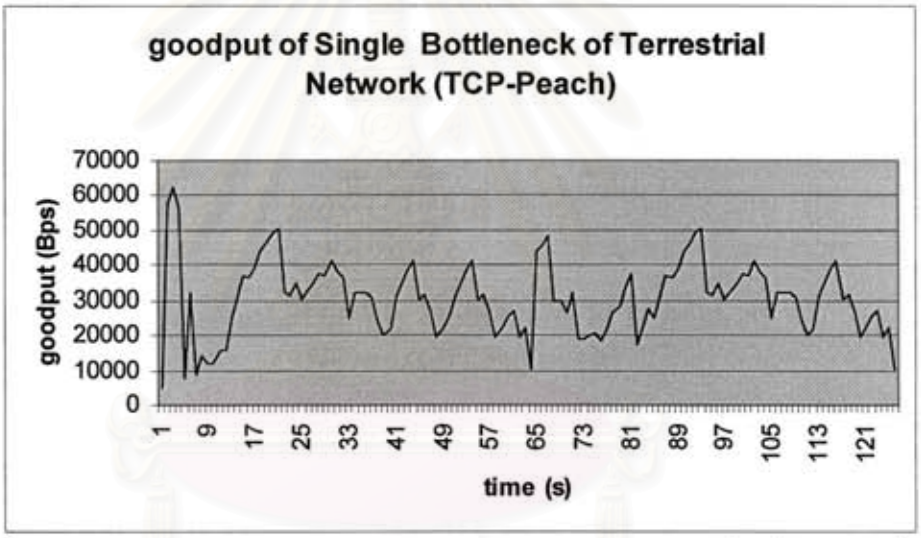


รูปที่ 4.15 ค่า goodput ของ TCP-Newreno ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายภาคพื้นดิน

จากรูปที่ 4.16 แสดงค่า cwnd ของแบบจำลองที่มีความคับคั่งจุดเดียวบนโครงข่ายภาคพื้นดิน ในวินาทีที่ 0 ถึง 5 TCP-Peach จะอยู่ในกระบวนการ sudden start จะมีค่า cwnd สูงซึ่งมีค่าตามที่อธิบายไว้ในหัวข้อ 2.2.2 และตั้งแต่วินาทีที่ 5 ขึ้นไป จะอยู่ในกระบวนการ congestion avoidance สังเกตุในวินาทีที่ 65.5 เกิดการสูญหายของแพ็กเกตเกิดขึ้นในช่วงแรกของการสูญหาย TCP-Peach จะยังคงไม่ทราบว่า การสูญหายของแพ็กเกตนั้นมีสาเหตุมาจากความเสียหายของแพ็กเกตหรือเกิดจากความคับคั่งในโครงข่าย ทำให้ในต้องลดค่า cwnd ลง จากนั้นการส่ง dummy segment ในระหว่างกระบวนการ rapid recovery ทำให้ทราบว่า การสูญหายของแพ็กเกตนั้นเกิดมาจากสาเหตุใด โดยถ้า ผู้ส่งได้รับ n_{dummy} ACK ตอบกลับมาจะระบุสาเหตุการสูญหายมาจากความเสียหายของแพ็กเกตเอง และกระบวนการ rapid recovery จะกำหนดให้ค่า cwnd ให้มีค่าเท่าเดิมก่อนที่จะเข้าสู่กระบวนการ fast retransmission ส่งผลให้ค่า goodput ในรูปที่ มีค่าลดลงชั่วขณะหนึ่ง แล้วจึงกลับมีค่าเพิ่มขึ้นอย่างรวดเร็ว อย่างไรก็ตามเมื่อเปรียบเทียบค่า goodput เฉลี่ยของการส่งทั้งหมดระหว่าง TCP-Newreno กับ TCP-Peach แล้ว TCP-Newreno จะมีค่า goodput เฉลี่ยสูงกว่า TCP-Peach เท่ากับ 5.22%

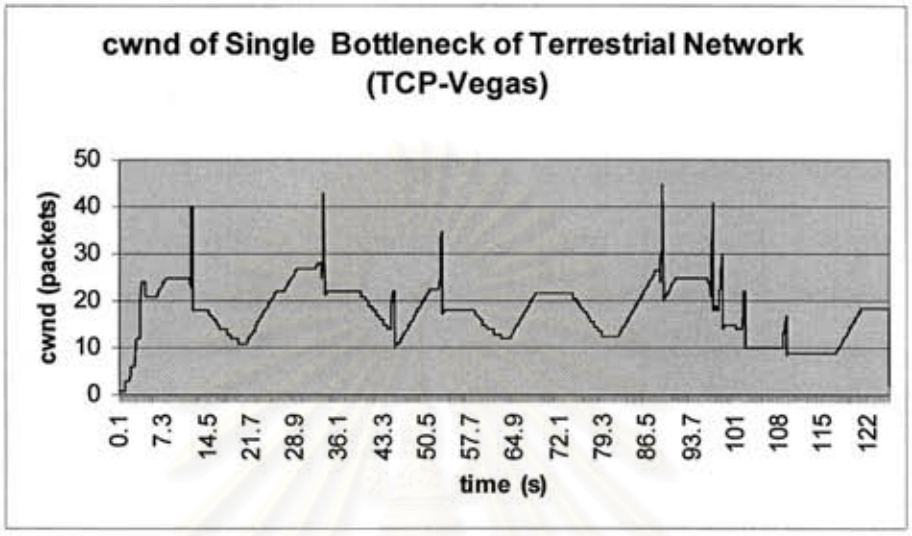


รูปที่ 4.16 ค่า *cwnd* ของ TCP-Peach ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียว บนโครงข่ายภาคพื้นดิน

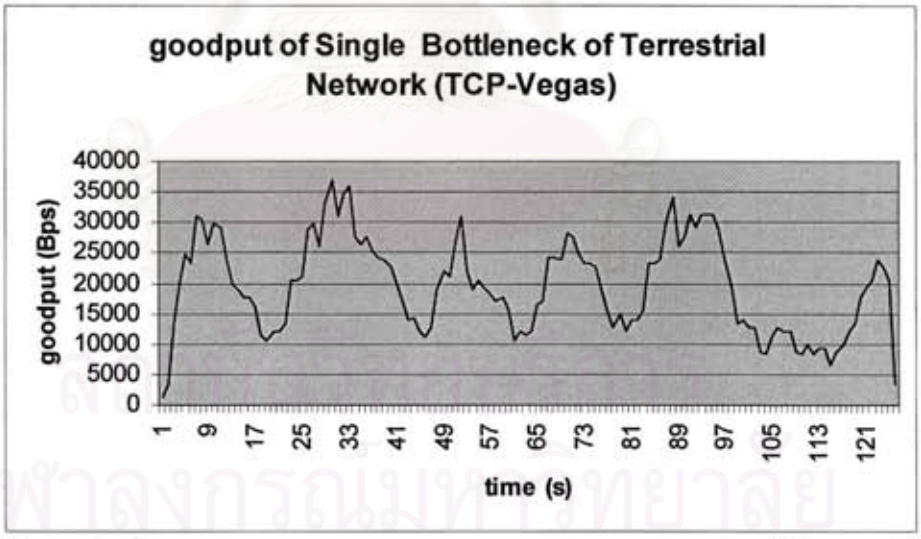


รูปที่ 4.17 ค่า goodput ของ TCP-Peach ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายภาคพื้นดิน

จากรูปที่ 4.18 แสดงค่า cwnd ของ TCP-Vegas ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายภาคพื้นดิน อธิบายได้ว่าในกระบวนการ congestion avoidance การเพิ่มขึ้นของ cwnd จะเพิ่มขึ้นเมื่อค่า *Diff* มีค่าน้อยกว่า α หากมีค่ามากกว่า β ให้ลดค่า *cwnd* และหากอยู่ระหว่างค่า α และ β ให้คงค่า *cwnd* ไว้ ส่วนช่วงที่เกิดยอดแหลมเกิดจากค่า cwnd มีค่าเท่ากับค่า *maxcwnd*



รูปที่ 4.18 ค่า *cwnd* ของ TCP-Vegas ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายภาคพื้นดิน

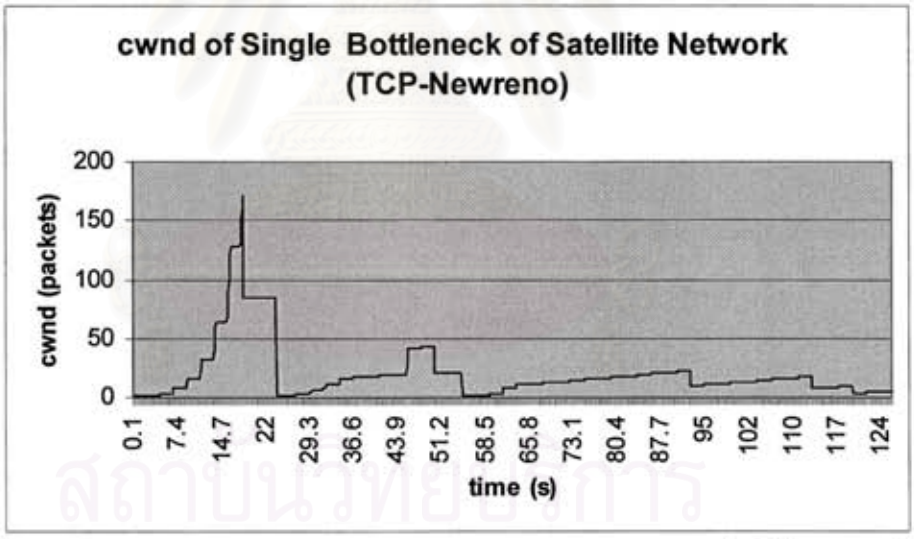


รูปที่ 4.19 ค่า goodput ของ TCP-Vegas ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายภาคพื้นดิน

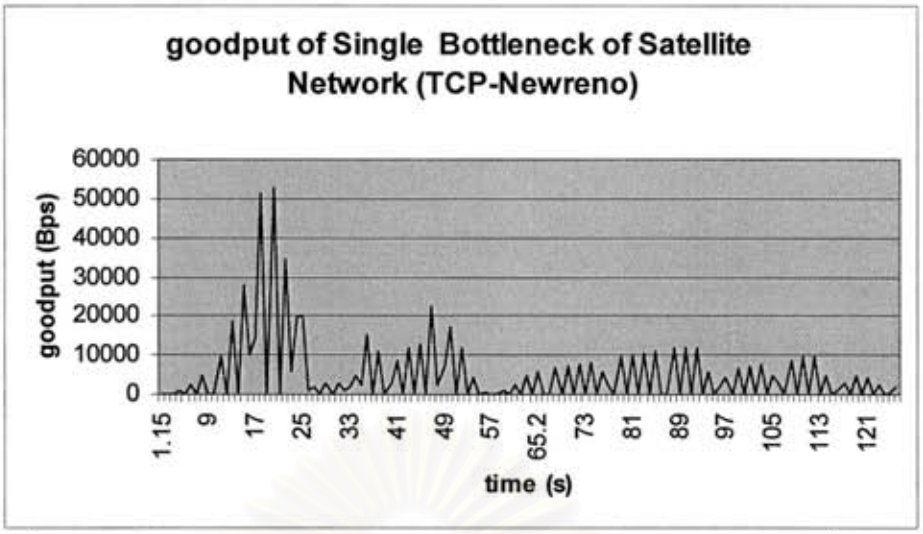
4.1.2.2 สภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายดาวเทียม (single bottleneck scenario of Satellite Network)

จากรูปที่ 4.20 แสดงค่า *cwnd* ของ TCP-Newreno ของแบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายดาวเทียม สังเกตุว่าจะมีค่าต่ำเมื่อเปรียบเทียบกับโครงข่ายภาคพื้นดิน เมื่อเปรียบเทียบค่า goodput เฉลี่ยระหว่าง TCP-Newrreno ของแบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายภาคพื้นดินกับโครงข่ายดาวเทียม ค่า goodput ของโครงข่ายดาวเทียมจะมีค่าน้อยกว่าถึง 84.21%

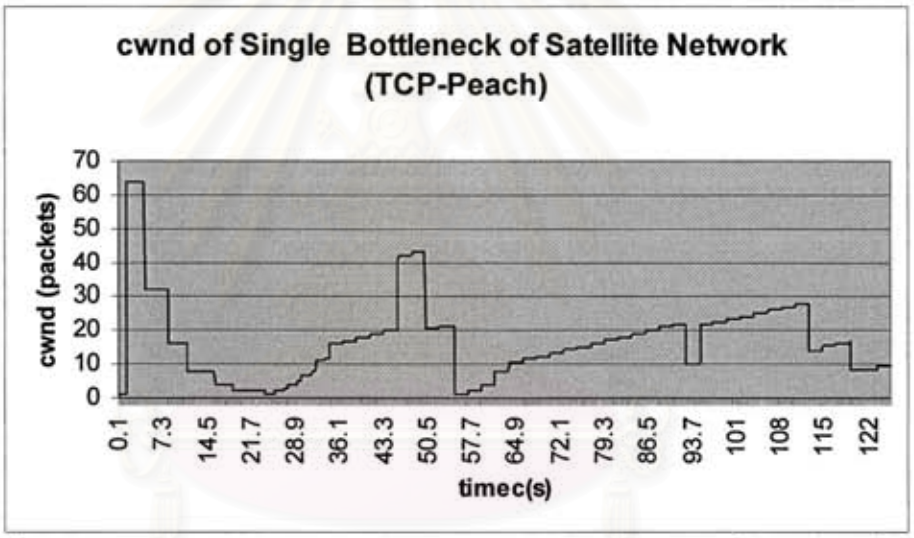
จากรูปที่ 4.22 แสดงค่า *cwnd* ของ TCP-Peach ของแบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายดาวเทียม สังเกตุว่าจะมีค่า *cwnd* ยังคงมีค่าต่ำเมื่อเปรียบเทียบกับโครงข่ายภาคพื้นดิน แต่เมื่อเปรียบเทียบค่า goodput เฉลี่ยระหว่าง TCP-Peach ที่แสดงไว้ในรูปที่ 35 กับ TCP-Newreno ของแบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายดาวเทียม ค่า goodput เฉลี่ยของ TCP-Peach จะมีค่ามากกว่าเท่ากับ 20.08%



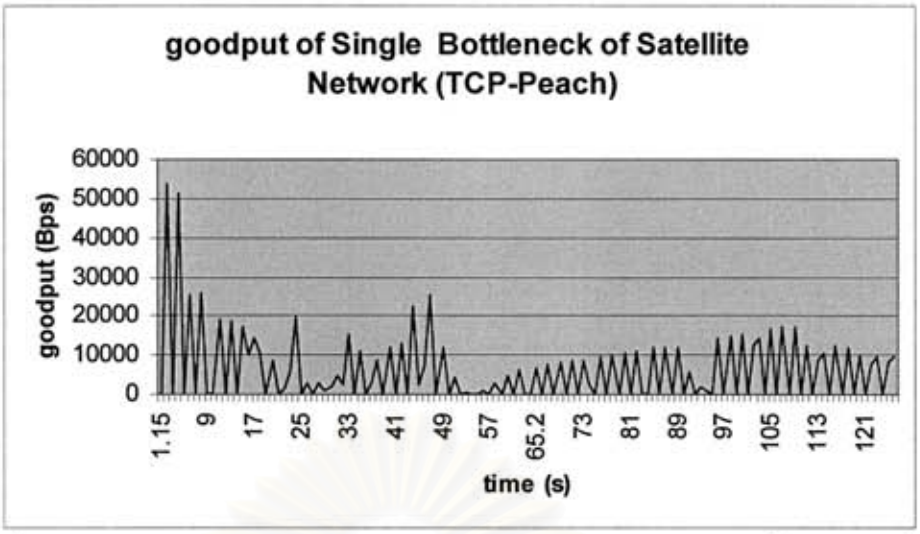
รูปที่ 4.20 ค่า *cwnd* ของ TCP-Newreno ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายดาวเทียม



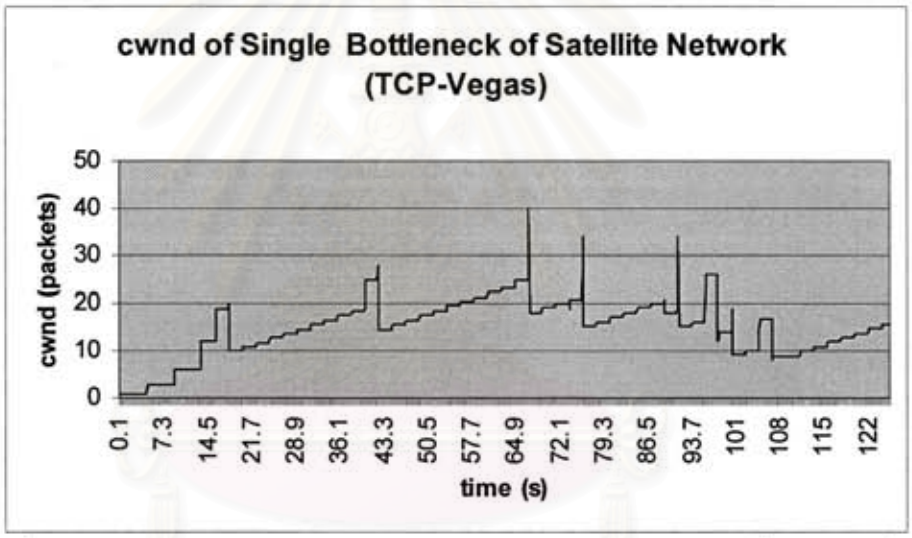
รูปที่ 4.21 ค่า goodput ของ TCP-Newreno ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายดาวเทียม



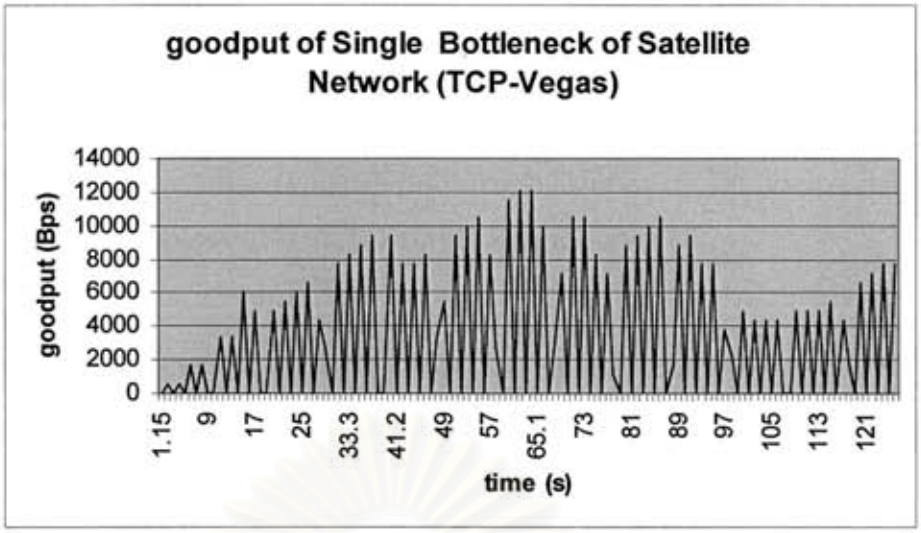
รูปที่ 4.22 ค่า *cwnd* ของ TCP-Peach ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายดาวเทียม



รูปที่ 4.23 ค่า goodput ของ TCP-Peach ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายดาวเทียม

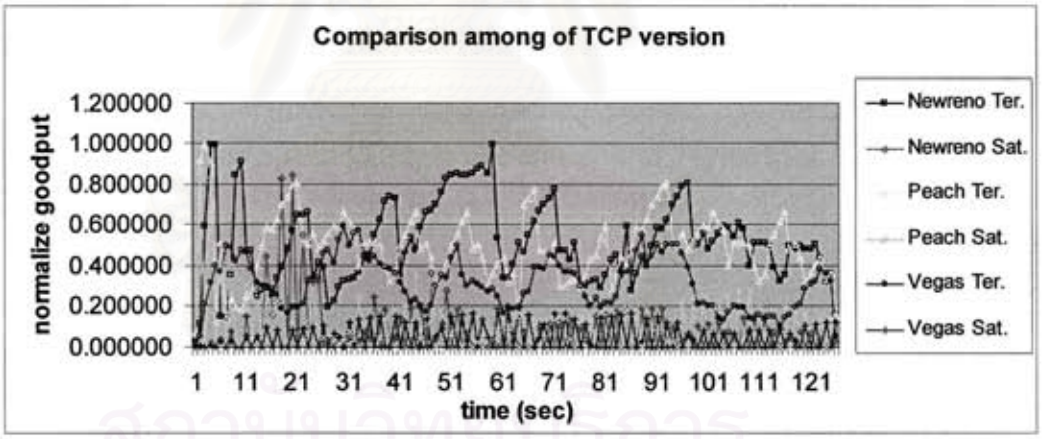


รูปที่ 4.24 ค่า *cwnd* ของ TCP-Vegas ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายดาวเทียม



รูปที่ 4.25 ค่า goodput ของ TCP-Vegas ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายดาวเทียม

จากรูปที่ 4.26 เปรียบเทียบค่า goodput ระหว่าง TCP-Newreno TCP-Peach และ TCP-Vegas บน แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียว โดยนำค่า goodput ที่มีค่ามากที่สุดในระบบมาหาร

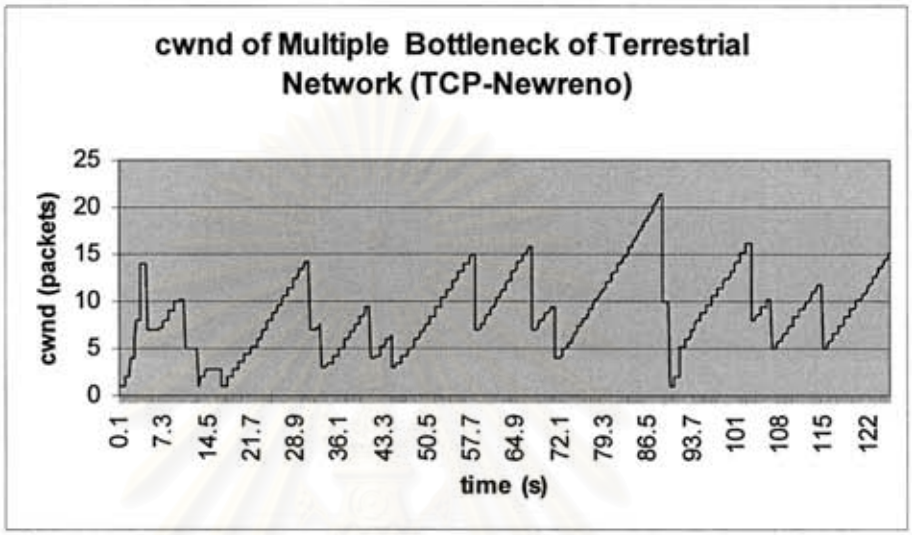


รูปที่ 4.26 เปรียบเทียบค่า goodput ระหว่าง TCP-Newreno TCP-Peach และ TCP-Vegas บน แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียว

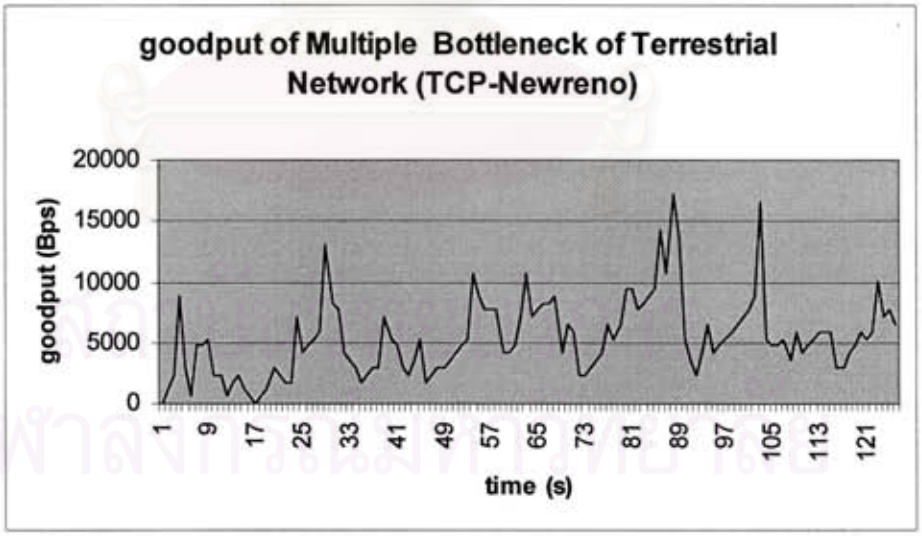
4.1.3 สภาพแวดล้อมที่มีความคับคั่งหลายจุด (multiple bottleneck scenario)

4.1.3.1 สภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายภาคพื้นดิน (multiple bottleneck scenario of Terrestrial Network)

ผลการจำลองสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายภาคพื้นดินจะเหมือนกับผลของการใช้งานจริงบนโครงข่ายภาคพื้นดิมากที่สุด

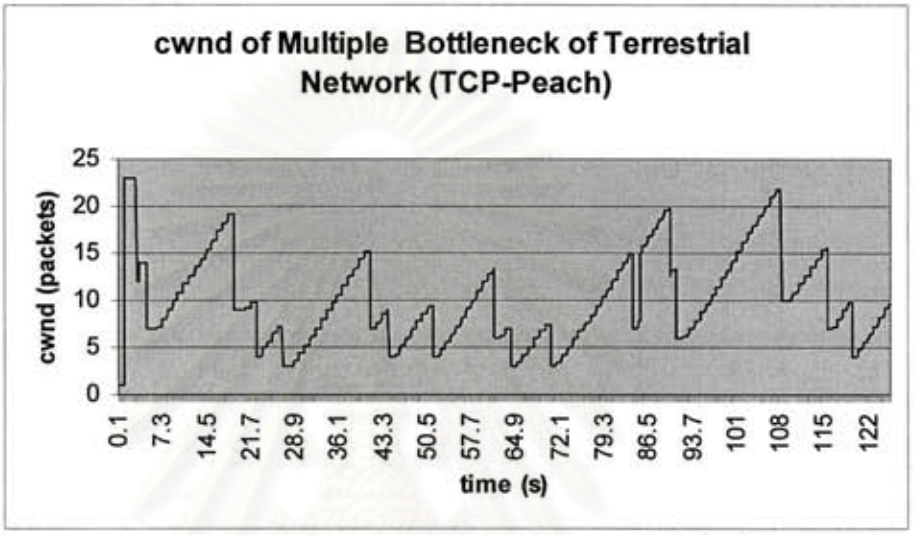


รูปที่ 4.27 ค่า *cwnd* ของ TCP-Newreno ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายภาคพื้นดิน

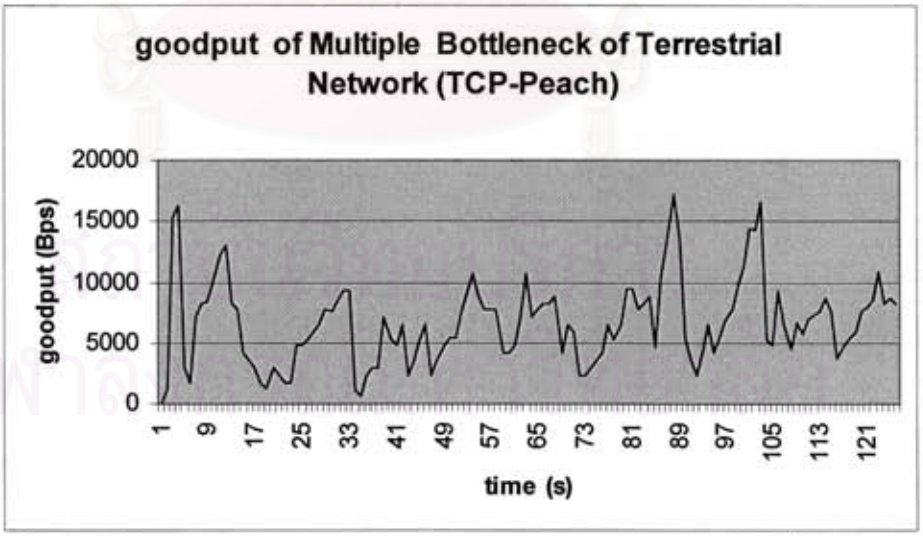


รูปที่ 4.28 ค่า goodput ของ TCP-Newreno ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายภาคพื้นดิน

จากรูปที่ 4.29 แสดงค่า cwnd ของ TCP-Peach ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายภาคพื้นดิน สังเกตได้ว่าลักษณะกราฟที่ได้จะคล้ายกับของแบบจำลองของสภาพแวดล้อมที่มีความคับคั่งจุดเดียวบนโครงข่ายภาคพื้นดิน แต่จะมีการลดลงของค่า cwnd มากกว่า เพราะมีจุดเสี่ยงการเกิดความคับคั่งมากกว่านั้นเอง เมื่อเปรียบเทียบค่า goodput เฉลี่ยระหว่าง TCP-Peach กับ TCP-Newreno ของแบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายภาคพื้นดิน จะได้ว่า TCP-Peach จะมีค่า goodput เฉลี่ยมากกว่าเท่ากับ 4.88%

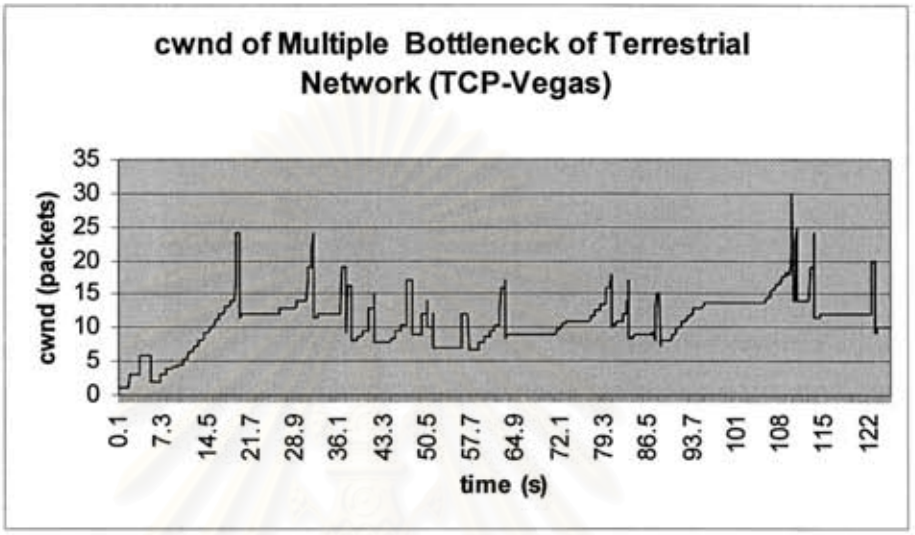


รูปที่ 4.29 ค่า *cwnd* ของ TCP-Peach ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายภาคพื้นดิน

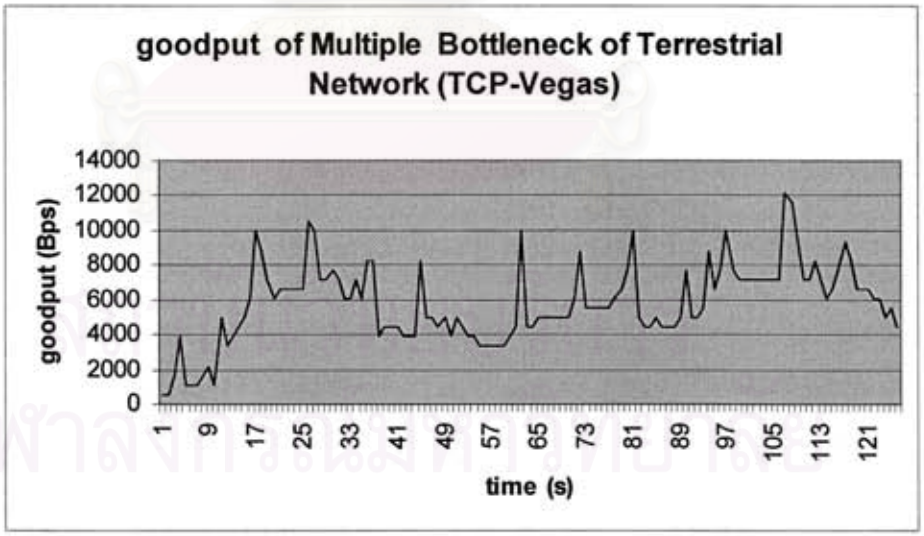


รูปที่ 4.30 ค่า goodput ของ TCP-Peach ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายภาคพื้นดิน

จากรูปที่ 4.31 แสดงค่า cwnd ของ TCP-Vegas ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายภาคพื้นดิน เมื่อเปรียบเทียบค่า goodput เฉลี่ยระหว่าง TCP-Vegas กับ TCP-Newreno ของแบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายภาคพื้นดิน จะได้ว่า TCP-Veags จะมีค่า goodput เฉลี่ยมากกว่าเท่ากับ 7.36%



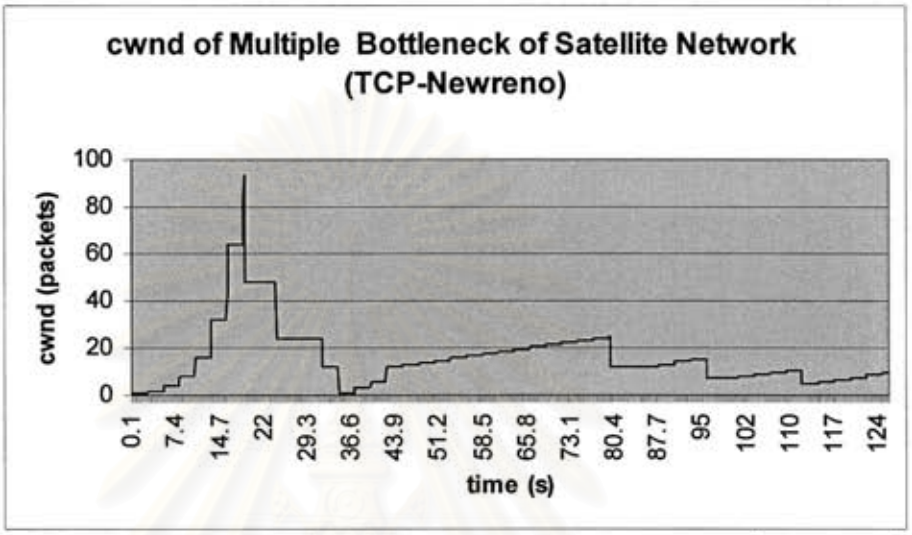
รูปที่ 4.31 ค่า *cwnd* ของ TCP-Vegas ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายภาคพื้นดิน



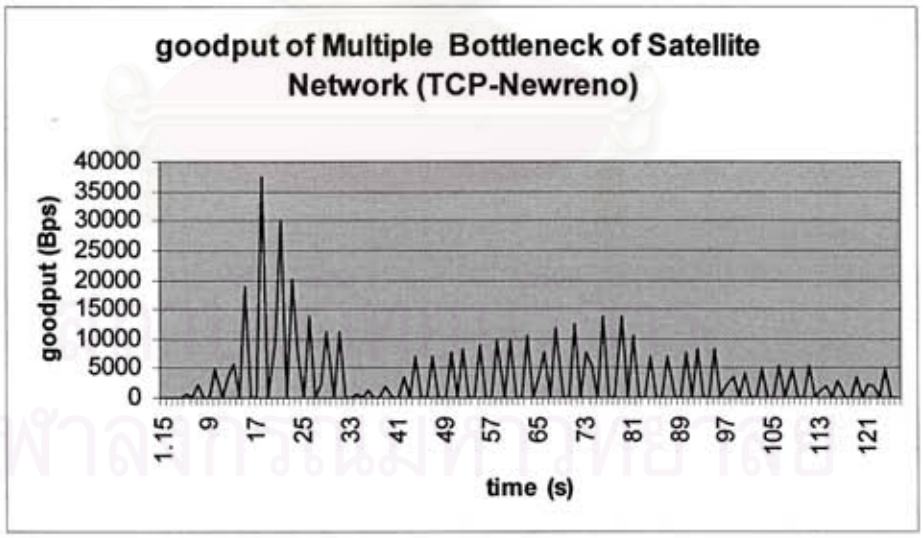
รูปที่ 4.32 ค่า goodput ของ TCP-Vegas ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายภาคพื้นดิน

4.1.3.2 สภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายดาวเทียม (multiple bottleneck scenario of Satellite Network)

จะเห็นว่าแบบจำลองนี้ จะมีค่า goodput ที่ต่ำกว่าแบบจำลองทุกแบบที่กล่าวมา เพราะว่ามีค่า RTT มากที่สุด อีกทั้งยังมีโอกาสการเกิดความเสียหายของแพ็กเกตสูงสุดด้วย จากรูปที่ 4.29 จะเห็นว่าช่วงเวลาของกระบวนการ slow start ใช้เวลาถึง 18.8 วินาที

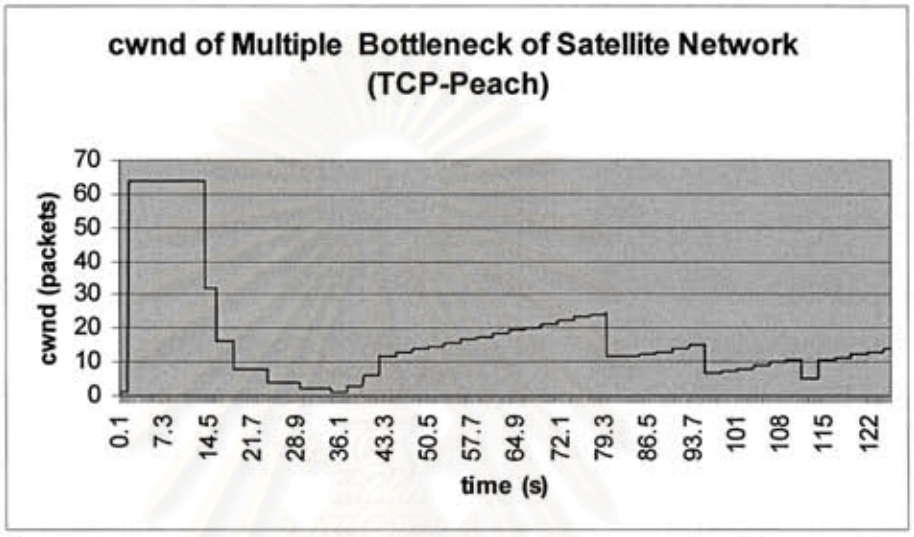


รูปที่ 4.33 ค่า *cwnd* ของ TCP-Newreno ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายดาวเทียม

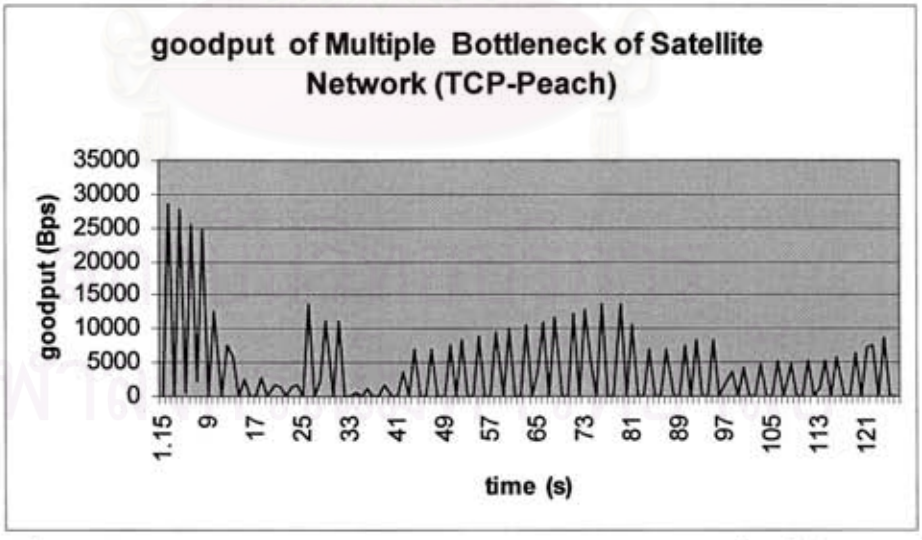


รูปที่ 4.34 ค่า goodput ของ TCP-Newreno ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายดาวเทียม

จากรูปที่ 4.31 จะแสดงให้เห็นถึงข้อดีของ TCP-Peach ที่สามารถทำให้ค่า goodput ในช่วงแรกของการส่งมีค่าสูงได้จนถึงวินาทีที่ 14.1 ค่า goodput จึงลดลงเพราะเกิดความคับคั่งขึ้น เมื่อเปรียบเทียบค่า goodput เฉลี่ยระหว่าง TCP-Peach กับ TCP-Newreno ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายดาวเทียม จะได้ว่า TCP-Peach มีค่า goodput เฉลี่ยมีค่ามากกว่า TCP-Newreno เท่ากับ 8.34%

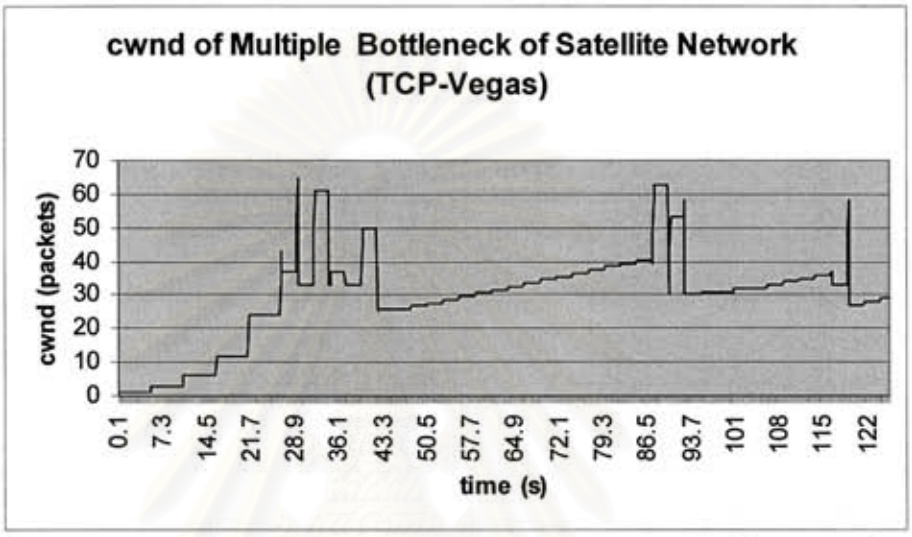


รูปที่ 4.35 ค่า *cwnd* ของ TCP-Peach ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายดาวเทียม

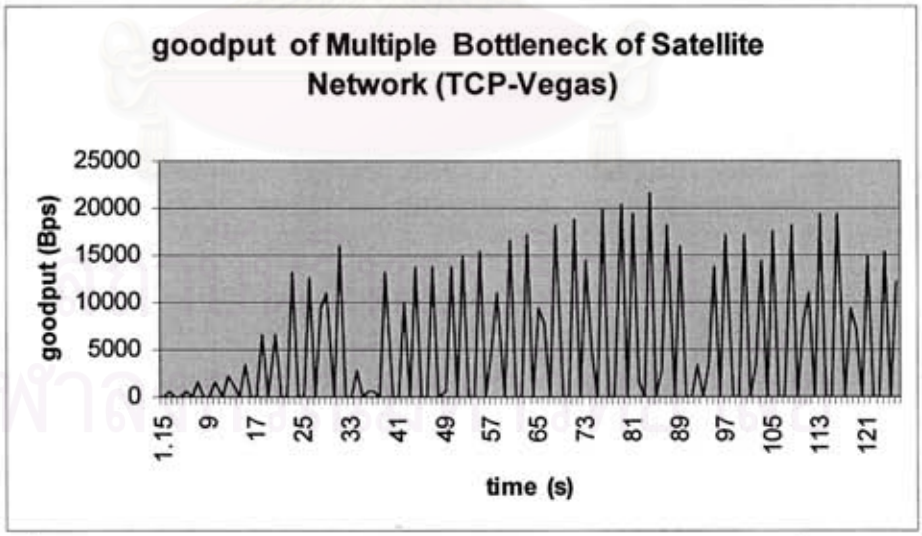


รูปที่ 4.36 ค่า goodput ของ TCP-Peach ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายดาวเทียม

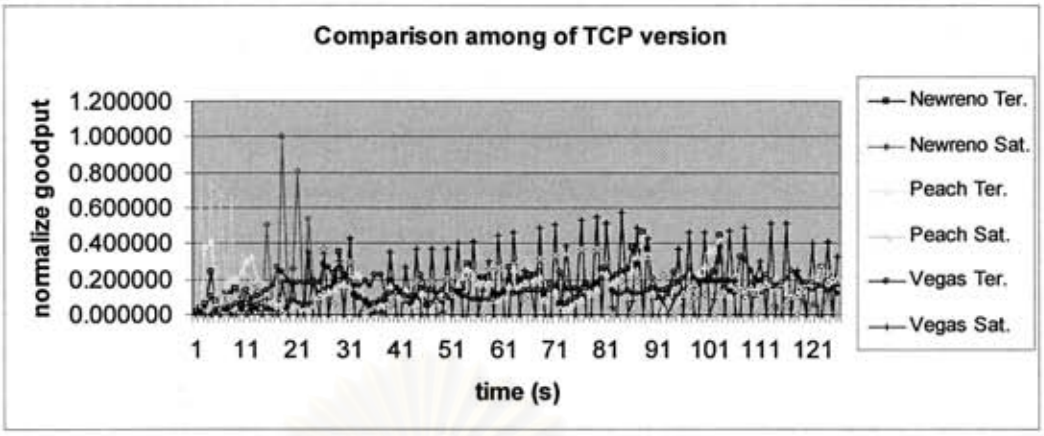
จากรูปจะเห็นว่า TCP-Vegas มีค่า goodput ในช่วงเริ่มต้นของการส่งไม่สูงนัก แต่จะมีค่าสูงขึ้นหลังออกจากระบวนการ slow start เมื่อเปรียบเทียบค่า goodput เฉลี่ยระหว่าง TCP-Vegas กับ TCP-Newreno ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายดาวเทียม จะได้ว่า TCP-Vegas มีค่า goodput เฉลี่ยมีค่ามากกว่า TCP-Newreno เท่ากับ 34.5%



รูปที่ 4.37 ค่า *cwnd* ของ TCP-Vegas ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายดาวเทียม



รูปที่ 4.38 ค่า goodput ของ TCP-Vegas ของ แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายดาวเทียม



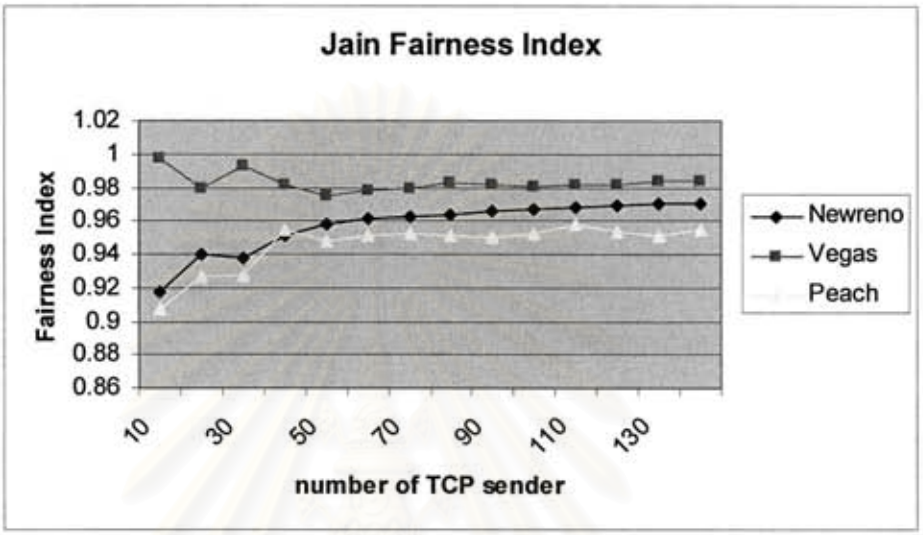
รูปที่ 4.39 เปรียบเทียบค่า goodput ระหว่าง TCP-Newreno TCP-Peach และ TCP-Vegas บนแบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายเดียว

ตารางที่ 4.1 สรุปค่า goodput เฉลี่ยของ TCP-Newreno TCP-Peach และ TCP-Vegas บนสภาพแวดล้อมต่างๆ

	Mean Goodput of Single TCP Connection	Percentage compare with TCP-Newreno	Mean Goodput of Single Bottleneck	Percentage compare with TCP-Newreno	Mean Goodput of Muti Bottleneck	Percentage compare with TCP-Newreno
TCP-Newreno on Terrestrial Network	61262.92	null	32231.43	null	5361.21	null
TCP-Newreno on Satellite Network	3246.92	null	5088.70	null	3397.27	null
TCP-Peach on Terrestrial Network	61423.47	0.26	30547.28	-5.51	5636.35	4.88
TCP-Peach on Satellite Network	4302.53	24.53	6367.53	20.08	3706.39	8.34
TCP-Vegas on Terrestrial Network	61320.19	0.09	29486.48	-9.31	5787.24	7.36
TCP-Vegas on Satellite Network	4665.71	30.41	5237.44	2.84	5187.05	34.50

4.2. ผลการวัดค่า fairness

จากรูปที่ 51 เปรียบเทียบค่า Fairness ระหว่าง TCP-Newreno TCP-Peach และ TCP-Vegas บน แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายดาวเทียม สามารถหาค่าได้จาก สมการที่ 7 ผลที่ได้แสดงให้เห็นว่าการจัดสรรการใช้แบนด์วิดท์ของ TCP-Vegas จะดีที่สุดใน TCP ทั้งสามรุ่นนี้ รองลงมาคือ TCP-Newreno และสุดท้ายคือ TCP-Peach



รูปที่ 4.40 เปรียบเทียบค่า Fairness ระหว่าง TCP-Newreno TCP-Peach และ TCP-Vegas บน แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายเดียวบนโครงข่ายดาวเทียม

3. ผลการวัดค่า friendliness

จากตารางที่ 4.2 เปรียบเทียบค่า Freindliness ระหว่าง TCP-Newreno TCP-Peach และ TCP-Vegas บน แบบจำลองของสภาพแวดล้อมที่มีความคับคั่งหลายจุดบนโครงข่ายดาวเทียม สามารถหา จำลองโครงข่ายเหมือนโครงข่ายในรูปที่ 3.5 โดยกำหนดให้ N มีค่าเท่ากับ 20 และกำหนดให้ จำนวนผู้ส่ง TCP มีค่าเหมือนในตารางที่ 4.1 แล้วหาค่า goodput เฉลี่ยของ TCP รุ่นต่าง TCP ที่มี friendliness ที่ดีต้องสามารถทำงานกับ TCP ต่างรุ่นได้ดี โดยต้องมีค่า goodput เฉลี่ยใกล้เคียงกับค่า แบนด์วิดท์ที่น้อยที่สุดในโครงข่ายหารด้วยจำนวนผู้ส่ง TCP ทั้งหมด เราจะเรียกค่านี้ว่าค่า "Fair Share" ในการจำลองนี้เรากำหนดให้แบนด์วิดท์ของช่องสัญญาณที่มีค่าน้อยที่สุดเท่ากับ 20 Mbps ดังนั้นค่า Fair Share จึงมีค่าเท่ากับ 1 Mbps

ตารางที่ 4.2 เปรียบเทียบค่า freinedliness ของ TCP-Newreno TCP-Peach และ TCP-Vegas

No. of Newreno	No. of Peach	No. of Vegas	Mean Goodput of Newreno (bps)	Mean Goodput of Peach (bps)	Mean Goodput of Vegas (bps)
6	7	7	989.92	1003.76	903.39
10	5	5	948.10	925.55	906.46
5	10	5	883.43	1104.35	857.71
5	5	10	914.65	956.23	932.28

บทที่ 5

สรุปผลการวิจัย อภิปรายผลและข้อเสนอแนะ

จากผลการทดลองสามารถสรุปในส่วนต่างๆ ได้ดังนี้

1. การใช้ TCP-Newreno, TCP-Peach และ TCP Vegas บนโครงข่ายภาคพื้นดิน จะให้ค่า goodput ที่มีค่าไม่แตกต่างกันมากนัก เนื่องจากในสภาพแวดล้อมนี้มีค่าการสูญหายของแพ็กเกตและค่าเวลาไปกลับน้อยเมื่อเทียบการเวลาไปกลับบนโครงข่ายดาวเทียม และเมื่อพิจารณาบนโครงข่ายดาวเทียม ในช่วงต้นของการส่งข้อมูล TCP-Peach จะมีค่า goodput ที่ดีกว่า TCP-Newreno และ TCP-Vegas ด้วยกระบวนการ sudden start แต่ในช่วงกลางของการส่งข้อมูล TCP-Peach และTCP-Newreno จะมีค่าไม่แตกต่างกันมากนัก จะมีความแตกต่างกันก็ต่อเมื่อเมื่อมีการสูญหายของแพ็กเกตเกิดขึ้น TCP-Peach จะมีจากกระบวนการ rapid recovery จึงทำให้ค่า goodput มีค่าไม่ลดลงเพราะในกระบวนการ rapid recovery จะมีการส่ง dummy segment ไปประเมินหาความคับคั่งของโครงข่ายอยู่ตลอดเวลา จึงทำให้ทราบว่าเกิดความคับคั่งในโครงข่ายจริงหรือไม่ หรือเกิดจากความเสียหายของแพ็กเกต หากเกิดจากความเสียหายของแพ็กเกตก็ไม่จำเป็นต้องลดอัตราการส่งข้อมูล ส่วน TCP-Vegas ในช่วงเวลากลางระหว่างการส่งข้อมูลจะมีค่า goodput ที่ดีที่สุด

2. เมื่อพิจารณาค่า fairness TCP-Vegas จะให้ค่า fairness index ดีที่สุด รองลงมาคือ TCP-Newreno และสุดท้ายคือ TCP-Peach แต่เมื่อพิจารณาค่า friendliness TCP-Peach และ TCP-Newreno จะให้ค่าที่ดีกว่า TCP-Vegas

3. ข้อดีของ TCP-Newreno คือง่ายต่อการสร้างและสามารถใช้บนโครงข่ายภาคพื้นดินได้ดี ส่วนข้อเสียคือเมื่อใช้งานบนโครงข่ายดาวเทียมค่า goodput จะลดลงมาก

4. ข้อดีของ TCP-Peach คือในช่วงแรกของการส่งข้อมูลจะให้ค่า goodput ที่สูง จึงเหมาะกับการนำไปใช้ในงานที่มีการส่งข้อมูลทีละน้อยๆ ส่วนข้อเสียคือการสร้างมีความซับซ้อน เพราะต้องใช้โพรโตคอลในชั้น IP ในการบรรจุ dummy segment และในบางสถานการณ์ TCP-Newreno จะให้ค่า goodput ที่สูงกว่าในการใช้งานบนโครงข่ายภาคพื้นดิน

5. ข้อดีของ TCP-Vegas คือเหมาะกับการใช้งานในสภาพแวดล้อมที่มีค่า *RTT* สูงหรือมีค่า *BER* สูง และมีค่าเฉลี่ยของ goodput ที่สูงทั้งบนโครงข่ายภาคพื้นดินและโครงข่ายดาวเทียม โดยเฉพาะในโครงข่ายดาวเทียมที่สามารถทำ goodput ได้สูงกว่า TCP-Newreno และ TCP-Peach แต่ TCP-Vegas มีข้อเสียคือในช่วงของ

กระบวนการ slow start จะมี goodput ที่ต่ำ จึงไม่เหมาะกับการใช้งานที่ส่งข้อมูลที่ละน้อยๆ เช่น การใช้งานด้วยโพรโตคอล http บนอินเตอร์เน็ต

จากการทดลองแสดงให้เห็นว่าแบบแผนควบคุมความคับคั่งแต่ละแบบจะมีข้อดีและข้อเสียแตกต่างกันไป เมื่อใช้บนสภาพแวดล้อมบนโครงข่ายดาวเทียม TCP-Vegas จะเป็นแบบแผนควบคุมความคับคั่งที่ดีที่สุด

ข้อเสนอแนะ

สำหรับงานที่ควรได้รับการศึกษาหรือพัฒนาต่อไปในอนาคต

1. ศึกษาการแปลงโพรโตคอล TCP ผ่านอุปกรณ์ส่งผ่านของโครงข่ายดาวเทียม
2. ศึกษาและพัฒนาหาแบบแผนควบคุมความคับคั่งใหม่ที่เหมาะกับการสื่อสารผ่านดาวเทียม
3. ศึกษาสภาพแวดล้อมของโครงข่ายดาวเทียมในกรณีที่มีฝนตก
4. การศึกษาเพิ่มเติมในส่วนของอัตราความผิดพลาดของข้อมูล (Bit Error Rate) กับการแก้ปัญหาของความคับคั่ง
5. การศึกษาตัวดาวเทียม ระบบภายในตัวดาวเทียมซึ่งสามารถทำการวิจัยและพัฒนาในอนาคต

รายการอ้างอิง

1. Transmission Control Protocol. Internet RFC 791, 1981. Avialable from: http://www.ietf.org [2005,May 13]
2. W. Stevens, TCP slow start, congestion avoidance, fast retransmit and fast recovery algorithms. RFC 2001, 1997. Avialable from: http://www.ietf.org [2005,May 13]
3. M. Allman, D. Glover, and L. Sanchez, Enhancing TCP over satellite channels using standard mechanism. RFC 2488, May 1999. Avialable from: http://www.ietf.org [2005,May 13]
4. M. Allman et al., Ongoing TCP research related to satellites. RFC2760, February 2000. Avialable from: http://www.ietf.org [2005,May 13]
5. R. C. Durst, G. J. Miller, and E. J. Travis, TCP extensions for space communications. Proc. ACM Mobicom, (November 1996).
6. T. R. Henderson and R. H. Katz, Transport protocols for Internet-compatible satellite networks. IEEE J. Select. Areas Commun. 17 (February 1999) : 326 – 344.
7. V. Jacobson, Congestion avoidance and control. Proc. ACM SIGCOMM, (August 1988) : 314 – 329.
8 C. Metz, TCP over satellite… The final frontier. IEEE Internet Comput., (January/February 1999) : 76 – 80.
9. C. Partridge and T. J. Shepard, TCP/IP performance over satellite links. IEEE Network Mag. (September/October 1997) : 44 – 49.
10. I. F. Akyildiz, G. Morabito, and S. Palazzo, TCP-Peach: A new congestion control scheme for satellite IP networks. IEEE/ACM Trans. Networking. 9 (June 2001) : 307 – 321.
11. Luigi A. Grieco and Saverio Mascolo, Performance evaluation and comparison of Westwood+, New Reno, and Vegas TCP congestion control. ACM SIGCOMM Computer Communication Review. 34 (April 2004) : 25 - 38.
12. Kai Xu, Ye Tian, and Nirwan Ansari, TCP-Jersey for Wireless IP Communications. IEEE Journal on selected area in communications. 22 (May 2004) : 747 - 756.
13. Lawrence S., Sean W., and Larry L., TCP Vegas: for New Technique for Congestion Detection and Avoidance. ACM SIGCOMM Computer Communication Review. 28 (April 2002) : 11 - 23.

ประวัติผู้เขียนวิทยานิพนธ์

นายสนามชัย มาสุธน เกิดวันที่ 13 มกราคม พ.ศ. 2524 ที่จังหวัดเพชรบูรณ์ สำเร็จการศึกษาปริญญาวิศวกรรมศาสตรบัณฑิต สาขาวิชาวิศวกรรมไฟฟ้า จากคณะวิศวกรรมศาสตร์ มหาวิทยาลัยธรรมศาสตร์ ในปีการศึกษา 2544 หลังจากสำเร็จการศึกษาในระดับปริญญาวิศวกรรมศาสตรบัณฑิตได้เข้าทำงานในบริษัทเอกชนเป็นเวลา 1 ปี หลังจากนั้นได้เข้าศึกษาต่อในหลักสูตรวิศวกรรมศาสตรมหาบัณฑิต ภาควิชาวิศวกรรมไฟฟ้า สาขาวิชาวิศวกรรมไฟฟ้า ที่จุฬาลงกรณ์มหาวิทยาลัยในภาคต้น ปีการศึกษา 2546