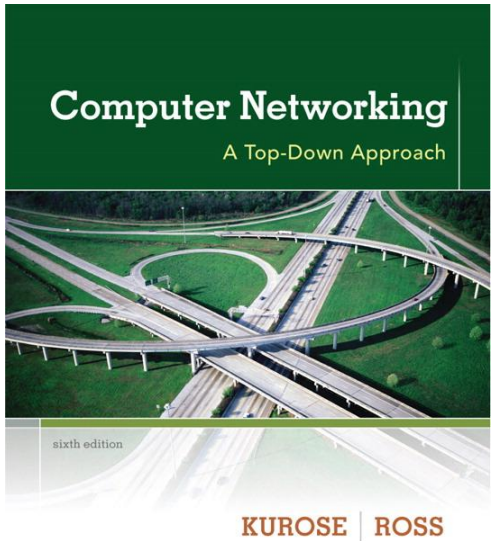
Chapter 3 Transport Layer



การใช้สไลด์ :
เนื้อหาในสไลด์เหล่านี้ถูกแปลมาจากสไลด์ต้นฉบับประกอบหนังสือของผู้แต่งชื่อ Kurose และ Ross

ผู้แปลอนุญาตให้ทุกท่านสามารถใช้สไลด์ทั้งหมดได้ ดังนั้นท่านสามารถดูภาพเคลื่อนไหว สามารถเพิ่ม ,แก้ไข และ ลบสไลด์ (นับรวมข้อความนี้) และเนื้อหาของสไลด์เพื่อให้เหมาะกับความต้องการของท่าน

สำหรับการแลกเปลี่ยน เราต้องการสิ่งต่อไปนี้เท่านั้น :

- ถ้าท่านใช้สไลด์เหล่านี้ (เป็นตัวอย่าง, ในห้องเรียน) อย่าลืมกล่าวถึงที่มาของสไลด์ (หลังจากนี้ เราต้องการให้ทุกคนอุดหนุนและใช้หนังสือของผู้แต่งด้านข้าง)
- ถ้าคุณโพสต์สไลด์ใด ๆ ในเวป, อย่าลืมกล่าวถึงว่า คุณแก้ไขจากสไลด์ต้นฉบับของเรา และ ระบุถึงลิขสิทธิ์ของเราด้วย

ขอขอบคุณและขอให้สนุก!
ณัฐนนท์ ลีลาตระกูล ผู้เรียบเรียง

Computer Networking: A Top Down Approach
6th edition
Jim Kurose, Keith Ross
Addison-Wesley
March 2012

Chapter 3: Transport Layer

เป้าหมาย:

- เข้าใจหลักการเบื้องหลังของการให้บริการชั้น Transport:
  - การรวม/การแยกข้อมูล (multiplexing, demultiplexing)
  - การส่งข้อมูลที่ไว้ใจได้ (reliable data transfer)
  - การควบคุมการไหลของข้อมูล (flow control)
  - การควบคุมความคับคั่งในเครือข่าย (congestion control)
- เรียนรู้เกี่ยวกับโปรโตคอลในชั้น Transport ของ Internet:
  - UDP: การส่งข้อมูลไม่ต้องมีการเชื่อมต่อ (connectionless transport)
  - TCP: การส่งข้อมูลที่ไว้ใจได้ แต่ต้องมีการเชื่อมต่อก่อน (connection-oriented reliable transport)
  - การควบคุมความคับคั่งของ TCP

Chapter 3 Outline

3.1 บริการในขั้น Transport

3.2 การรวมและการแยกข้อมูล (multiplexing and demultiplexing)

3.3 การส่งข้อมูลที่ไม่ต้องมีการเชื่อมต่อ (connectionless transport): UDP

3.4 หลักการต่าง ๆ ของการส่งข้อมูลที่ไว้ใจได้ (principles of reliable data transfer)

3.5 การส่งข้อมูลที่ต้องมีการเชื่อมต่อก่อน (connection-oriented transport): TCP

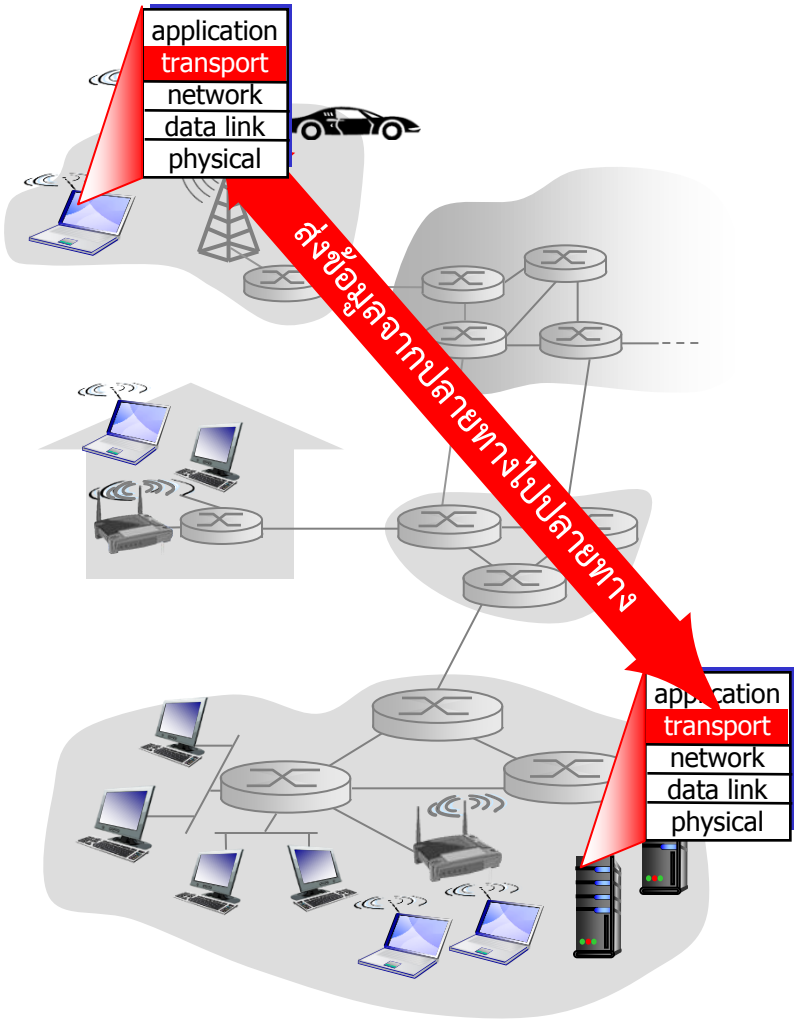
- โครงสร้างส่วนข้อมูล (segment)
- การส่งข้อมูลที่น่าเชื่อถือ/น่าไว้ใจได้ (reliable data transfer)
- การควบคุมการไหล (flow control)
- การจัดการการเชื่อมต่อ (connection management)

3.6 หลักการของการควบคุมความคับคั่ง (congestion control)

3.7 congestion control ของ TCP

Transport services and protocols

- ให้บริการการสื่อสารที่เชื่อม (เชิง *logic*) ระหว่างโพรเซสของแอพพลิเคชันที่ทำงานอยู่บนเครื่องต่างๆ
- transport protocols ทำงานอยู่บนเครื่องปลายทาง
  - เครื่องปลายทางตัวส่ง : ทำการแยกข้อมูลออกเป็น segments แล้วส่งต่อไปยังชั้น network layer
  - เครื่องปลายทางตัวรับ : รวบรวม segments และประกอบให้เป็นข้อความแล้วส่งต่อไปยังชั้น app
- มีโปรโตคอลที่ให้บริการในการส่งข้อมูลมากกว่าหนึ่งโปรโตคอล
  - Internet: TCP and UDP



ชั้น Transport vs. ชั้น Network

- *ชั้น network:* การสื่อสาร (เชิงตรรกะ) ระหว่างเครื่อง
- *ชั้น transport:* การสื่อสาร (เชิงตรรกะ) ระหว่างโพรโซส
  - ใช้บริการด้วย และ ช่วยเสริมบริการของชั้น network

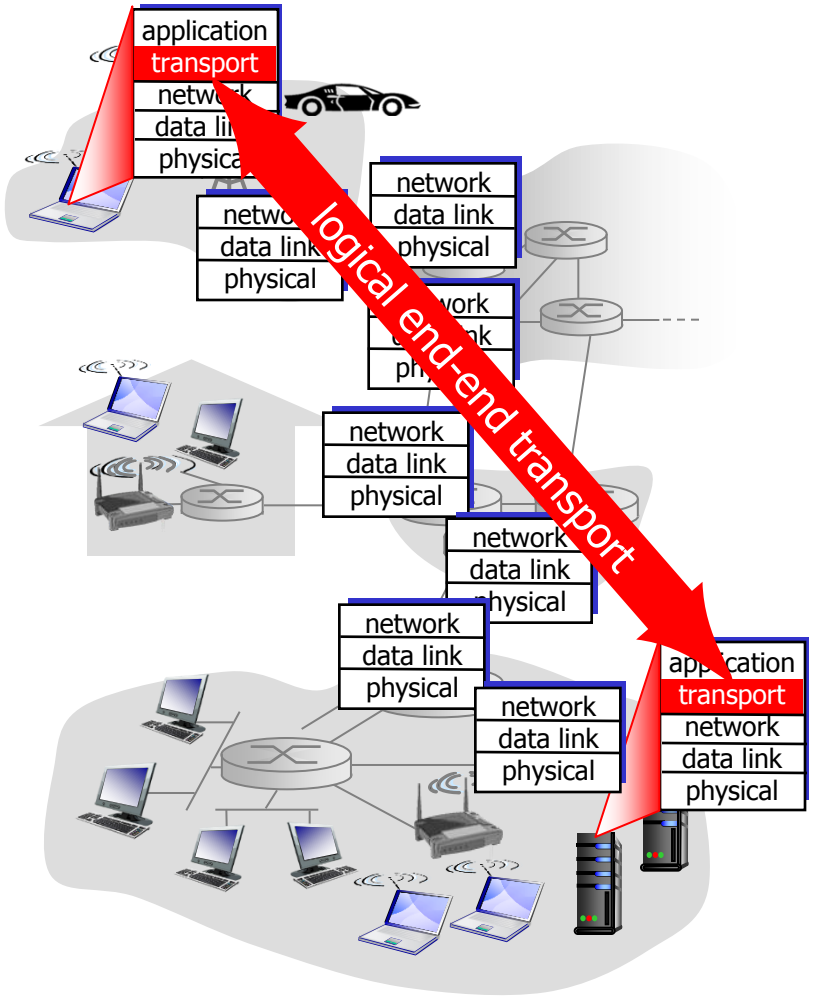
อุปมาเหมือนกับบ้าน

เด็ก 12 คนในบ้าน Ann ส่งจดหมาย 12 ฉบับไปหาเด็ก 12 คนในบ้านของ Bill:

- เครื่อง = บ้าน
- โพรเซส = เด็ก
- ข้อความของโปรแกรม = จดหมายในซอง
- โพรโตคอลชั้น transport = Ann และ Bill ที่ต้องแยกจดหมายให้ลูก ๆ
- โพรโตคอลชั้น network = บริการของไปรษณีย์

โปรโตคอลชั้นทรานสปอร์ตของอินเตอร์เน็ต

- ส่งข้อมูลแบบเรียงลำดับและน่าไว้ใจหรือน่าเชื่อถือได้ (TCP)
  - การควบคุมความคับคั่งของข้อมูลในเครือข่าย
  - การควบคุมการไหลของข้อมูล
  - ต้องมีการเชื่อมต่อก่อนการส่งข้อมูล
- ส่งข้อมูลแบบไม่เรียงลำดับและไม่น่าเชื่อถือ : UDP
  - ไม่มีอะไรเพิ่มเติมจากชั้น IP ที่เป็น “best-effort”
- ไม่ให้บริการเหล่านี้:
  - การรับประกันความล่าช้า
  - การรับประกันแบนด์วิดธ์



Chapter 3 Outline

3.1 บริการในขั้น Transport

3.2 การรวมและการแยกข้อมูล (multiplexing and demultiplexing)

3.3 การส่งข้อมูลที่ไม่ต้องมีการเชื่อมต่อ (connectionless transport): UDP

3.4 หลักการต่าง ๆ ของการส่งข้อมูลที่ไว้ใจได้ (principles of reliable data transfer)

3.5 การส่งข้อมูลที่ต้องมีการเชื่อมต่อก่อน (connection-oriented transport): TCP

- โครงสร้างส่วนข้อมูล (segment)
- การส่งข้อมูลที่น่าเชื่อถือ/น่าไว้ใจได้ (reliable data transfer)
- การควบคุมการไหล (flow control)
- การจัดการการเชื่อมต่อ (connection management)

3.6 หลักการของการควบคุมความคับคั่ง (congestion control)

3.7 congestion control ของ TCP

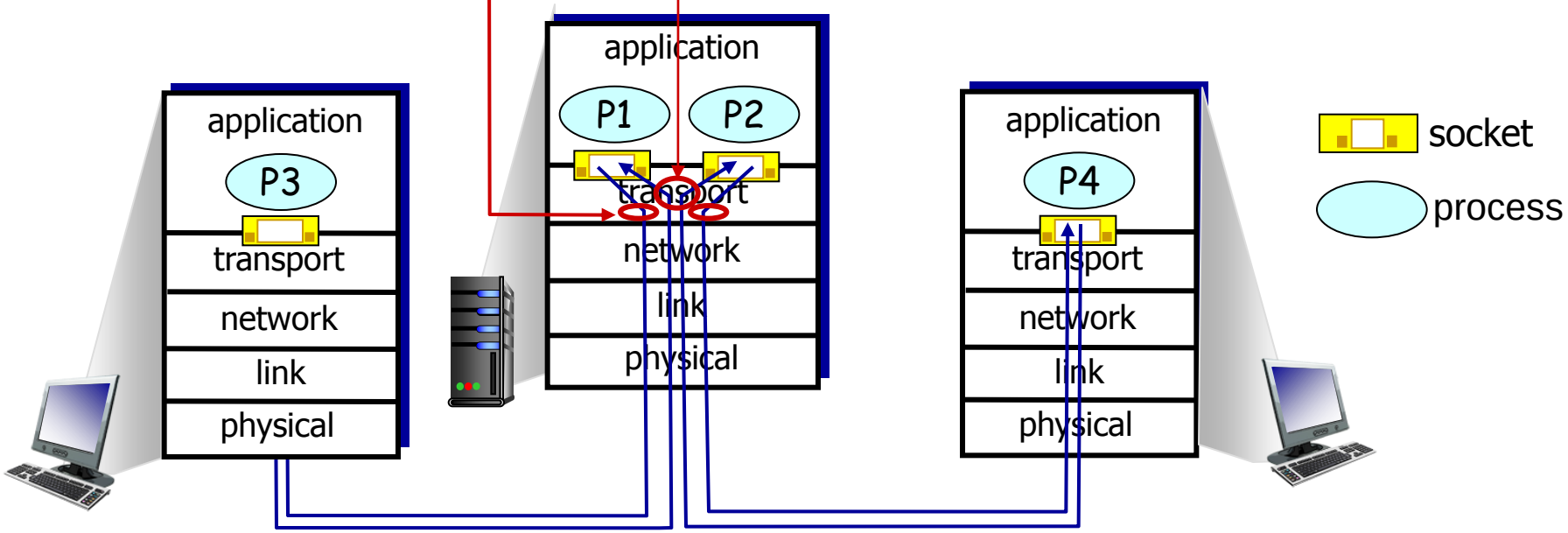
การรวมและแยกข้อมูล (Multiplexing/demultiplexing)

multiplexing ที่ผู้ส่ง:

จัดการกับข้อมูลจากหลาย ๆ socket, ใส่ header ของชั้น transport เข้าไป (header จะถูกใช้เพื่อทำการ demultiplexing ภายหลัง)

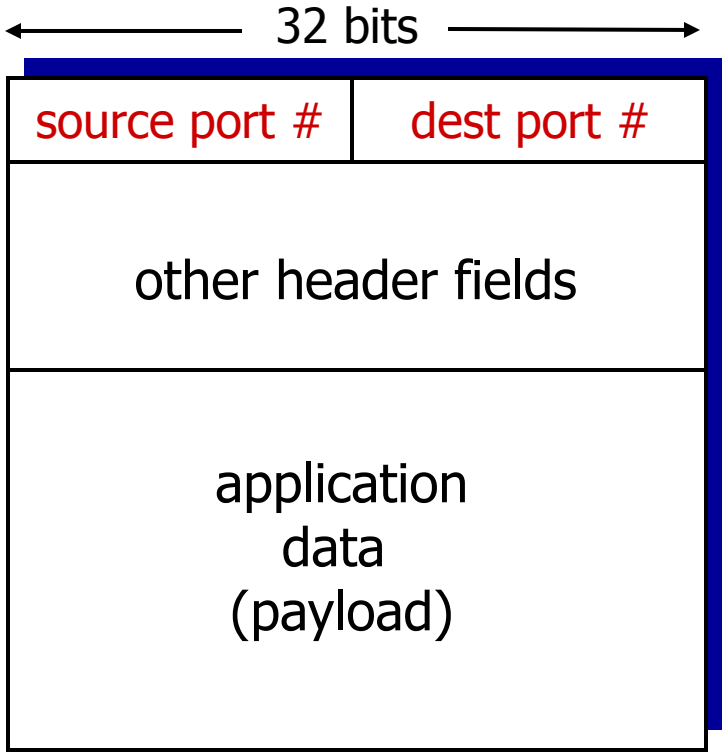
demultiplexing ที่ผู้รับ:

ใช้ข้อมูลส่วน header เพื่อส่ง segment ที่ได้มาไปยัง socket ที่ถูกต้อง



ทำ demultiplexing อย่างไร

- host รับดาต้าแกรมจากชั้น IP
  - แต่ละดาต้าแกรมจะมีหมายเลขไอพีต้นทาง, ไอพีปลายทาง
  - แต่ละดาต้าแกรมจะมี segment ของชั้น transportอยู่
  - แต่ละ segment จะมีหมายเลข port ต้นทางและปลายทาง (source and destination port number)
- host ใช้ข้อมูลหมายเลขไอพีและหมายเลขพอร์ต เพื่อนำเซ็กเมนต์ไปยัง socket ที่ถูกต้อง



TCP/UDP segment format

demultiplexing กรณีที่ไม่มีการเชื่อมต่อ (Connectionless)

- *จากบทที่แล้ว:*
  socket ที่ถูกสร้างจะมีหมายเลข port กำกับ:

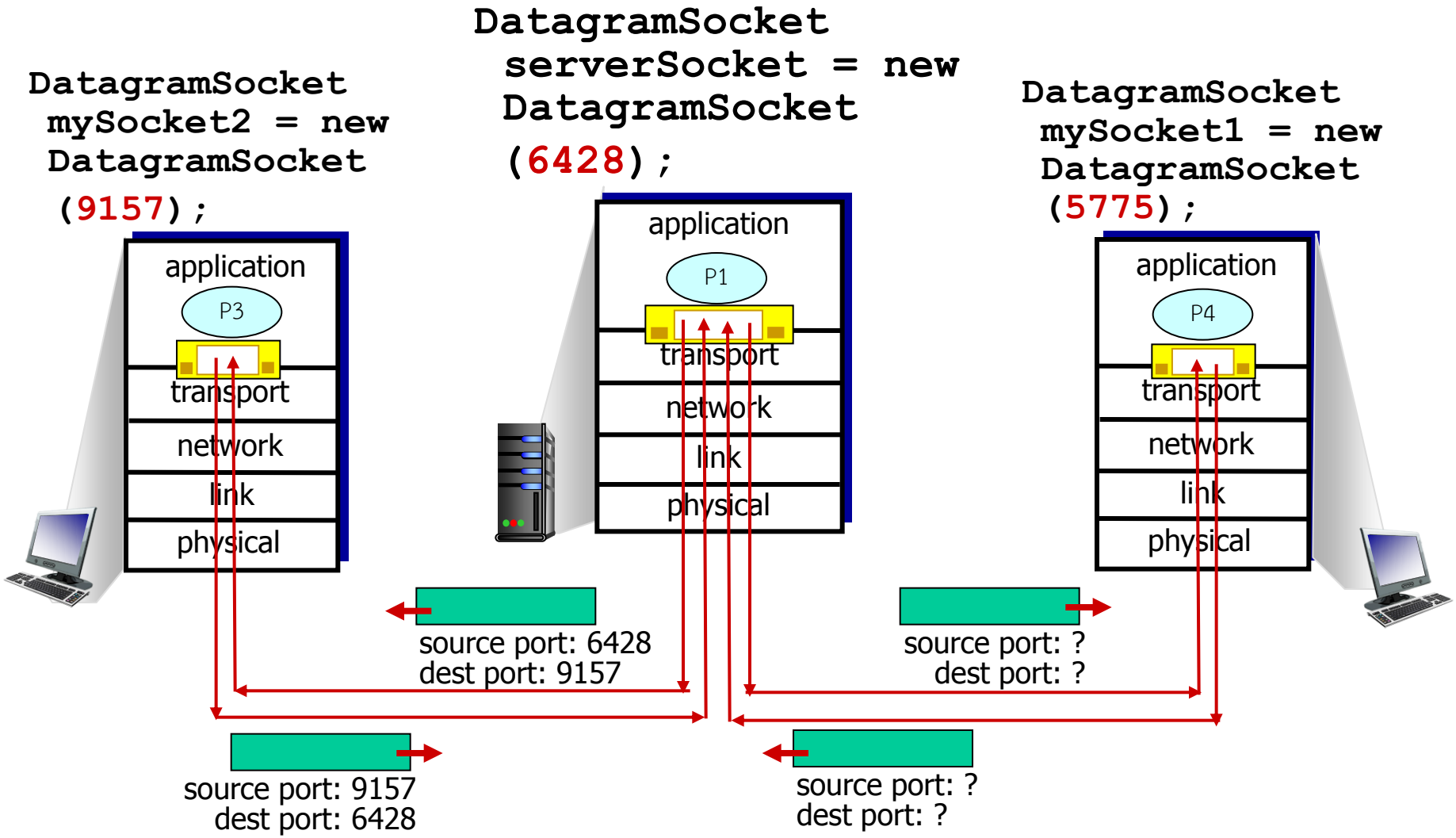
```
DatagramSocket mySocket1 = new
DatagramSocket(12534);
```

- เมื่อจะสร้างดาต้าแกรมเพื่อส่งไปยัง socket ที่เป็น UDP ต้องทำการระบุ
  - หมายเลขไอพีปลายทาง
  - หมายเลข port ปลายทาง

- เมื่อ host รับ segment ของ UDP:
  - ตรวจสอบหมายเลข port ใน segment
  - นำ segment ของ UDP ไปยัง socket ที่ผูกกับ port นั้น

datagrams ของ IP ที่มี *เลข port ปลายทางตัวเดียวกัน,* แม้จะมีที่อยู่ IP และ/หรือ เลข port ของเครื่องต้นทางต่างกัน (process ที่ส่ง datagram เป็นคนละ process แต่ส่งมาที่ process ผู้รับตัวเดียวกัน) จะถูกนำไปยัง *socket เดียวกัน* ที่เครื่องปลายทาง

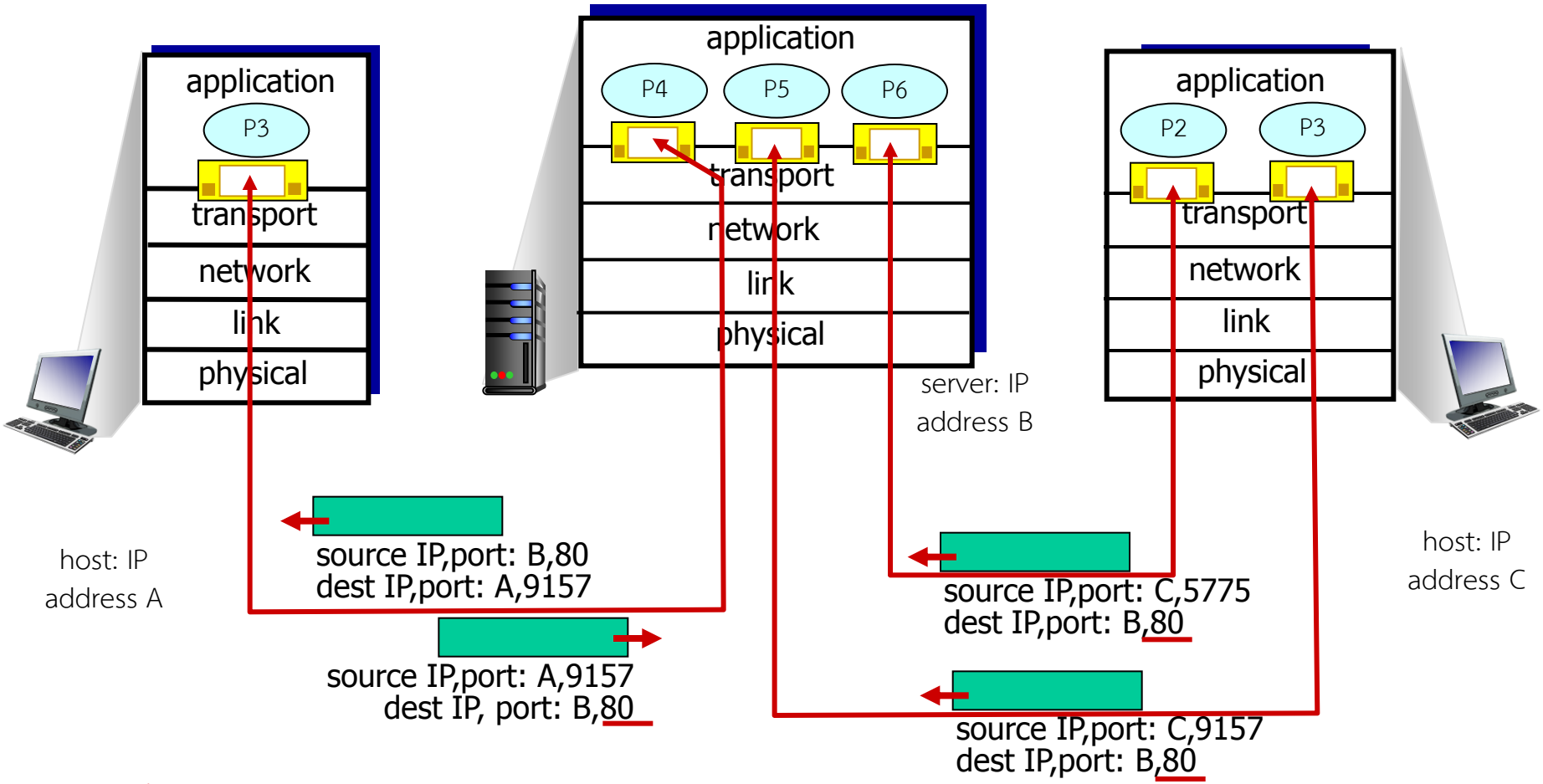
Connectionless demux: example



demux กรณีที่มีการเชื่อมต่อ (Connection-oriented)

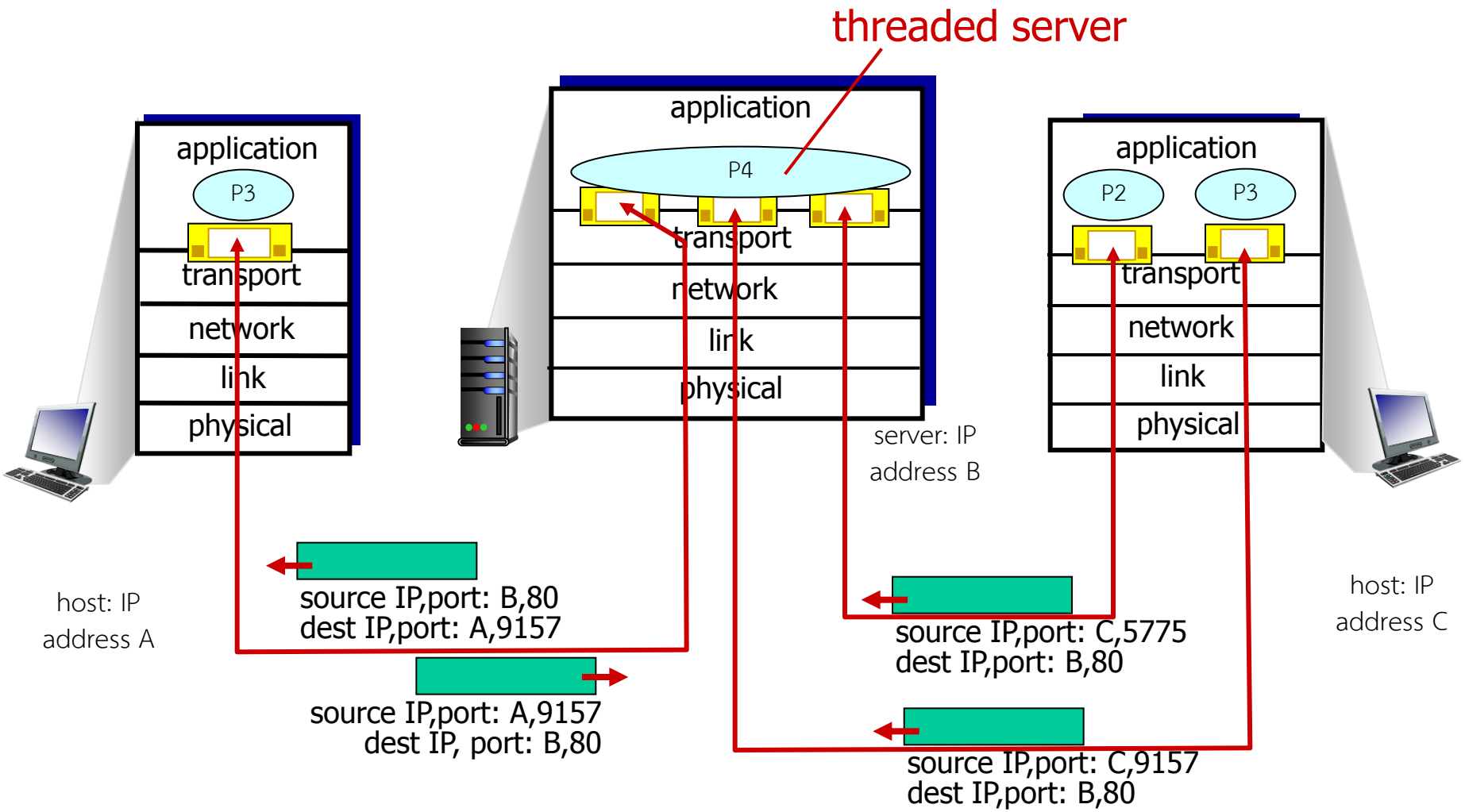
- socket แบบ TCP ถูกระบุด้วย 4 ส่วน ซึ่งมีองค์ประกอบดังนี้ :
  - source IP address
    - (หมายเลขไอพีต้นทาง)
  - source port number
    - (หมายเลขพอร์ตต้นทาง)
  - dest IP address
    - (หมายเลขไอพีปลายทาง)
  - dest port number
    - (หมายเลขพอร์ตปลายทาง)
- demux: จะดูค่าทั้งสี่ส่วนเพื่อนำ packet ไปยัง sockets ที่เหมาะสม
- เครื่องเซิร์ฟเวอร์อาจเปิด sockets แบบ TCP จำนวนมากพร้อมกัน:
  - ซึ่ง แต่ละ socket ถูกระบุด้วย 4 ส่วน (แต่จะมีส่วนที่ 3 คือหมายเลข IP ปลายทางเหมือน ๆ กัน)
- web servers จะมี socket หลาย ๆ socket ไว้สำหรับเชื่อมต่อกับแต่ละ client
  - HTTP แบบ non-persistent จะใช้ socket ที่แตกต่างกันสำหรับแต่ละ request

Connection-oriented demux: example



ทั้ง 3 segment มีที่อยู่ IP ปลายทาง คือ B และ เลขพอร์ตปลายทางคือ 80, แต่ถูกแยกส่งไปยัง socket ที่ต่างกัน

Connection-oriented demux: example



Chapter 3 Outline

3.1 บริการในขั้น Transport

3.2 การรวมและการแยกข้อมูล (multiplexing and demultiplexing)

3.3 การส่งข้อมูลที่ไม่ต้องมีการเชื่อมต่อ (connectionless transport): UDP

3.4 หลักการต่าง ๆ ของการส่งข้อมูลที่ไว้ใจได้ (principles of reliable data transfer)

3.5 การส่งข้อมูลที่ต้องมีการเชื่อมต่อก่อน (connection-oriented transport): TCP

- โครงสร้างส่วนข้อมูล (segment)
- การส่งข้อมูลที่น่าเชื่อถือ/น่าไว้ใจได้ (reliable data transfer)
- การควบคุมการไหล (flow control)
- การจัดการการเชื่อมต่อ (connection management)

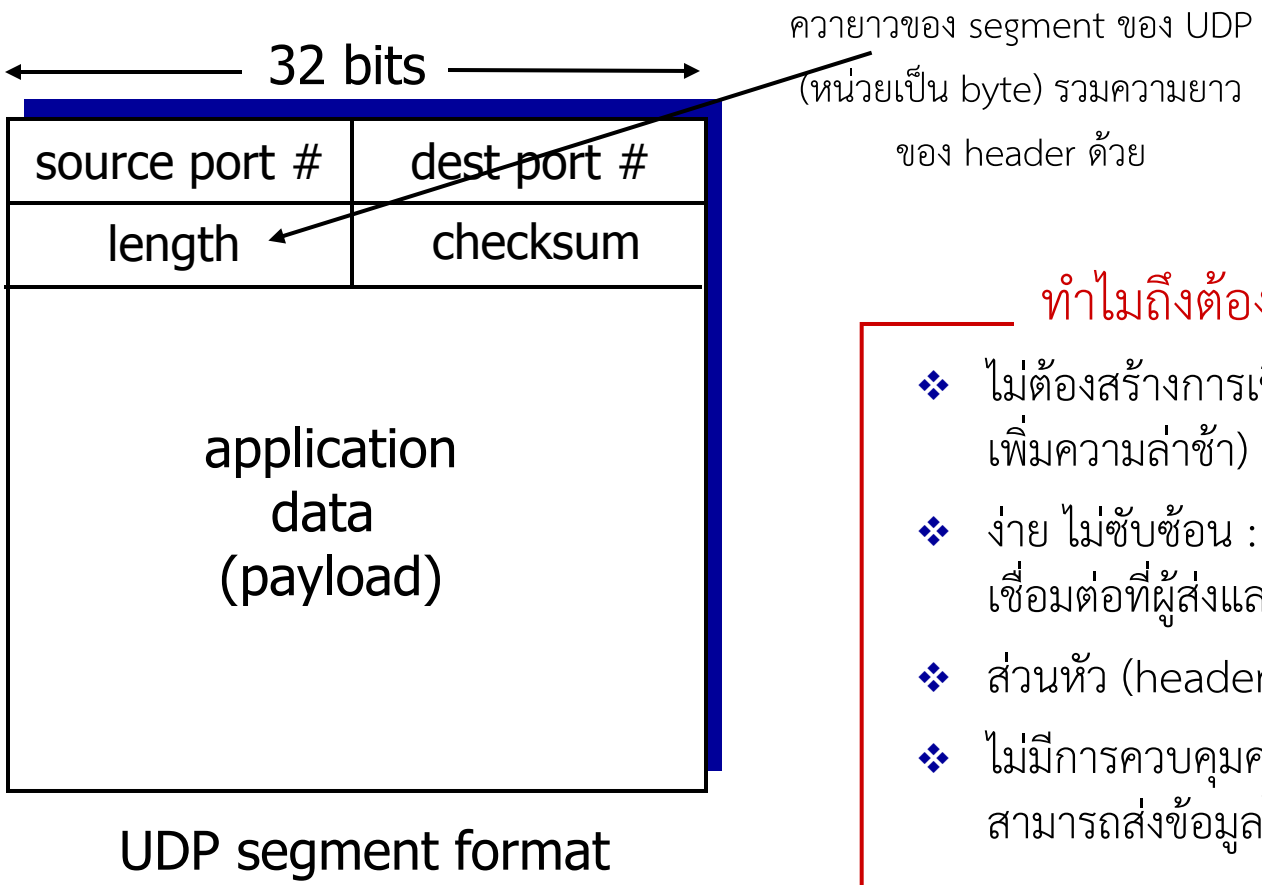
3.6 หลักการของการควบคุมความคับคั่ง (congestion control)

3.7 congestion control ของ TCP

UDP: User Datagram Protocol [RFC 768]

- เป็น protocol ของ Internet ที่ไม่ได้มีส่วนเพิ่มเติมอะไร
- บริการแบบ “จะพยายามให้ดีที่สุด” (best effort) ดังนั้น segment แบบ UDP อาจ:
  - สูญหาย
  - ไปถึง app ของผู้รับแต่ไม่เรียงลำดับ
- *ไม่มีการเชื่อมต่อ:*
  - ไม่มีการทำ handshake ระหว่างผู้ส่งและผู้รับ
  - แต่ละ segment ที่เป็น UDP จะถูกจัดการเป็นอิสระจาก segment อื่น ๆ
- UDP ใช้ใน:
  - app ที่ส่ง multimedia อย่างต่อเนื่อง (ซึ่งทนต่อการสูญเสีย แต่ sensitive กับอัตราการส่ง)
  - DNS
  - SNMP
- ถ้าจะให้ การถ่ายโอนข้อมูลบน UDP มีความน่าเชื่อถือกว่านี้:
  - เพิ่มความน่าเชื่อถือที่ชั้น application
  - มีการกู้คืนข้อผิดพลาด (ขึ้นอยู่กับ แต่ละ application)!

UDP: ส่วนหัวของ segment (segment header)



UDP segment format

ทำไมถึงต้องมี UDP?

- ไม่ต้องสร้างการเชื่อมต่อ (ซึ่งเป็นส่วนที่เพิ่มความล่าช้า)
- ง่าย ไม่ซับซ้อน : ไม่ต้องมีสถานะของการเชื่อมต่อที่ผู้ส่งและผู้รับ
- ส่วนหัว (header) มีขนาดเล็ก
- ไม่มีการควบคุมความคับคั่ง: UDP สามารถส่งข้อมูลได้รวดเร็วตามที่ต้องการ

checksum ของ UDP

จุดมุ่งหมาย: ตรวจสอบข้อผิดพลาด เช่น bits ที่ค่าโดนกลับใน segment

ผู้ส่ง:

- จัดการข้อมูลทั้งหมดที่จะไปอยู่ใน segment รวมทั้งข้อมูลส่วนหัวต่าง ๆ ที่เป็นลำดับของจำนวนเต็ม 16 บิตหลายจำนวน
- checksum: ผลรวมของข้อมูลต่าง ๆ ใน segment
- ผู้ส่งใส่ checksum เข้าไปใน field checksum ใน header ของ UDP

ผู้รับ:

- คำนวณหา checksum ของ segment ที่ได้รับมาใหม่
- ตรวจสอบค่า checksum ใน field ของ header กับ ค่าที่เพิ่งได้จากการคำนวณ ว่าเท่ากันหรือไม่:
  - ถ้าไม่ แสดงว่าพบข้อผิดพลาด
  - *ถ้าใช่ แสดงว่า ไม่พบข้อผิดพลาด แต่จริง ๆ แล้ว ก็อาจจะมีข้อผิดพลาดได้อยู่ดี (รายละเอียดจะกล่าวภายหลัง)*

Internet checksum: ตัวอย่าง

ตัวอย่าง : การบวก 16-bit integers สองจำนวน

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	(1)	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Note: ขณะที่บวกตัวเลข, เมื่อ bitที่ 16 (bit ซ้ายสุด) มีการทดเลข ให้นำเลขนั้นไปบวกกับ bit ที่ 1 แล้วจึง flip bit ทุก bit เพื่อให้ได้ checksum

Chapter 3 Outline

3.1 บริการในขั้น Transport

3.2 การรวมและการแยกข้อมูล (multiplexing and demultiplexing)

3.3 การส่งข้อมูลที่ไม่ต้องมีการเชื่อมต่อ (connectionless transport): UDP

3.4 หลักการต่าง ๆ ของการส่งข้อมูลที่ไว้ใจได้ (principles of reliable data transfer)

3.5 การส่งข้อมูลที่ต้องมีการเชื่อมต่อก่อน (connection-oriented transport): TCP

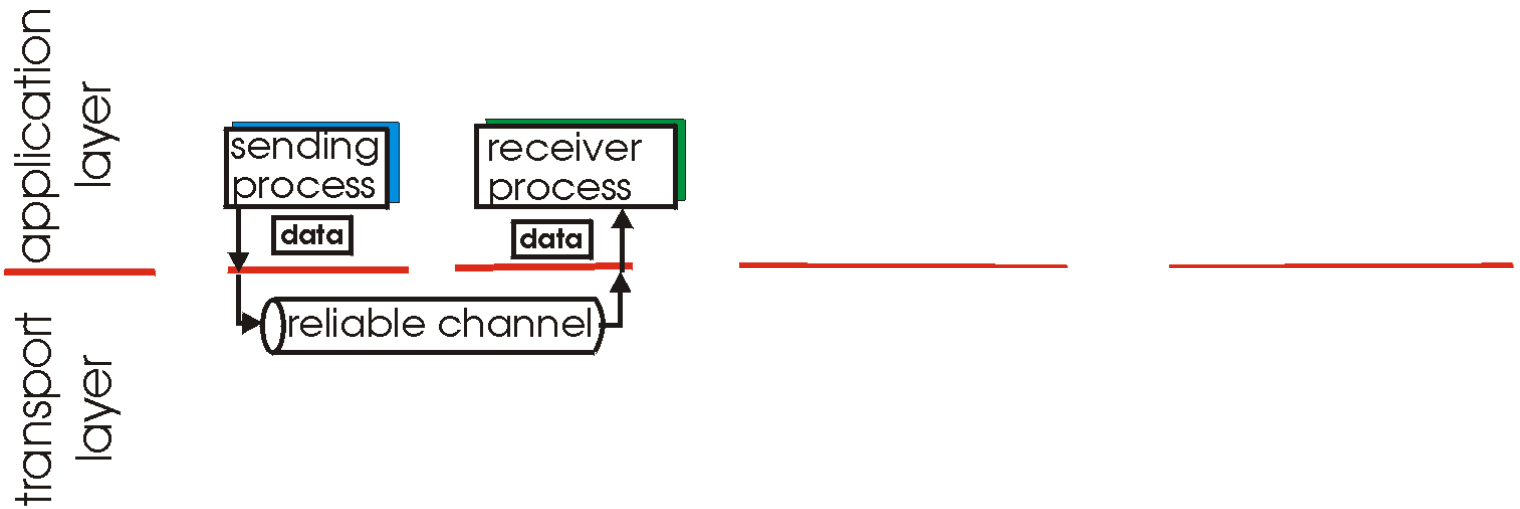
- โครงสร้างส่วนข้อมูล (segment)
- การส่งข้อมูลที่น่าเชื่อถือ/น่าไว้ใจได้ (reliable data transfer)
- การควบคุมการไหล (flow control)
- การจัดการการเชื่อมต่อ (connection management)

3.6 หลักการของการควบคุมความคับคั่ง (congestion control)

3.7 congestion control ของ TCP

Principles of reliable data transfer

- สำคัญในชั้น application, transport, link
  - ติด top-10 ของหัวข้อทางด้านเครือข่าย

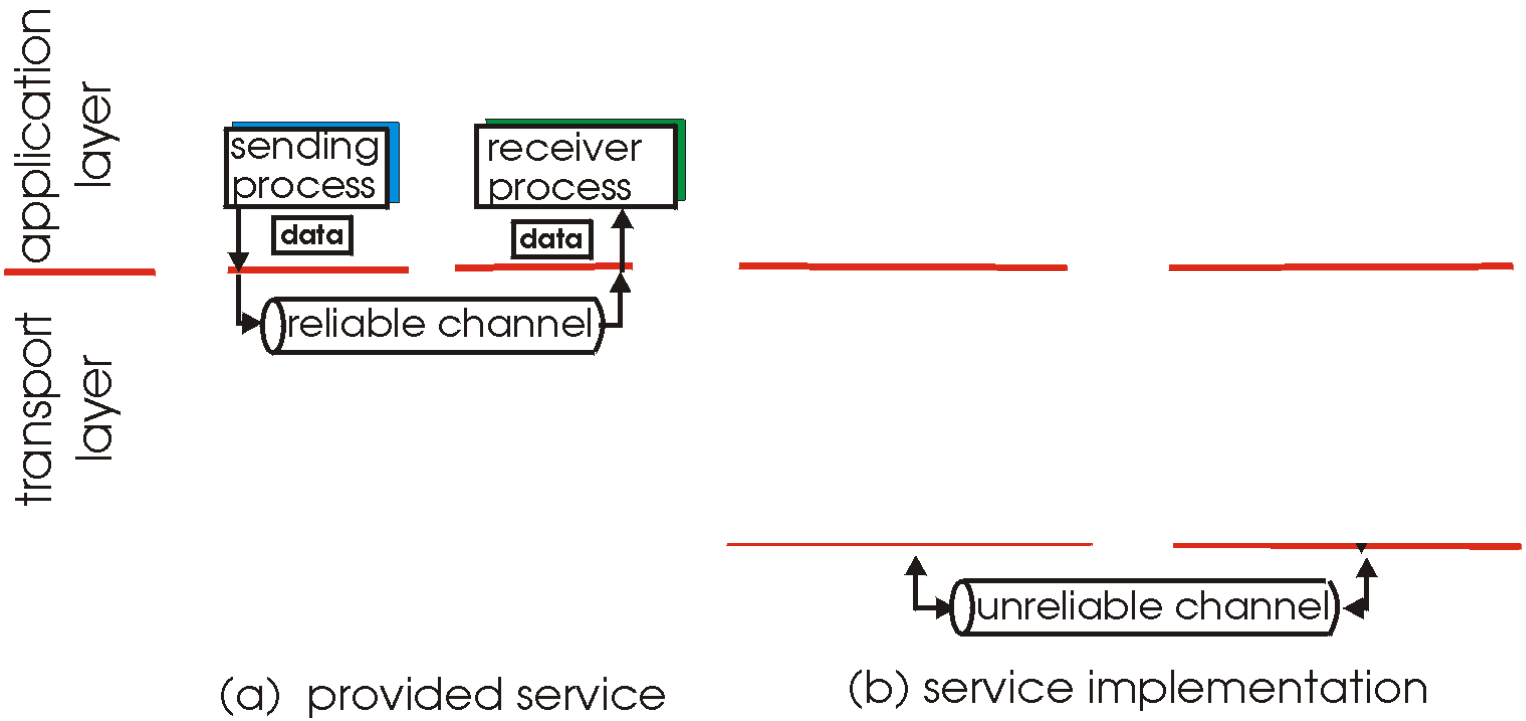


(a) provided service

- ช่องทางที่ไม่น่าเชื่อถือมีลักษณะที่จะเป็นตัวกำหนดความซับซ้อนของ protocol โอนอ่ายข้อมูลที่ reliable (reliable data transfer protocol, rdt)

หลักการของความน่าเชื่อถือ data transfer

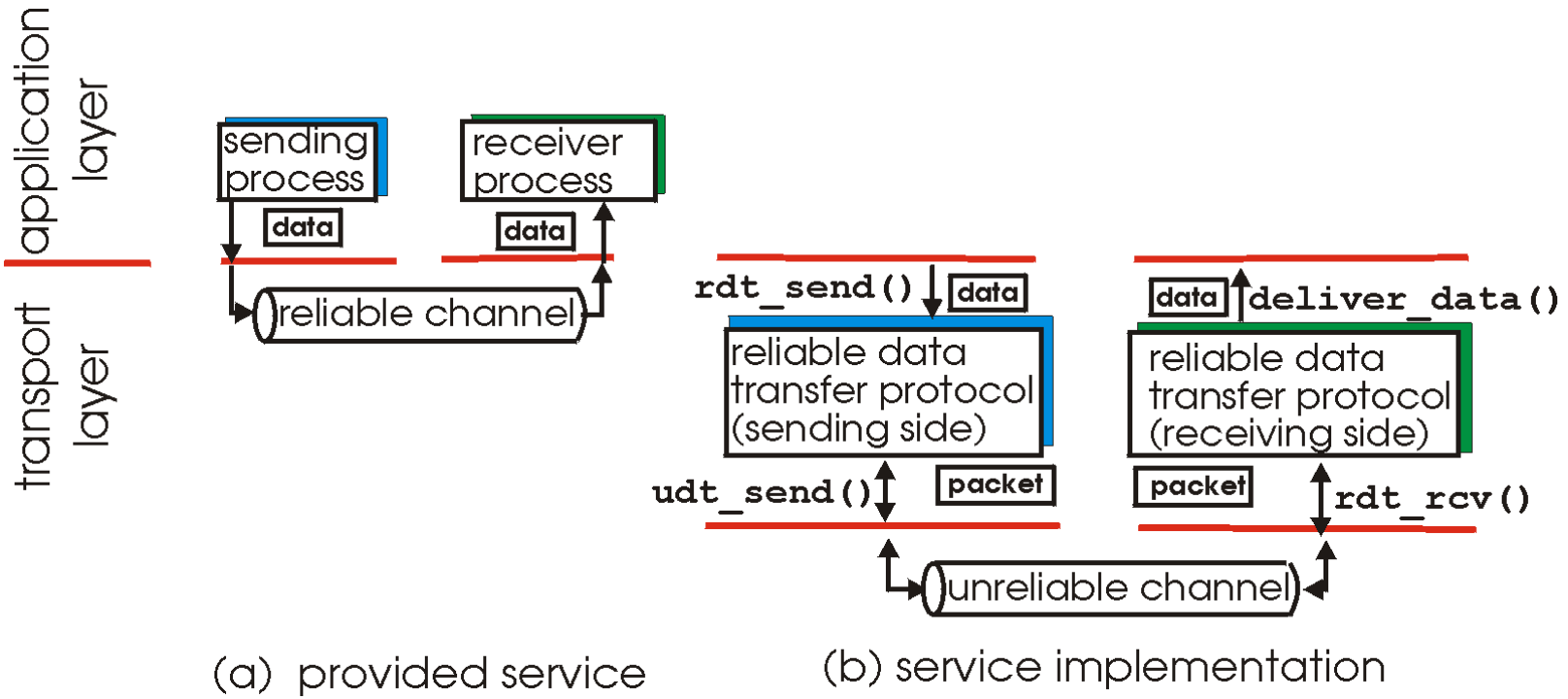
- สำคัญในชั้น application, transport, link
  - ติด top-10 ของหัวข้อทางด้านเครือข่าย



- ช่องทางที่ไม่น่าเชื่อถือมีลักษณะที่จะเป็นตัวกำหนดความซับซ้อนของ protocol โอนอ่ายข้อมูลที่ reliable (reliable data transfer protocol, rdt)

หลักการของความน่าเชื่อถือ data transfer

- สำคัญในชั้น application, transport, link
  - ติด top-10 ของหัวข้อทางด้านเครือข่าย

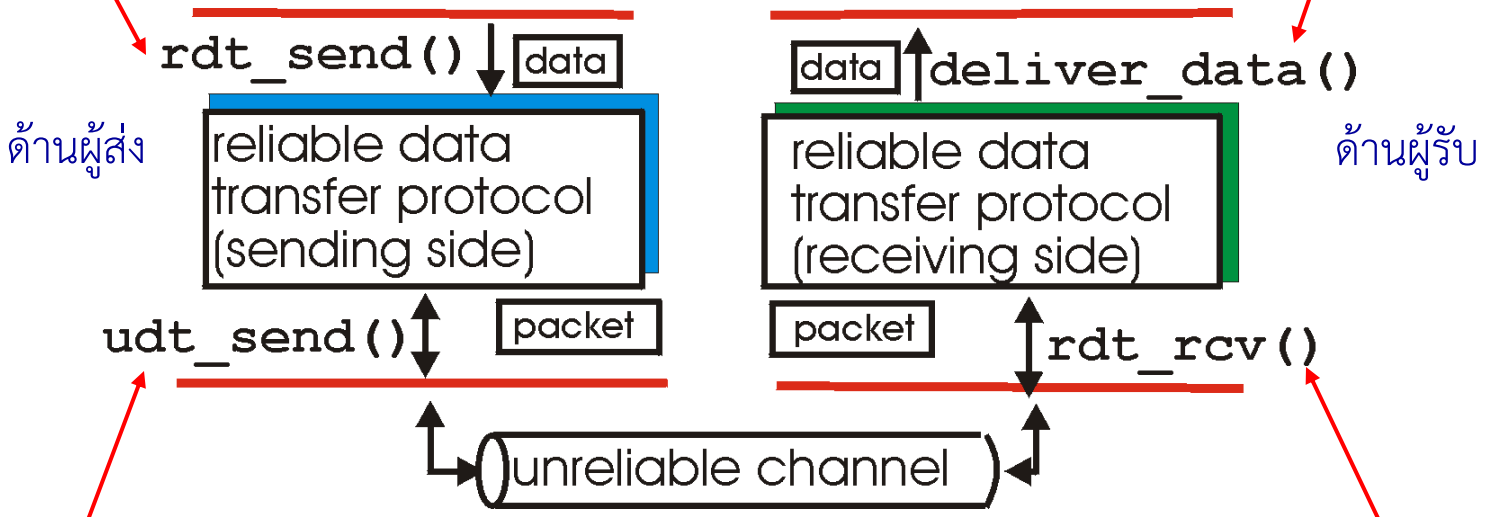


- ช่องทางที่ไม่น่าเชื่อถือมีลักษณะที่จะเป็นตัวกำหนดความซับซ้อนของ protocol โอนอ่ายข้อมูลที่ reliable (reliable data transfer protocol, rdt)

Reliable data transfer: เริ่มต้น

rdt_send(): ถูกเรียกจากชั้นด้านบน (เช่น โดย app), ทำหน้าที่ผ่านข้อมูลเพื่อให้ข้อมูลถูกส่งไปยังชั้นด้านบนของผู้รับ

deliver_data(): ถูกเรียกโดย rdt เพื่อให้ส่งข้อมูลไปยังชั้นด้านบน



udt_send(): ถูกเรียกโดย rdt, เพื่อที่จะโอนถ่าย packet ผ่านช่องการสื่อสารที่ unreliable ไปยังผู้รับ

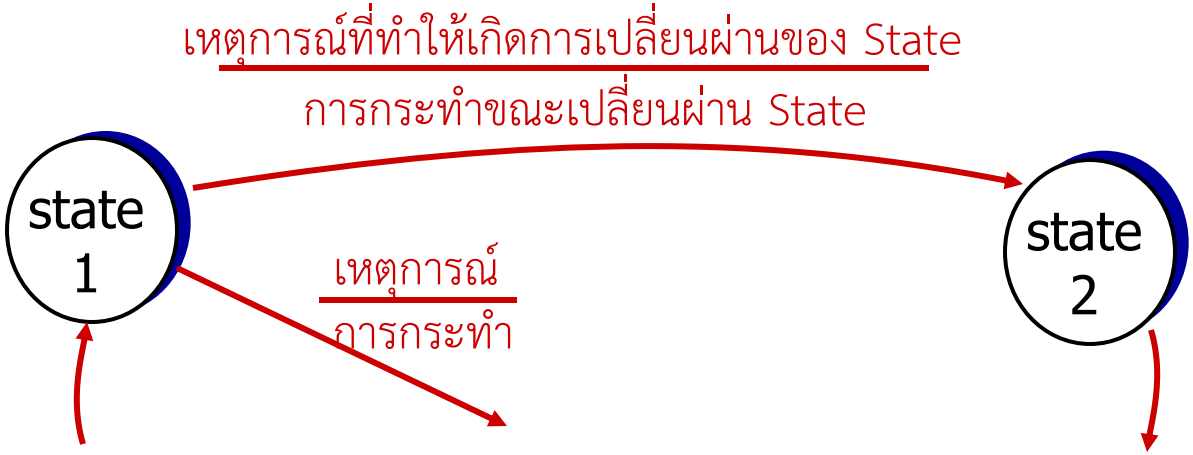
rdt_rcv(): ถูกเรียกโดย OS เมื่อ OS ได้รับ packet ที่มาถึงช่องการสื่อสารด้านผู้รับ

Reliable data transfer: เริ่มต้น

เราจะ:

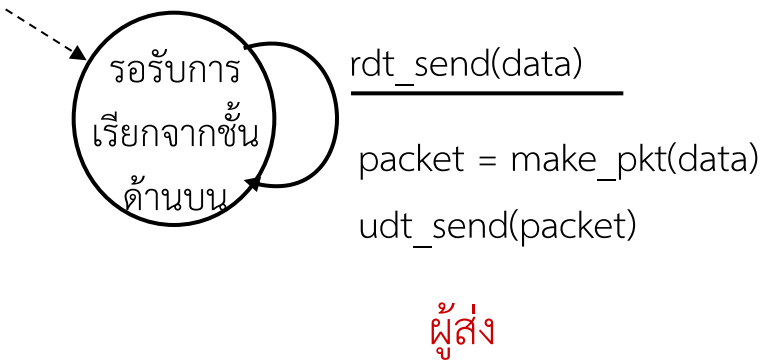
- ค่อย ๆ พัฒนา protocol สำหรับการโอนถ่ายข้อมูลแบบ reliable หรือ reliable data transfer protocol (rdt)
- ตอนนี้ พิจารณาเฉพาะการโอนถ่ายข้อมูลแบบทางเดียวก่อน
  - แต่ข้อมูลสำหรับการควบคุม (control info) จะถูกส่งไปกลับทั้งสองทาง!
- ใช้ finite state machines (FSM) เพื่อกำหนดการกระทำของผู้ส่งและผู้รับ

สถานะ (state): เมื่ออยู่ใน state ปัจจุบัน, state ถัดไปจะขึ้นอยู่กับเหตุการณ์ที่จะเกิดขึ้น ณ state ปัจจุบัน

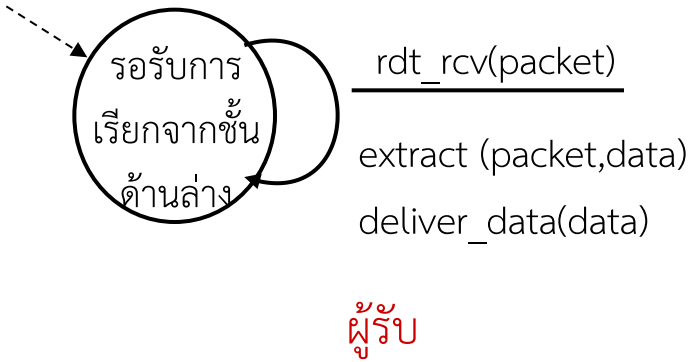


rdt1.0: การโอนถ่ายข้อมูลแบบ reliable ผ่านช่องสัญญาณแบบ reliable

- สมมติให้ ช่องทางที่อยู่ข้างใต้เป็นช่องทางที่เชื่อถือได้
  - ไม่มีการผิดพลาดของบิต
  - ไม่มีข้อมูลสูญหาย
- FSMs สำหรับผู้ส่ง และ สำหรับผู้รับ :
  - ผู้ส่งส่งข้อมูลไปยังชั้นด้านล่าง
  - ผู้รับรับข้อมูลจากชั้นด้านล่าง



ผู้ส่ง



ผู้รับ

rdt2.0: ช่องทางการสื่อสารที่มี bit errors

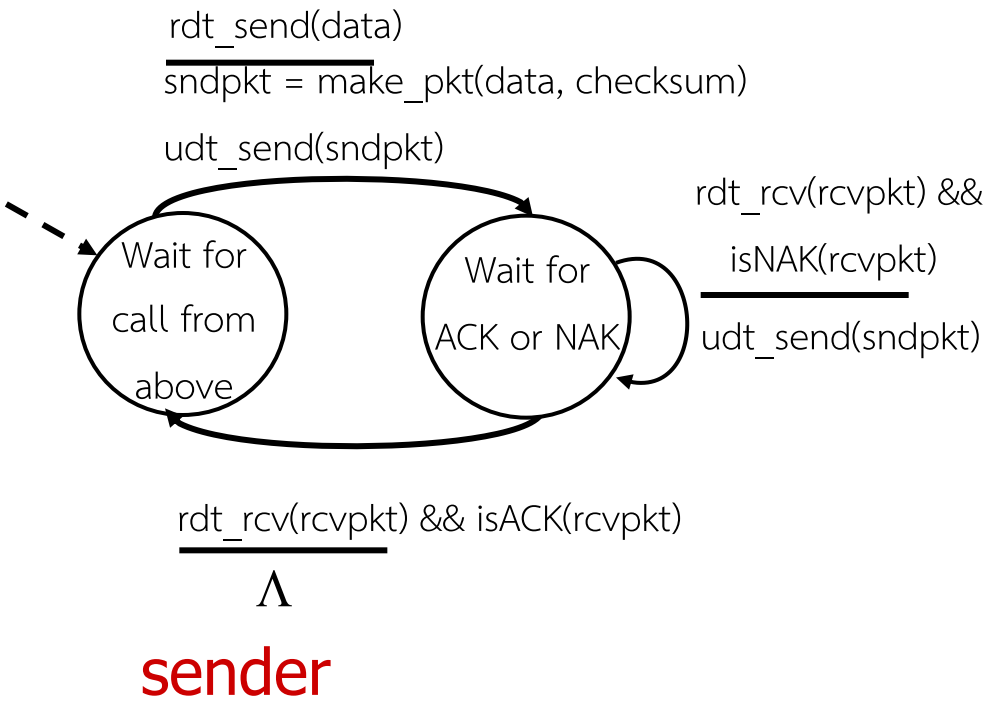
- ช่องสัญญาณอาจกลับค่าบาง bit ใน packet
  - ใช้วิธีการ checksum เพื่อตรวจสอบ bit errors
- *คำถาม*: จะสามารถแก้ไขข้อมูลให้ถูกต้องได้อย่างไร:

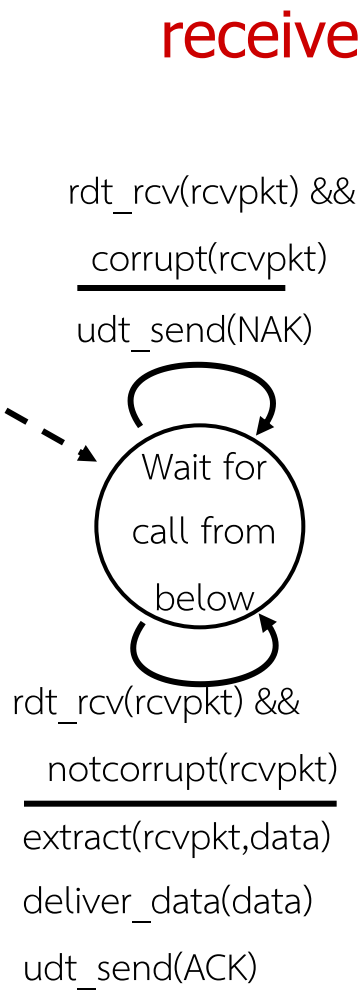
ปกติแล้ว มนุษย์ใช้วิธีการใดในการแก้ไข “errors” ระหว่างการสนทนา?

rdt2.0: ช่องทางการสื่อสารที่มี bit errors

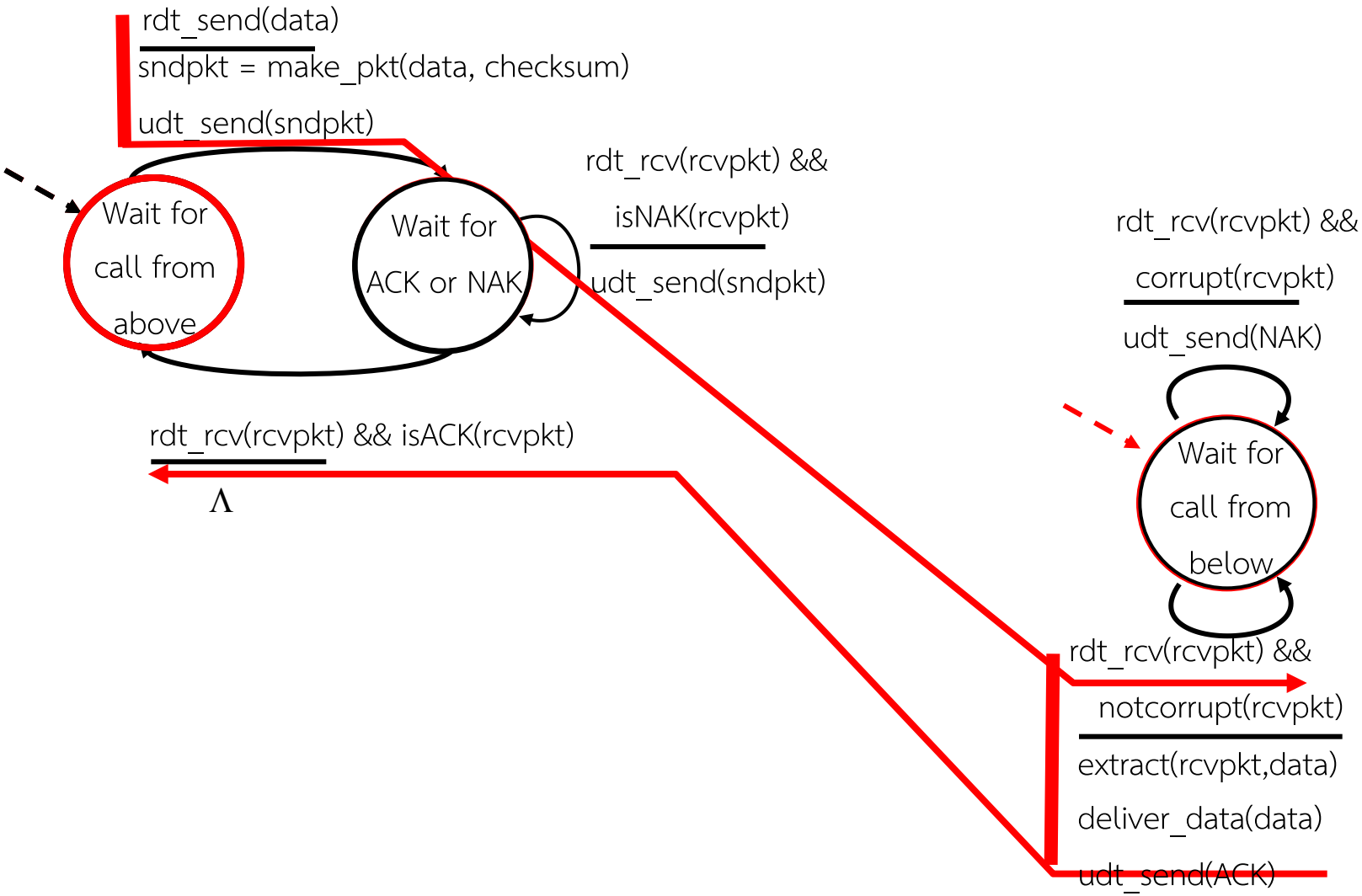
- ช่องสัญญาณอาจกลับค่าบาง bit ใน packet
  - ใช้วิธีการ checksum เพื่อตรวจสอบ bit errors
- *คำถาม*: จะสามารถแก้ไขข้อมูลให้ถูกต้องได้อย่างไร:
  - *การตอบรับ (acknowledgements (ACKs))*: ผู้รับแจ้งผู้ส่งว่าได้รับ packet แล้ว
  - *การตอบปฏิเสธ (negative acknowledgements (NAKs))*: ผู้รับแจ้งผู้ส่งว่า packet มี errors
  - sender ส่งข้อมูลซ้ำอีกครั้ง เมื่อได้รับ NAK
- กลไกที่เพิ่มขึ้น in **rdt2.0** (เพิ่มจาก **rdt1.0**):
  - การตรวจจับความผิดพลาด
  - การแจ้งกลับ (feedback): ข้อมูลสำหรับการควบคุม (ACK,NAK) จากผู้รับไปยังผู้ส่ง

rdt2.0: ข้อกำหนดของ FSM

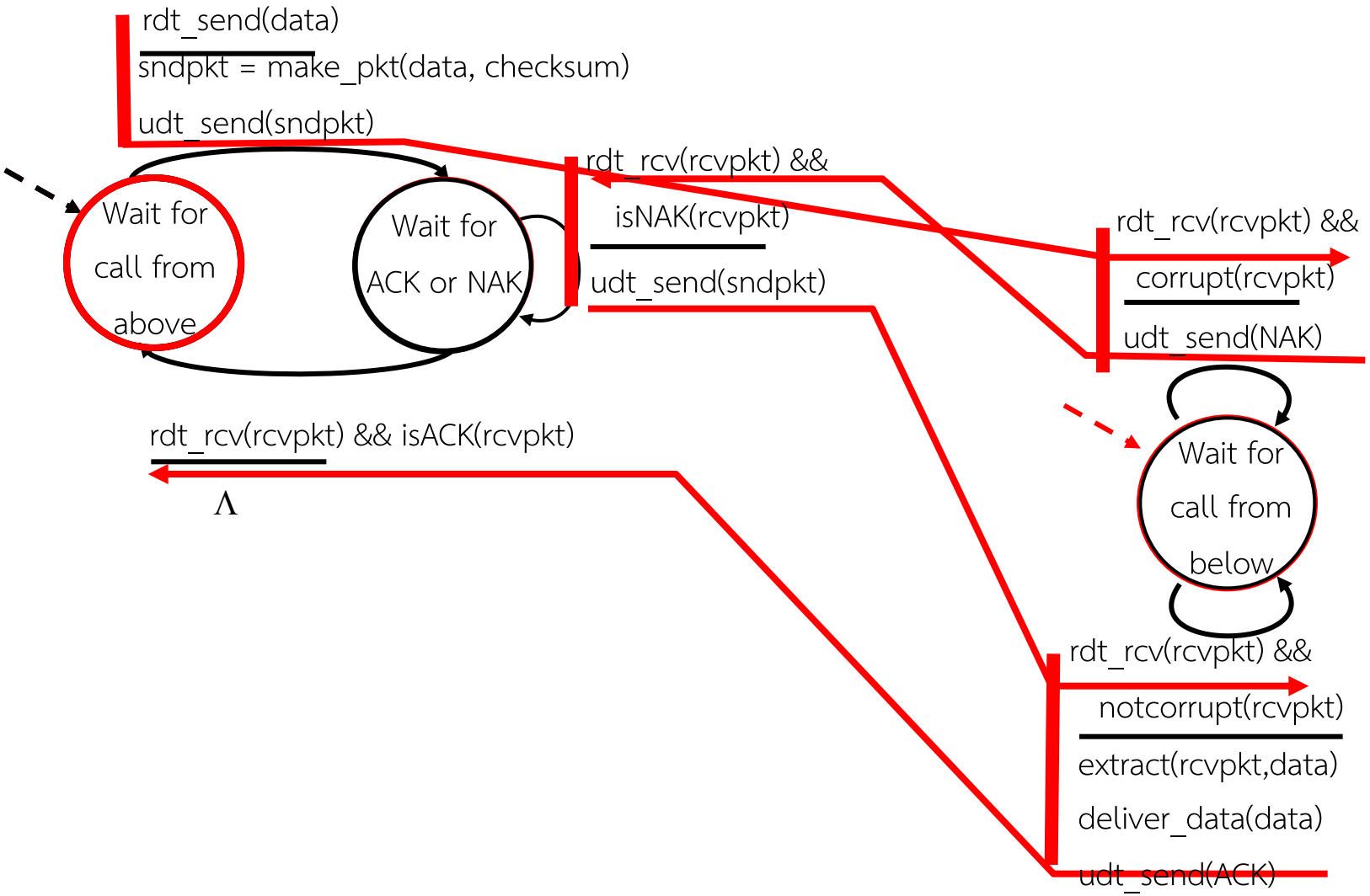




rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

จะเกิดอะไรขึ้นถ้า ACK/NAK เกิดเสียหาย?

- ผู้ส่งไม่รู้ว่าผู้รับรับ pkt ครบถ้วนหรือไม่
- ไม่ควรส่ง pkt ซ้ำอีกครั้ง: เพราะ pkt ซ้ำกัน

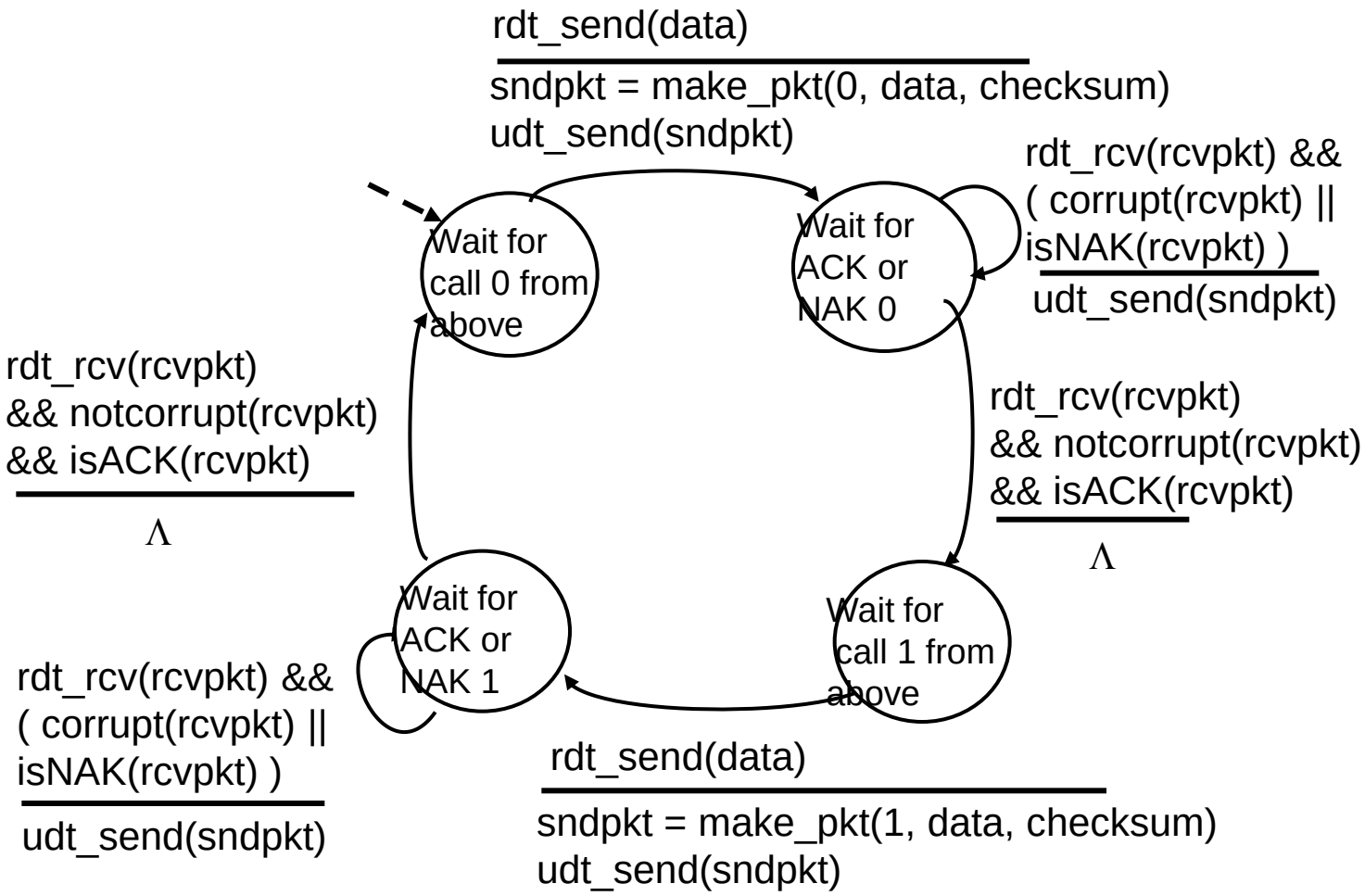
การจัดการกับข้อมูลที่ซ้ำซ้อน:

- ผู้ส่งจะส่ง packet อีกครั้งถ้า ACK/NAK เกิดเสียหาย
- เติมหมายเลข *sequence* ให้แต่ละ packet
- ผู้รับต้องทิ้ง packet ที่ซ้ำกัน

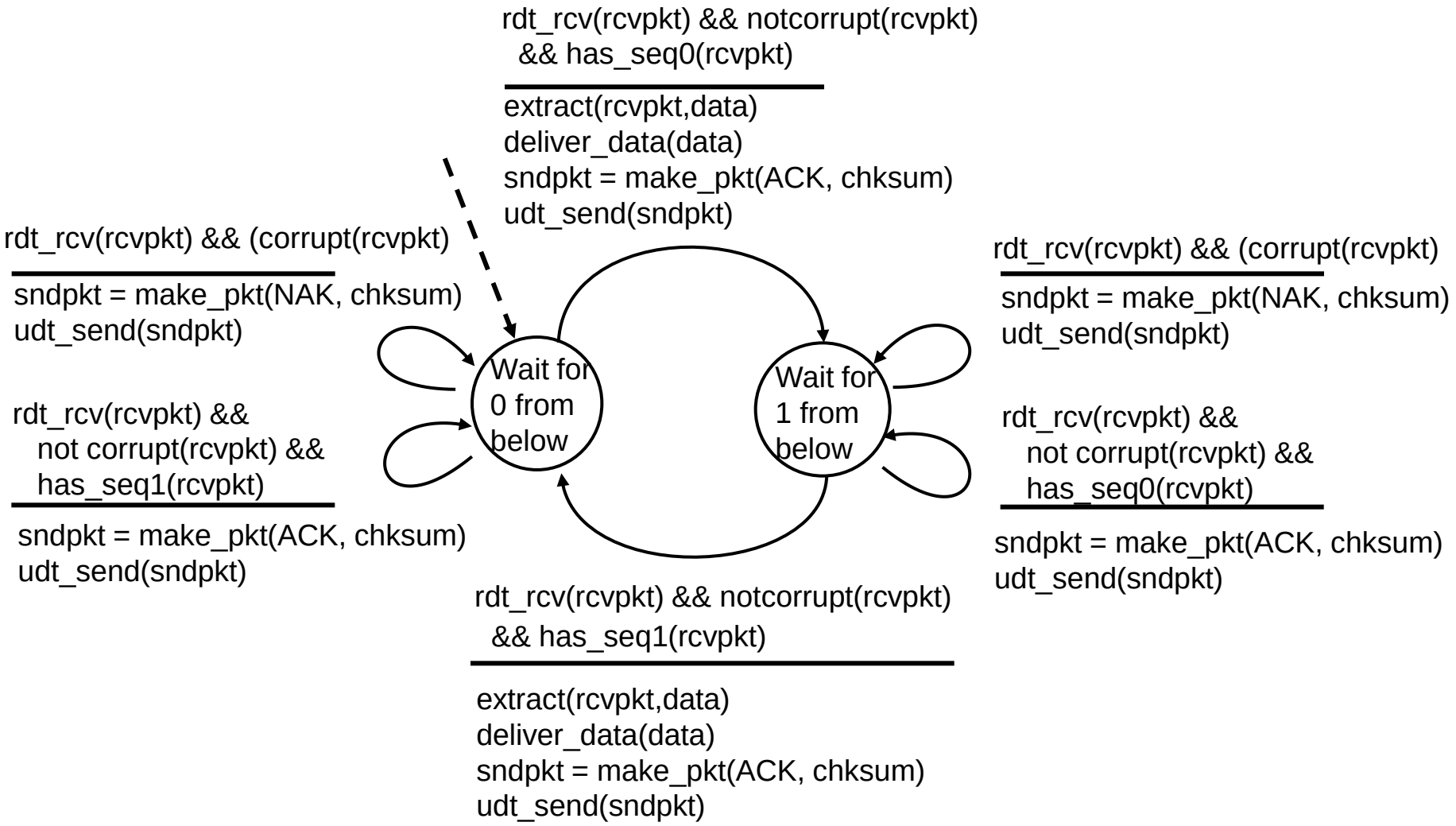
stop and wait

ผู้ส่งทำการส่งpacketและรอให้ผู้รับตอบกลับด้วย ACK/NAK ก่อนที่จะส่ง packet ต่อไป

rdt2.1: ผู้ส่งจัดการกับ ACK/NAKs ที่ไม่ถูกต้อง



rdt2.1: ผู้ส่งจัดการกับ ACK/NAKs ที่ไม่ถูกต้อง



rdt2.1: อภิปราย

ผู้ส่ง:

- ใส่หมายเลข seq ในpacket
- หมายเลข seq (0,1) ก็พอ, นึกสิทำไม?
- ต้องตรวจสอบว่า ACK/NAK ที่ได้รับเสียไปหรือไม่?
- จำนวนหมายเลข seq ต้องเป็นสองเท่าของจำนวนสถานะ
  - เพราะผู้ส่งต้องเก็บสถานะเพื่อ“จำ” ว่า pkt ถัดไปจะต้องมีเลข seq 0 หรือ 1

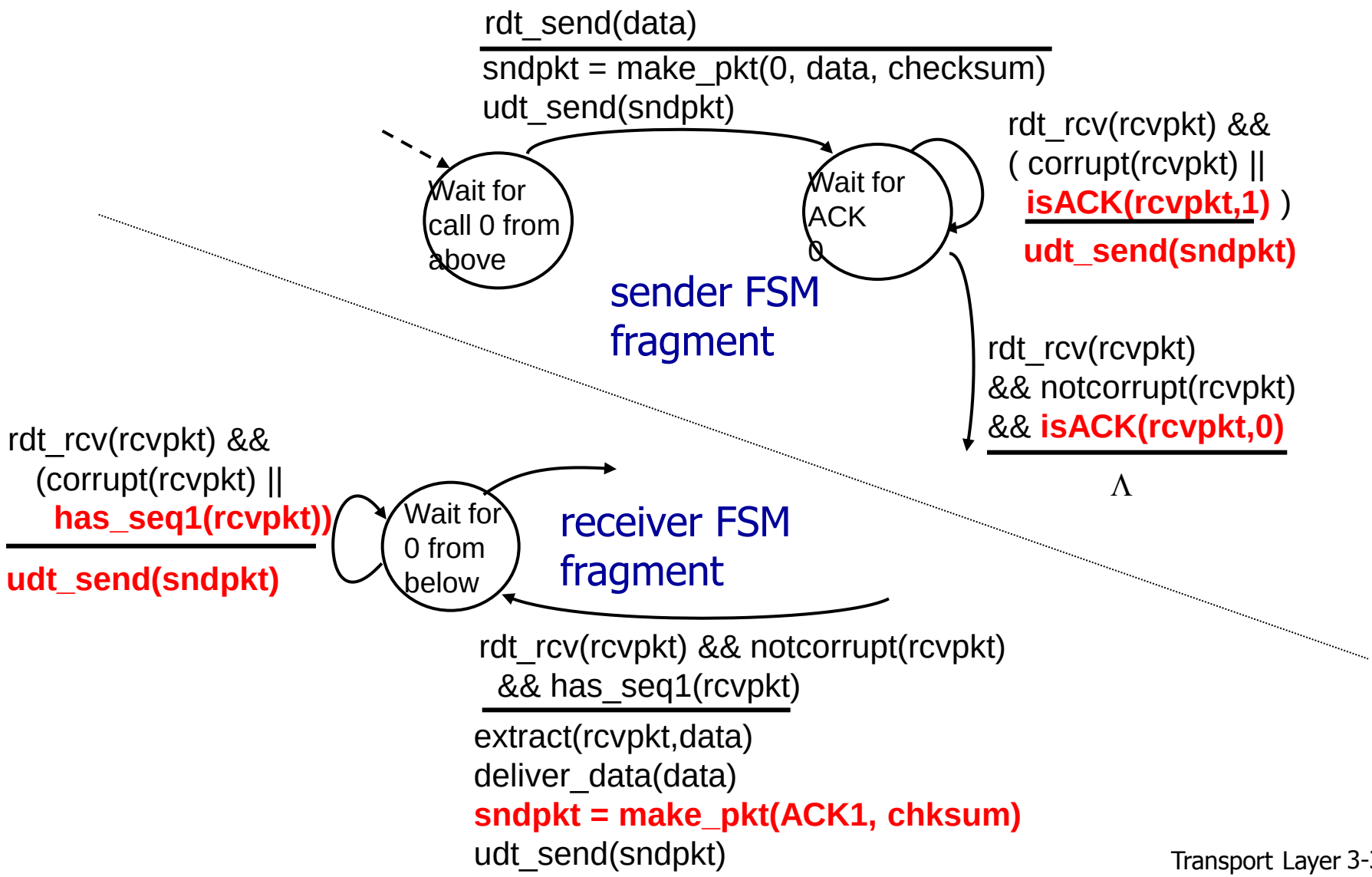
ผู้รับ:

- ต้องตรวจสอบว่า pkt ที่ได้รับซ้ำกับ pkt ที่ได้รับก่อนหน้าหรือไม่
  - ผู้รับจะเก็บสถานะที่ระบุว่า 0 หรือ 1 คือหมายเลข seq ที่คาดว่าจะได้
- ข้อสังเกต: ผู้รับไม่สามารถรู้ว่า ACK/NAK สุดท้ายที่ผู้ส่งได้รับถูกต้องหรือไม่

rdt2.2: protocol ที่ไม่ต้องใช้ NAK

- ทำงานเหมือนกับ rdt2.1, แต่ใช้แค่ ACKs
- แทนที่จะส่ง NAK, ผู้รับส่ง ACK สำหรับ pkt สุดท้ายที่ได้รับถูกต้อง
  - ฝั่งรับจะต้องรวมหมายเลข seq ของ pkt ที่กำลังถูก ACK ไปด้วย
- จะส่ง ACK ซ้ำไปที่ผู้ส่งถ้าไม่ได้รับ pkt ที่มีหมายเลข seq ที่คาดหวัง, ผลลัพธ์จะออกมาเหมือนกรณีที่ส่ง NAK: *ผู้ส่งต้องส่ง pkt ตัวล่าสุดไปใหม่อีกครั้ง*

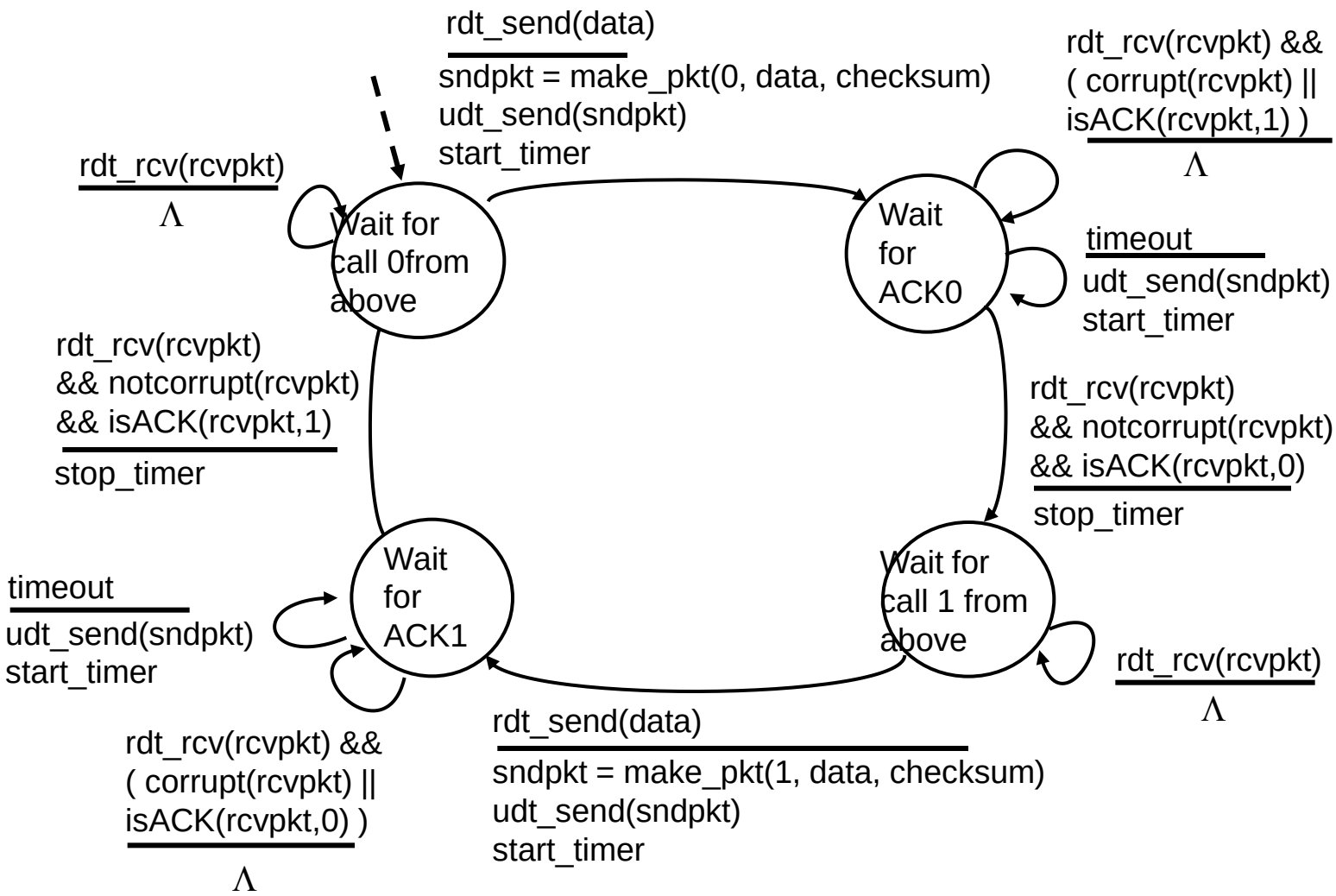
rdt2.2: sender, receiver fragments



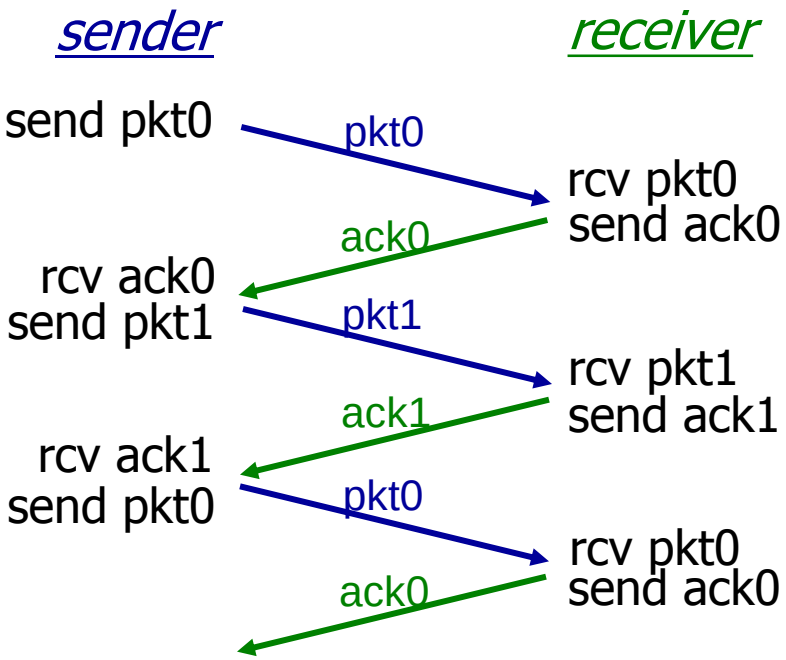
rdt3.0: ช่องสัญญาณที่มีความผิดพลาดและ pkt สูญหาย

- เพิ่มข้อสันนิษฐาน: ช่องสัญญาณสามารถสูญเสียแพ็คเกจ (ไม่ว่าจะเป็นข้อมูลหรือ ACKs) ได้
  - กลไก checksum, หมายเลข seq., ACKs, และ การส่งซ้ำ สามารถช่วย แต่มันก็ยังไม่ใช่ทางออก
- วิธี: ผู้ส่งรอ ACK กลับมาในช่วงเวลาที่ “สมเหตุสมผล”
- ส่ง pkt เดิมไปใหม่อีกครั้งถ้าไม่ได้รับ ACK ภายในช่วงเวลานี้
- ถ้า pkt (หรือ ACK) มาถึงล่าช้า (ไม่ใช่หายไป):
  - การส่งใหม่ก็จะเป็นการส่งซ้ำ ซึ่ง seq. # ได้จัดการเรื่องเกี่ยวกับข้อมูลซ้ำได้
  - ผู้รับต้องระบุเลข seq ของ pkt ที่กำลังถูก ACKed
- ต้องใช้การนับเวลาถอยหลัง (timer)

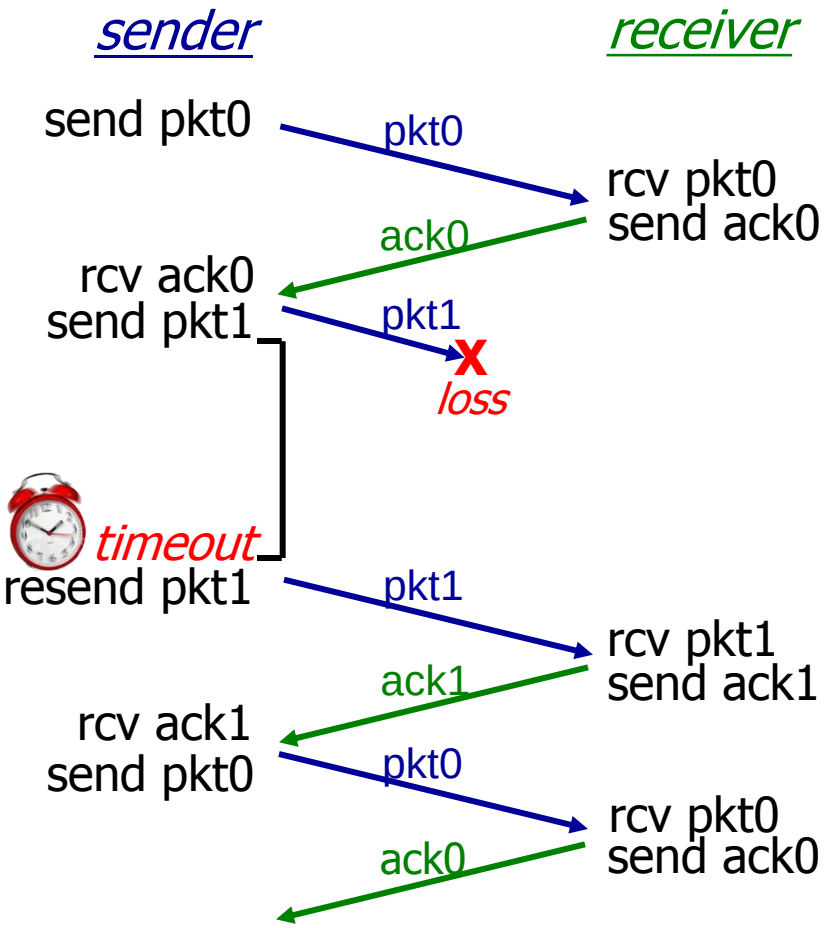
rdt3.0 sender



ขณะ rdt3.0 ดำเนินการ

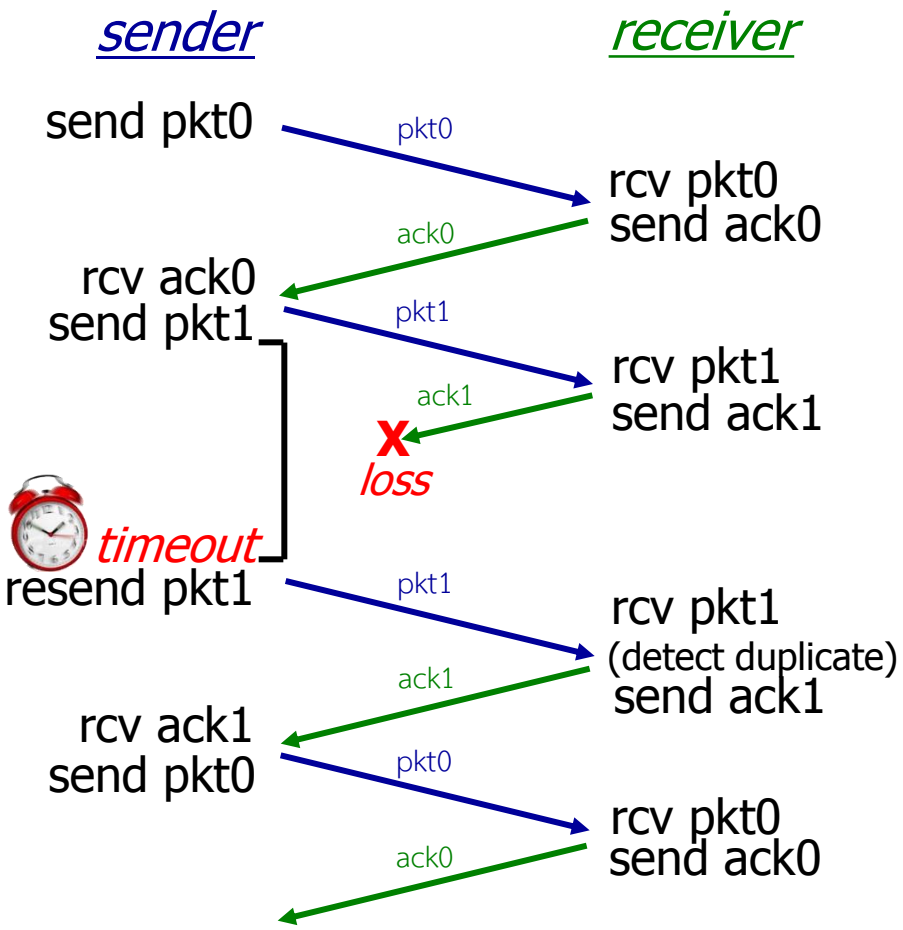


(a) no loss

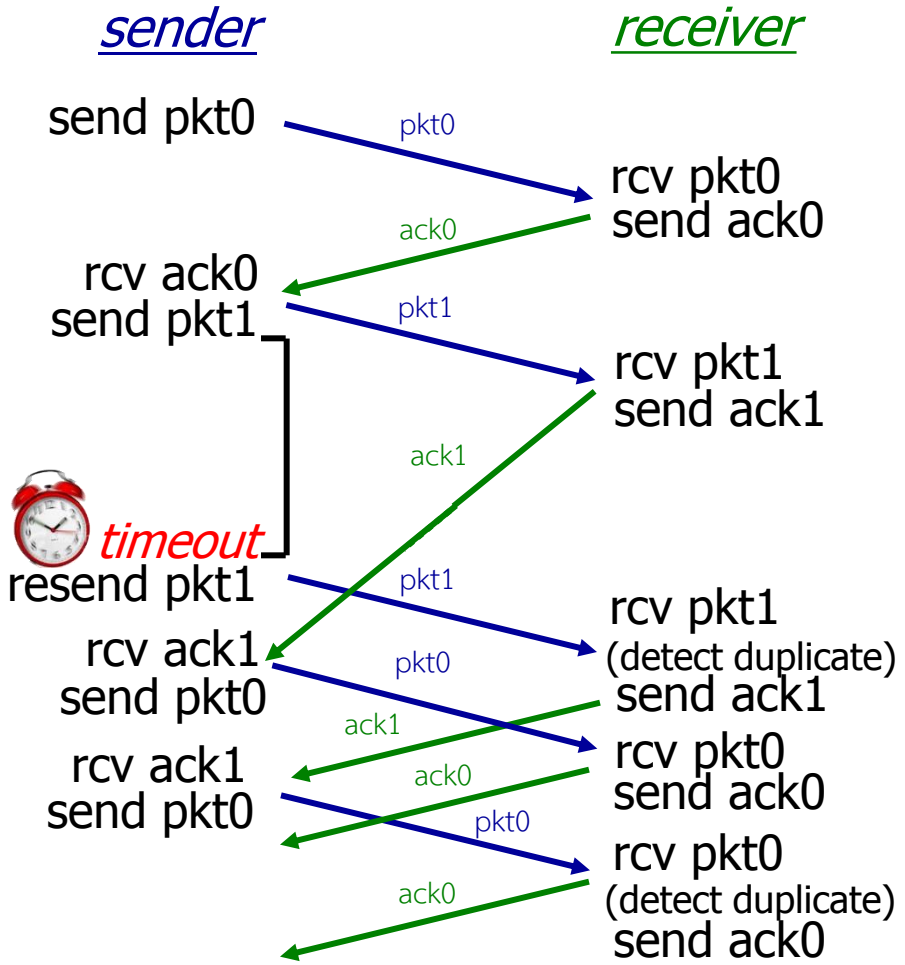


(b) packet loss

ขณะ rdt3.0 ดำเนินการ



(c) ACK loss



(d) premature timeout/ delayed ACK

ประสิทธิภาพของ rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

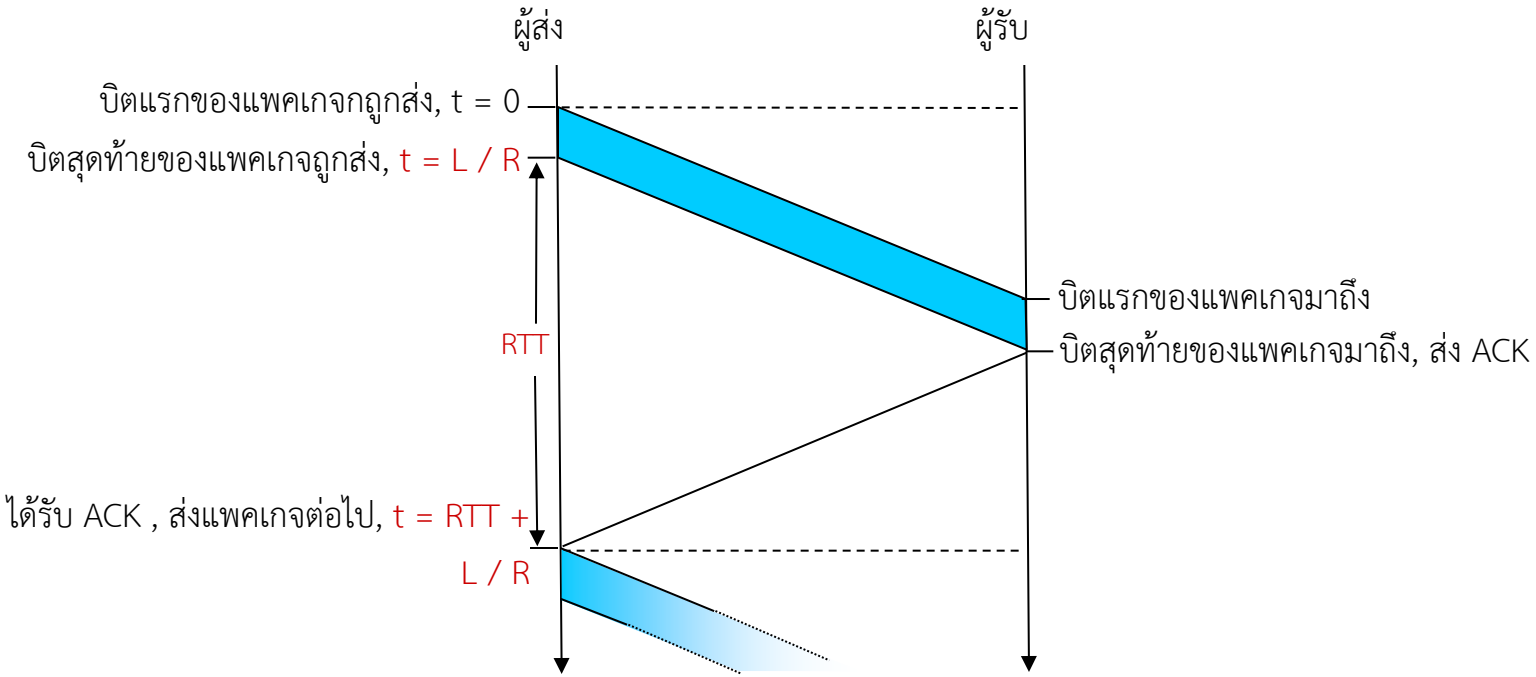
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

  - U_{sender}: *utilization* – fraction of time sender busy sending

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

  - if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation

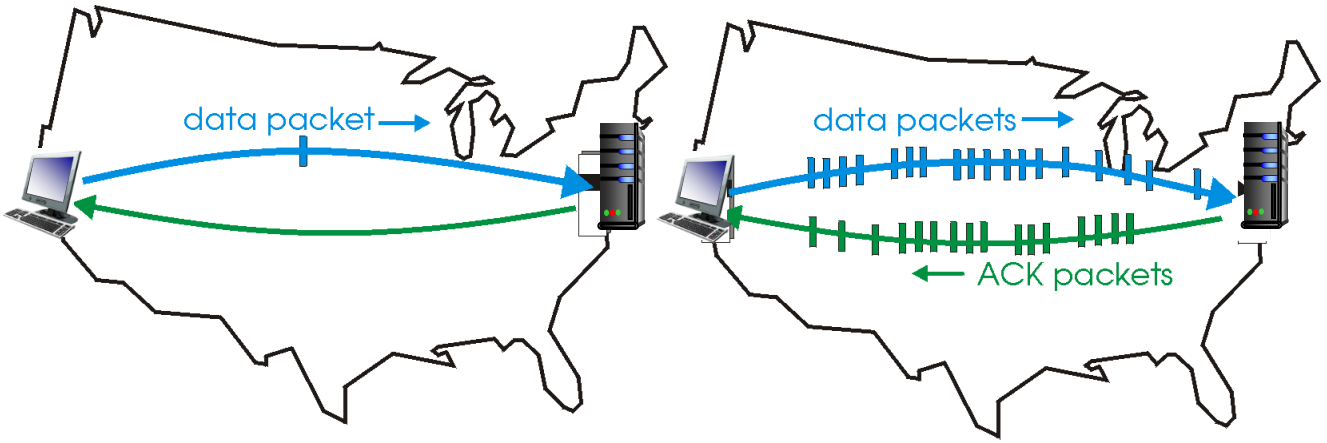


$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

protocols แบบ pipelined

pipelining: ผู้ส่งส่งหลาย packet ไปยังผู้รับทั้ง ๆ ที่ pkt เหล่านั้นยังไม่ได้รับการยืนยันตอบรับ (not yet acked)

- ช่วงของหมายเลข sequence จะต้องเพิ่มขึ้น
- ต้องพักข้อมูล (buffer) ที่ผู้ส่งและผู้รับ

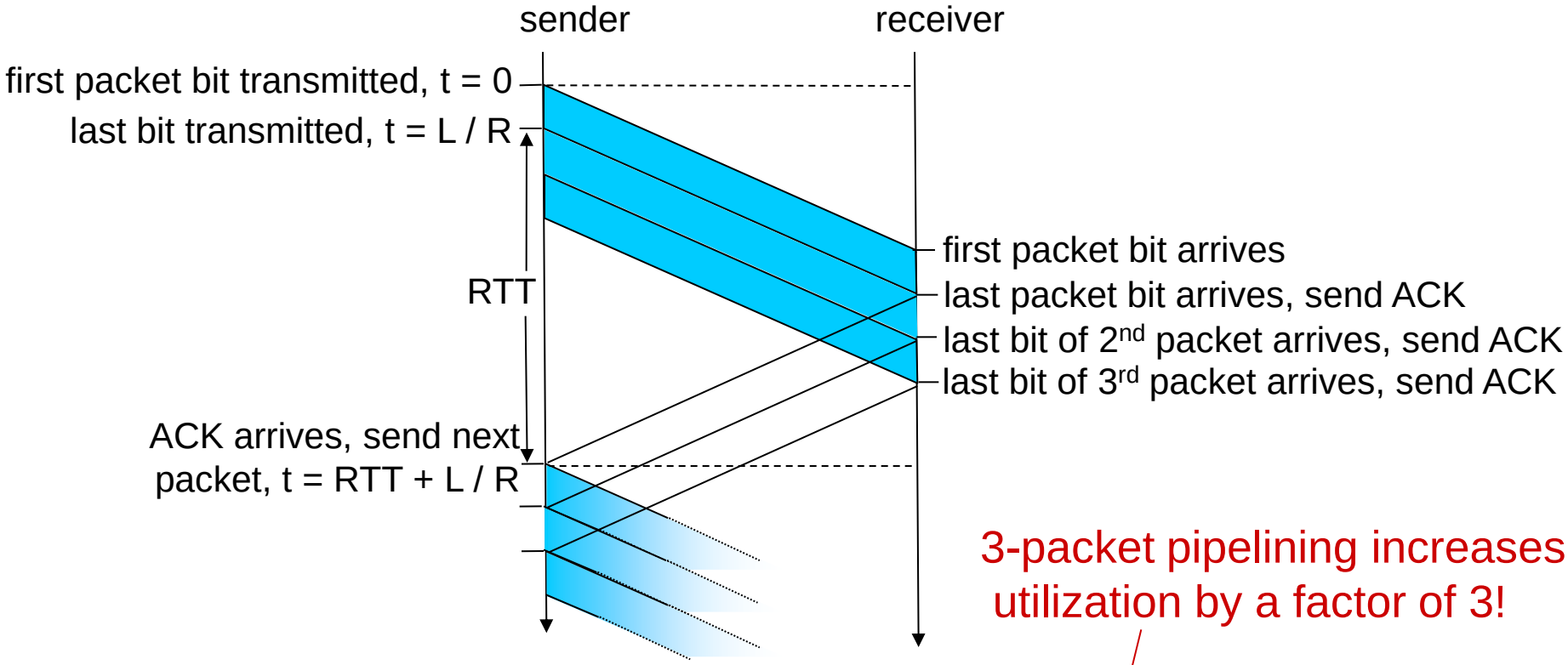


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

❖ protocols แบบ pipelined แบบทั่วไปมีสองแบบ : *go-Back-N, selective repeat*

Pipelining: ช่วยเพิ่ม utilization



3-packet pipelining increases utilization by a factor of 3!

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

protocols แบบ pipeline: ภาพรวม

Go-back-N:

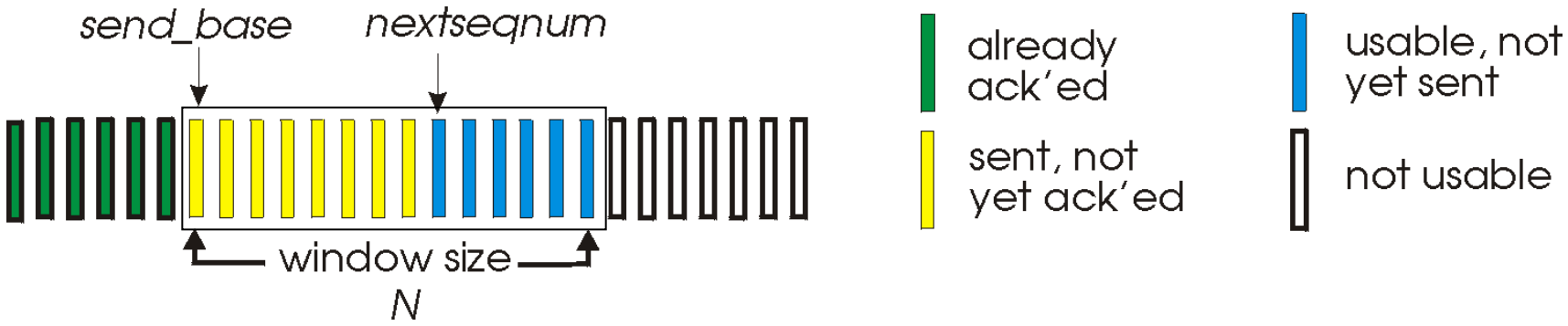
- ผู้ส่งสามารถส่งแพคเกจที่ไม่ยังถูก ack ได้ไม่เกิน N แพคเกจใน pipeline
- ผู้รับแค่ส่ง *cumulative ack*
  - ไม่ ack packet ถ้ามันก่อให้เกิด gap
- ผู้ส่งจับเวลาสำหรับแพคเกจที่ส่งไปนานที่สุดที่ยังไม่ได้ ack
  - เมื่อเวลาหมด ผู้ส่งจะส่งแพคเกจทั้งหมดที่ยังไม่ได้รับ ack อีกครั้ง

Selective Repeat:

- ผู้ส่งสามารถส่งแพคเกจที่ไม่ยังถูก ack ได้ไม่เกิน N แพคเกจใน pipeline
- ผู้รับจะต้องส่ง ack กลับให้แต่ละแพคเกจ
- ผู้ส่งจะจับเวลาสำหรับแต่ละแพคเกจที่ไม่ได้ ack
  - เมื่อเวลาสำหรับแพคเกจใดหมด ผู้ส่งจะส่งเฉพาะแพคเกจนั้นเท่านั้น

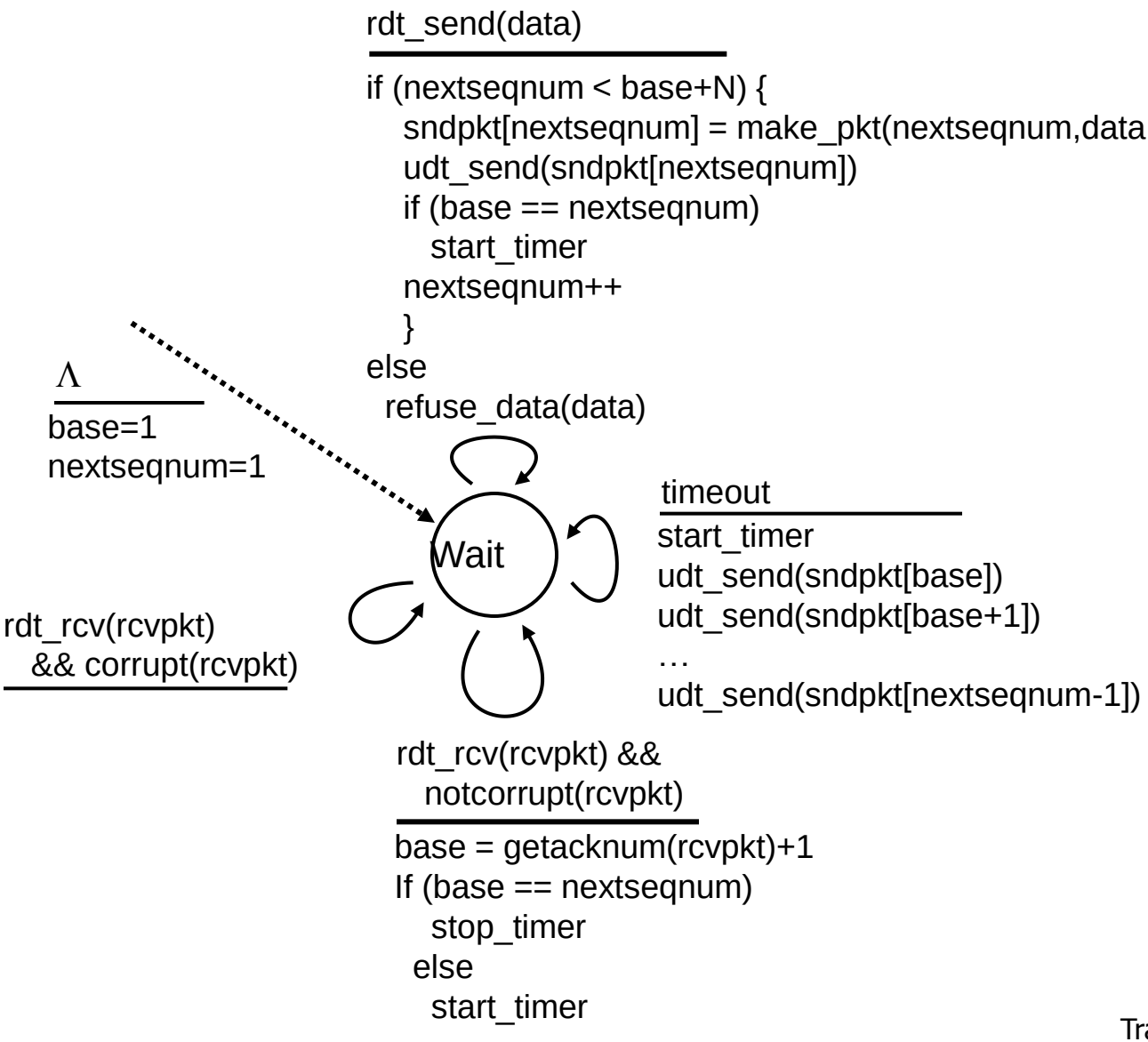
Go-Back-N: ผู้ส่ง

- หมายเลข seq (k bit) จะอยู่ใน header ของ packet
- อนุญาตให้ขนาด "window" (ของ pkt ที่ติดกันและยังไม่ถูก ack) ไม่เกิน N

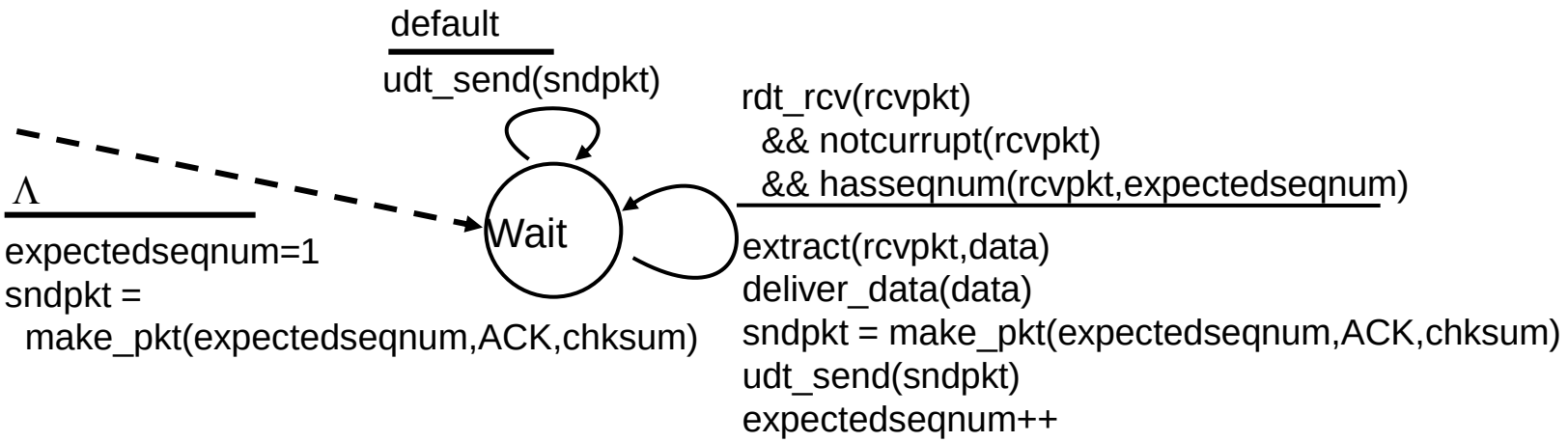


- ACK(n): จะ ack แพคเกจทั้งหมดไปจนถึงที่มีหมายเลข seq เท่ากับ n - *"การ ACK แบบรวม (cumulative ACK)"*
  - อาจมีการได้รับ ACKs หมายเลข seq เดิมซ้ำ (ดูรายละเอียดด้าน receiver)
- มีการจับเวลาสำหรับแพคเกจที่เดินทางอยู่ ซึ่งเดินทางนานที่สุดและยังไม่ได้ ACK (pkt ที่มีหมายเลข seq น้อยที่สุดใน window)
- *timeout(n):* เวลาหมดที่ pkt n, ผู้ส่งจะส่ง pkt n และ pkt ทั้งหมดใน window (ซึ่งมีหมายเลข seq สูงกว่า n) อีกครั้ง

GBN: FSM ด้านผู้ส่งที่เพิ่มจากของเดิม



GBN: FSM ด้านผู้รับที่เพิ่มจากของเดิม



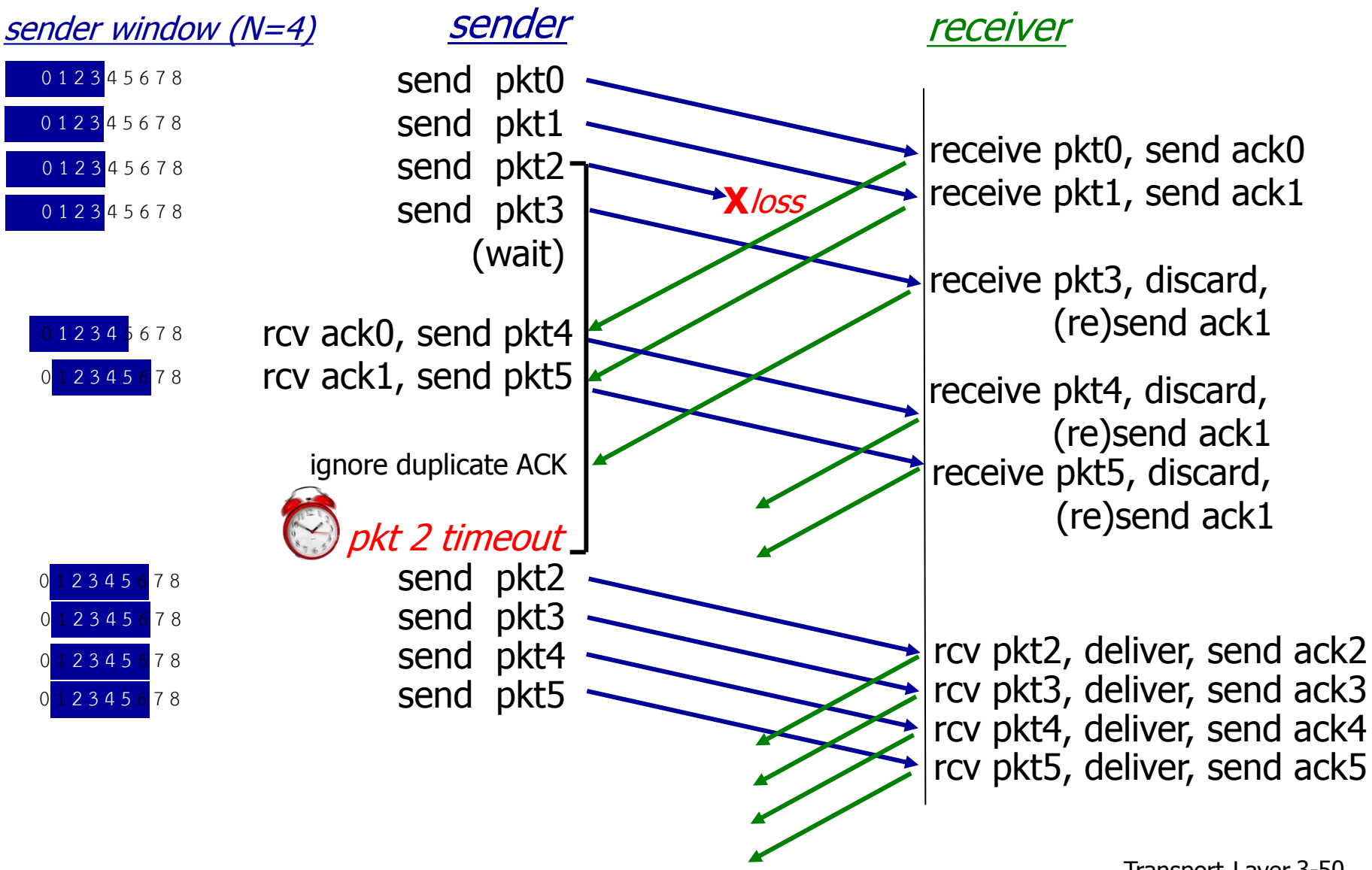
ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember **`expectedseqnum`**

❖ out-of-order pkt:

- discard (don't buffer): *no receiver buffering!*
- re-ACK pkt with highest in-order seq #

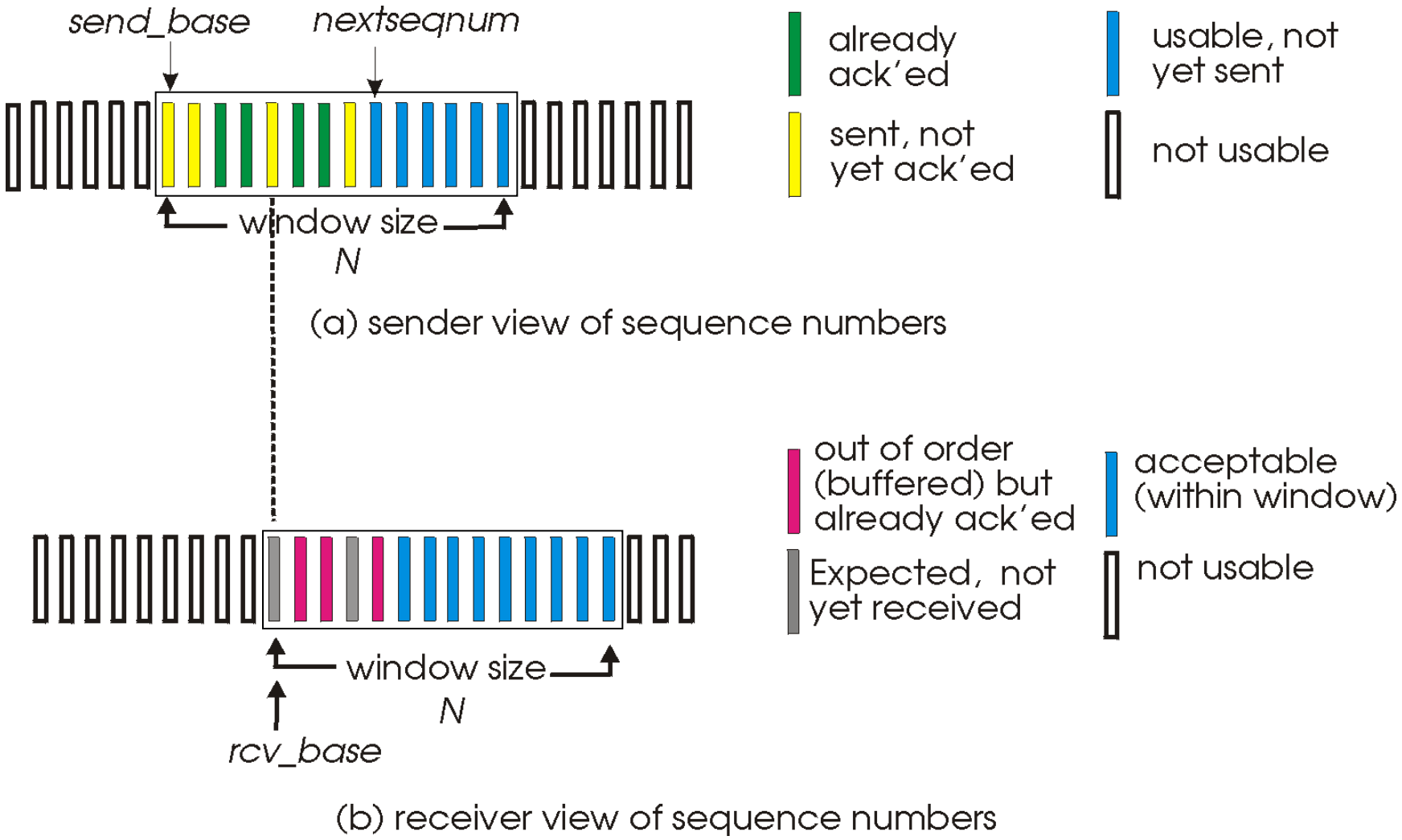
GBN in action



Selective repeat

- ผู้รับ Ack แต่ละแพคเก็จที่ได้รับมาถูกต้อง
  - เก็บ (buffer) แพคเก็จ, เท่าที่จำเป็น, เพื่อที่จะส่งข้อมูลที่เรียงลำดับแล้ว ไปยัง layer บนได้ในท้ายที่สุด
- ผู้ส่งแค่ส่งแพคเก็จซ้ำเฉพาะแพคเก็จที่ไม่ได้รับ Ack เท่านั้น
  - ตัวจับเวลา (ทางฝั่งผู้ส่ง) จะจับเวลาของแต่ละแพคเก็จที่ไม่ได้รับ Ack
- Window ของผู้ส่ง
  - มีหมายเลข seq อยู่ *N* หมายเลขติดกัน
  - จำกัดหมายเลข seq ของแพตเก็จที่ส่งไปแล้ว และไม่ได้รับ Ack

Selective repeat: sender, receiver windows



Selective repeat

ผู้ส่ง

ข้อมูลมาจาก layer ด้านบน:

- ถ้ามีหมายเลข seq ว่างใน window, ให้ส่ง pkt ได้

timeout(n):

- ส่ง pkt n ซ้ำ, จับเวลาใหม่

ACK(n) in [sendbase, sendbase+N]:

- รู้ว่า ผู้รับได้รับ pkt n แล้ว
- ถ้า n เกิดเป็นหมายเลขแรกสุดในหน้าต่าง (n เท่ากับ sendbase) ที่ไม่ได้ ถูก ack, เลื่อน window ไปทางด้านขวา โดยเลื่อนไปยังหมายเลขที่ไม่ได้ถูก ack

ผู้รับ

pkt n in [rcvbase, rcvbase+N-1]

- ส่ง ACK(n)
- ถ้าไม่เรียงตามลำดับ : buffer
- ถ้าตามลำดับ : ส่ง (รวมถึงให้ส่ง pkt ที่เรียงตามลำดับและอยู่ในบัฟเฟอร์), เลื่อน window ไปที่หมายเลข seq ถัดไปของ pkt ที่ยังไม่ได้รับจากผู้ส่ง

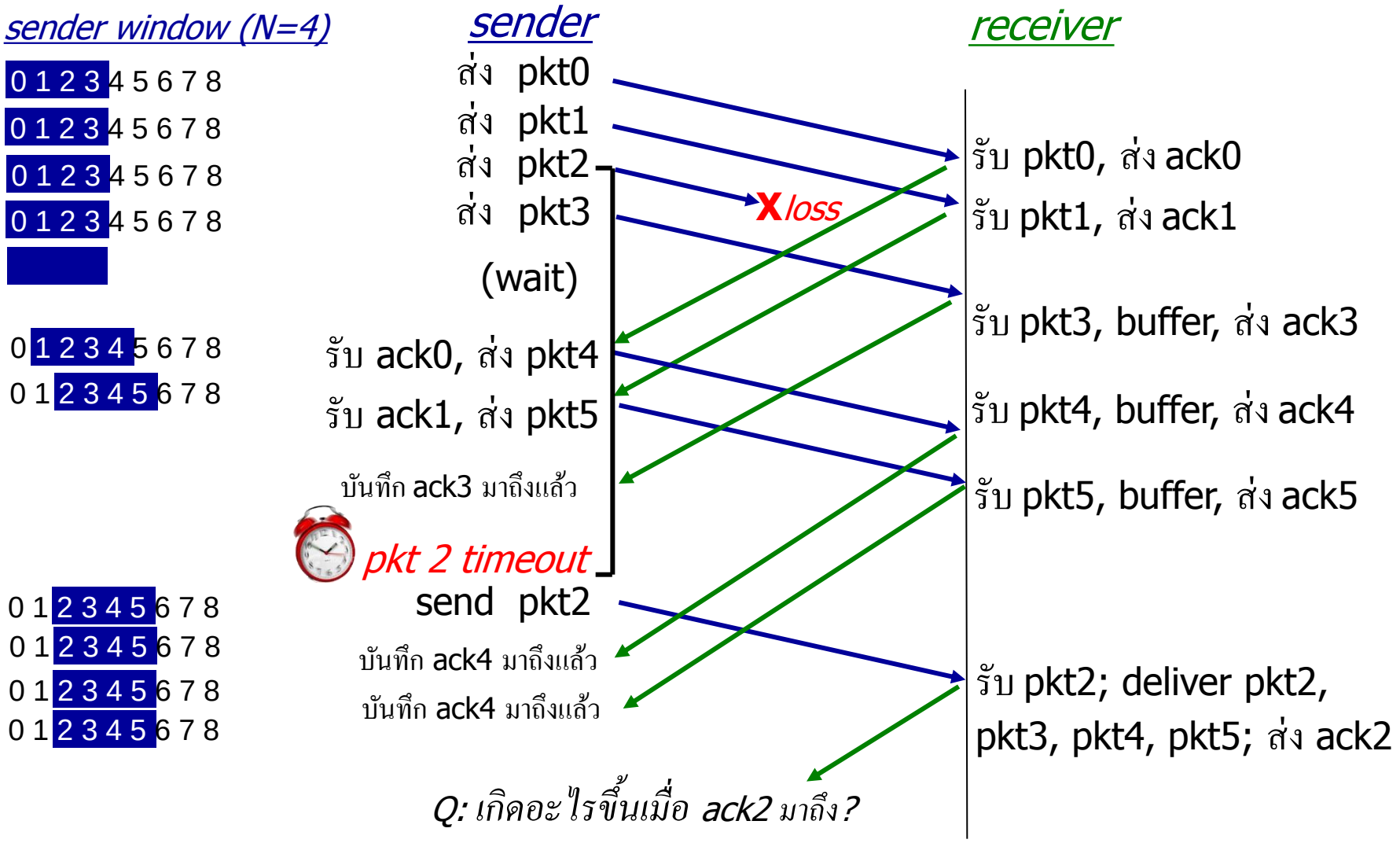
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ทิ้งไปไม่ต้องสนใจ

ขณะ Selective repeat ดำเนินการ

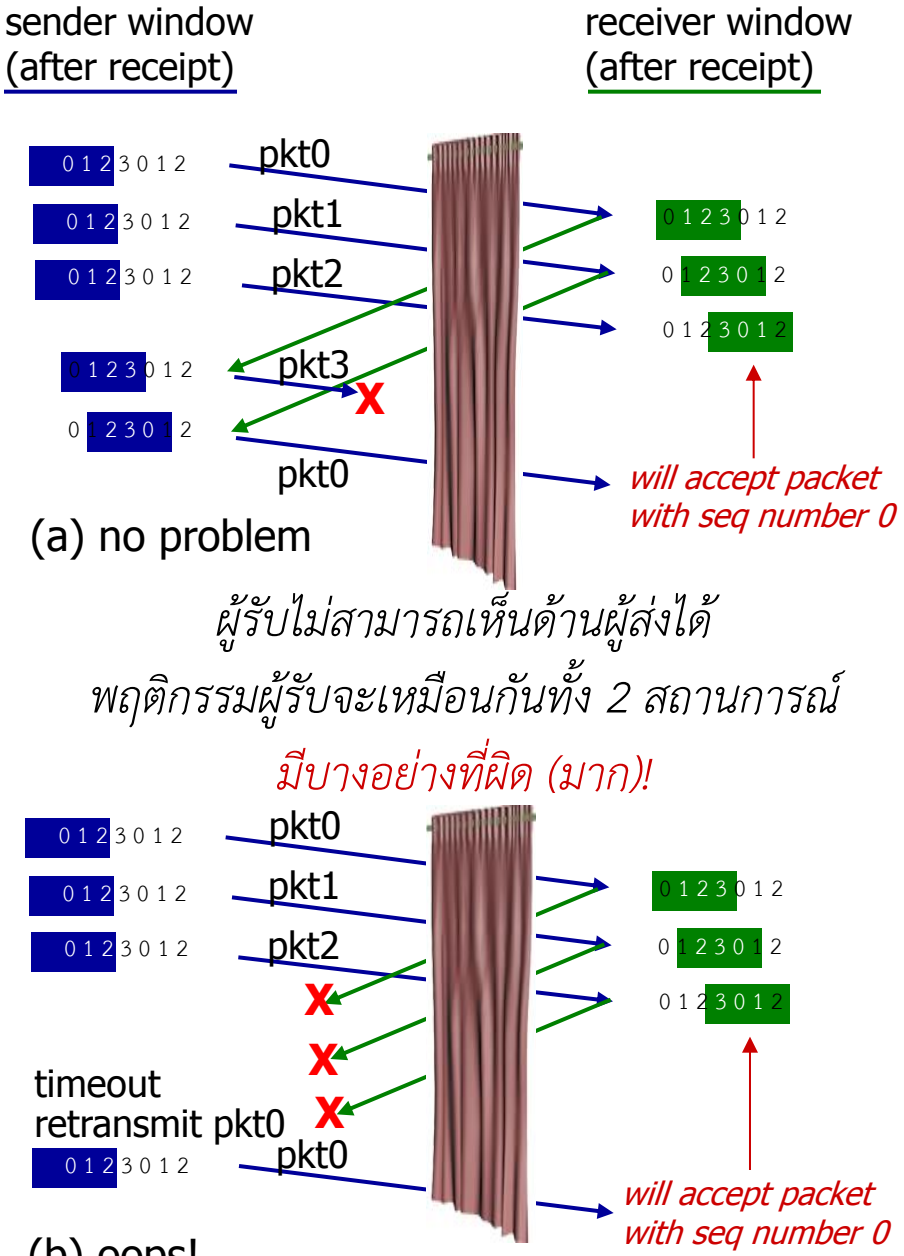


Selective repeat: dilemma

ตัวอย่าง:

- หมายเลข seq: 0, 1, 2, 3
- ขนาดของ window =3
- ผู้รับไม่เห็นถึงความแตกต่างของทั้ง 2 เหตุการณ์จำลองนี้!
- ในเหตุการณ์จำลอง (b) ข้อมูลเก่าที่ถูกส่งซ้ำ เมื่อผู้รับรับมา จะนึกว่าเป็นข้อมูลใหม่

Q: ขนาดของ seq # กับขนาดของ window เกี่ยวข้องกันยังไง เพื่อที่จะหลีกเลี่ยงปัญหาในตัวอย่าง (b) ?



Chapter 3 Outline

3.1 บริการในขั้น Transport

3.2 การรวมและการแยกข้อมูล (multiplexing and demultiplexing)

3.3 การส่งข้อมูลที่ไม่ต้องมีการเชื่อมต่อ (connectionless transport): UDP

3.4 หลักการต่าง ๆ ของการส่งข้อมูลที่ไว้ใจได้ (principles of reliable data transfer)

3.5 การส่งข้อมูลที่ต้องมีการเชื่อมต่อก่อน (connection-oriented transport): TCP

- โครงสร้างส่วนข้อมูล (segment)
- การส่งข้อมูลที่น่าเชื่อถือ/น่าไว้ใจได้ (reliable data transfer)
- การควบคุมการไหล (flow control)
- การจัดการการเชื่อมต่อ (connection management)

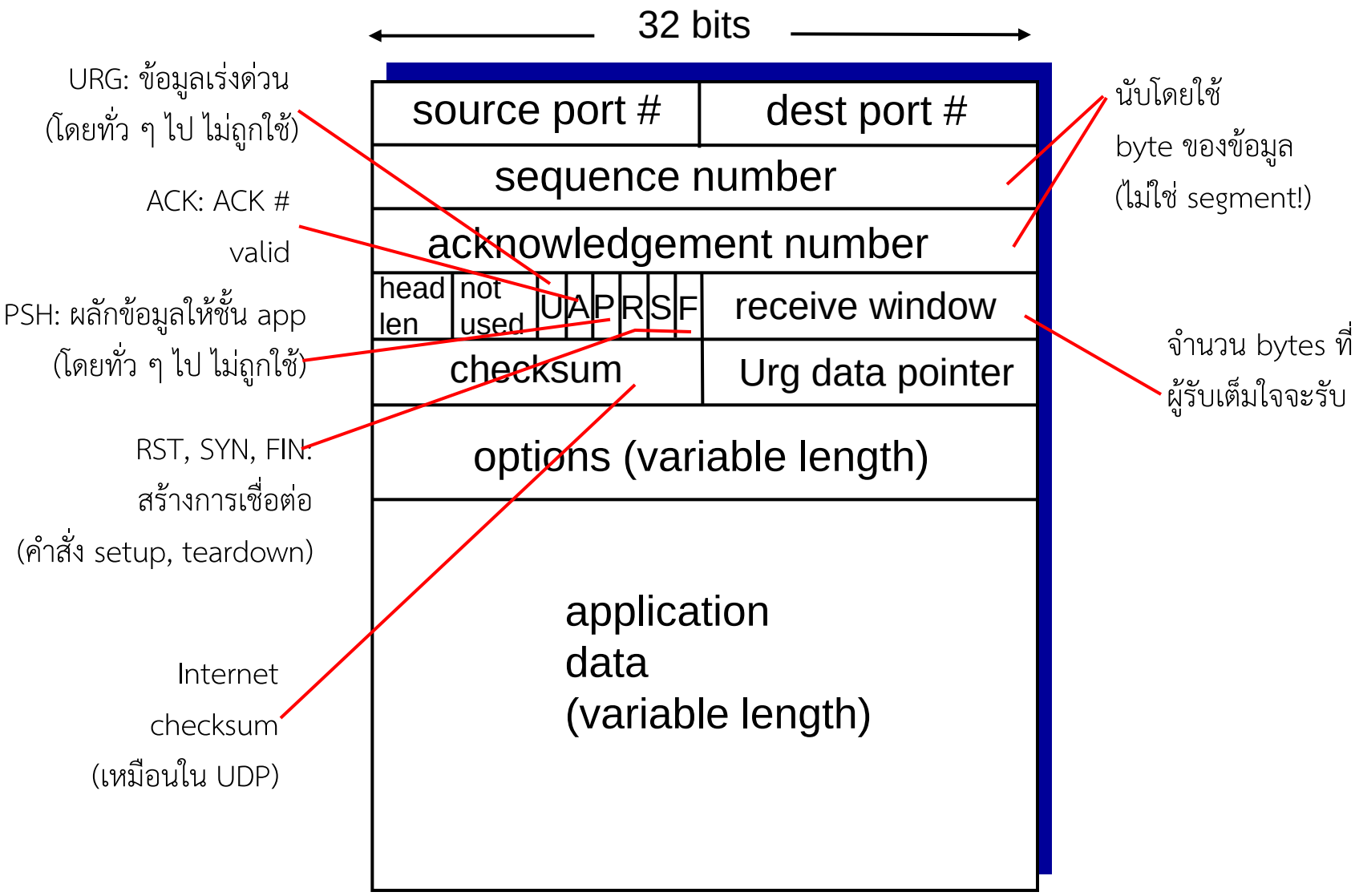
3.6 หลักการของการควบคุมความคับคั่ง (congestion control)

3.7 congestion control ของ TCP

TCP: ภาพรวม RFCs: 793,1122,1323, 2018, 2581

- point-to-point (สถานที่นึง-ไป-สถานที่นึง):
  - ผู้ส่งหนึ่งคน , ผู้รับหนึ่งคน
- สายข้อมูลเป็นไบต์ที่เรียงลำดับและข้อมูลไม่สูญหาย (reliable, in-order *byte steam):*
  - ไม่มี “ขอบเขต” ของข้อความ (ข้อความจะมาถึงผู้รับต่อเนื่องกัน)
- pipelined:
  - การควบคุมความคับคั่งและการควบคุมอัตราการไหลของ TCP จะกำหนดขนาดของ window
- ข้อมูลแบบสองทิศทาง:
  - มีการส่งข้อมูลทั้งไปและกลับในการเชื่อมต่อ (connection) เดียวกัน
  - MSS: ขนาด segment ที่ใหญ่สุดที่ TCP จะรับได้ (Max Segment Size)
- connection-oriented:
  - handshaking (การแลกเปลี่ยนของข้อความควบคุม) ตั้งค่าสถานะตั้งต้นให้ผู้ส่งและผู้รับ ก่อนการแลกเปลี่ยนข้อความ
- flow controlled:
  - ผู้ส่งจะไม่ส่งข้อมูลเกินกว่าที่ผู้รับจะรับได้

TCP segment structure



TCP seq. numbers, ACKs

หมายเลข sequence:

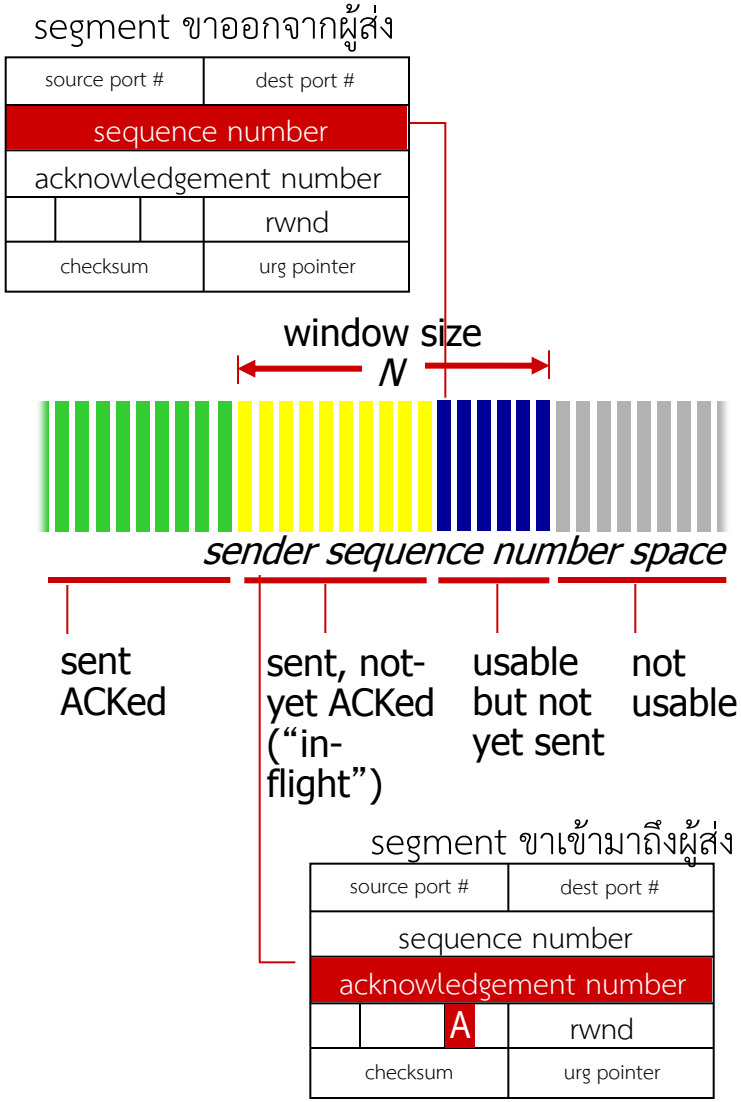
- เลขของ byte stream จะเป็น byte แรกของข้อมูลของ segment นี้

acknowledgements:

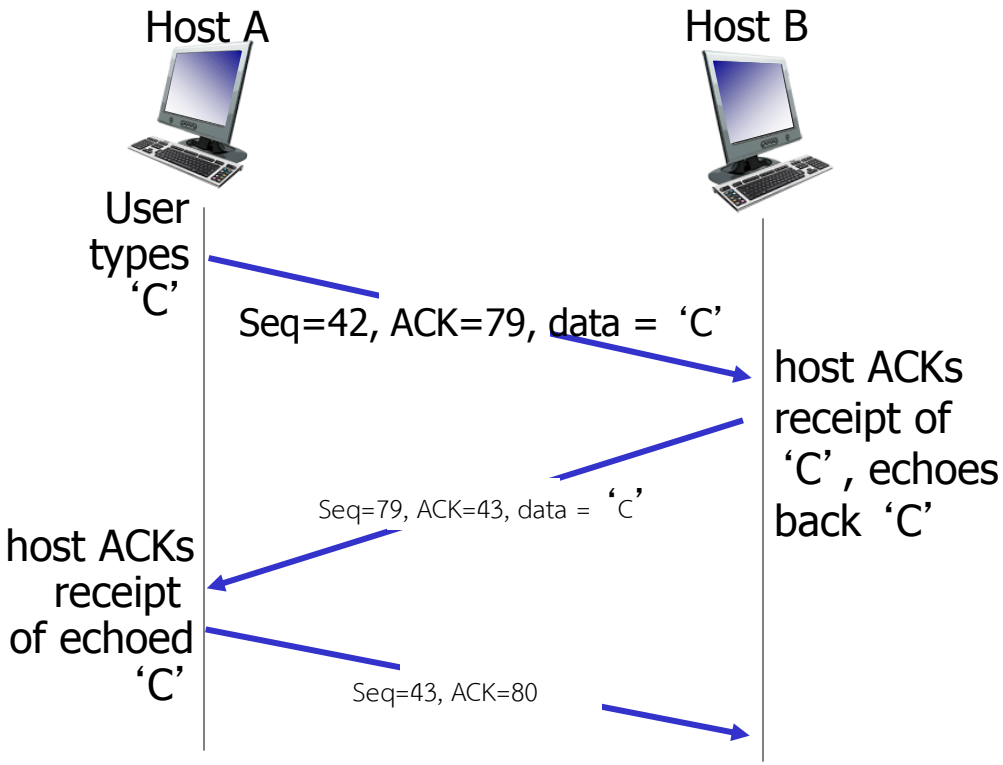
- เลข seq ของ byte ถัดไปที่อีกด้านหนึ่งคาดหวังว่าจะได้
- cumulative ACK (ACK แบบรวม)

Q: ผู้รับจัดการกับ segment ที่มาถึงไม่เรียงลำดับอย่างไร

- A: TCP ไม่มีรายละเอียดส่วนนี้ – ขึ้นอยู่กับคนที่จะมาเขียนโปรแกรม TCP



TCP seq. numbers, ACKs



เหตุการณ์จำลอง telnet อย่างง่าย

TCP round trip time, timeout

- Q: เราควรจะกำหนด time out ไว้ที่เท่าใด?
- มากกว่าเวลาของ RTT
  - แต่เวลาของ RTT นั้นแปรปรวน
- กำหนดไว้น้อยเกินไป: timeout ก่อนกำหนด ทำให้มีการ retransmissions โดยไม่จำเป็น
- กำหนดไว้มากเกินไป: ระบบตอบสนองต่อการสูญหายของ segment ช้าเกินไป

- Q: จะประมาณเวลาของ RTT อย่างไร?
- **SampleRTT**: วัดจากระยะเวลาตั้งแต่เริ่มส่ง sengment จนกระทั่งได้รับ ACK
  - ไม่สนใจการ retransmissions
- **SampleRTT** มีค่าแปรปรวน, เราต้องการประมาณเวลาของ RTT ให้ smooth กว่า
  - ใช้ค่าเฉลี่ยของค่าหลาย ๆ ค่า ไม่ใช่แค่เวลาของ **SampleRTT** ปัจจุบัน

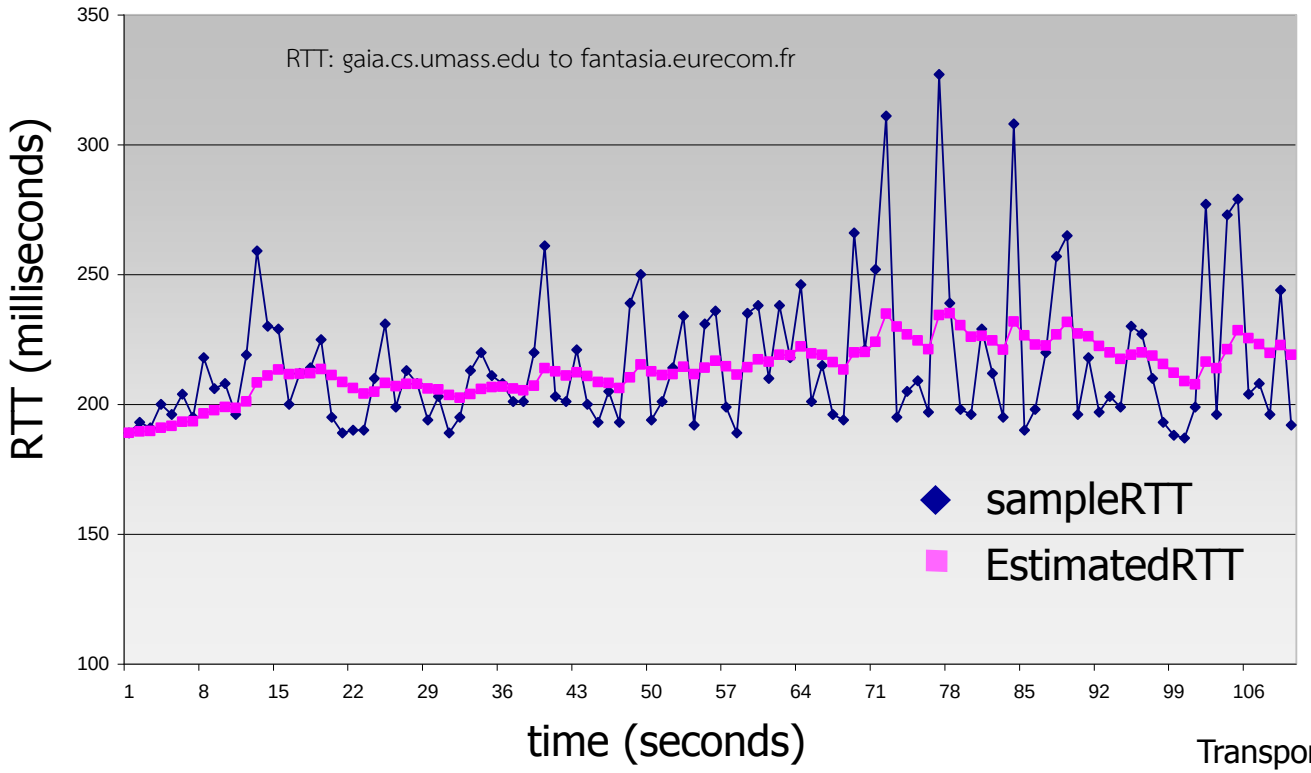
TCP round trip time, timeout

EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT

ค่าเฉลี่ยของการเคลื่อนที่ถ่วงน้ำหนักแบบ exponential

ค่า SampleRTT ที่ในอดีตจะมีผลต่อ EstimatedRTT ลดลงแบบ exponentially

โดยทั่วไปจะกำหนดค่า $\alpha = 0.125$



TCP round trip time, timeout

- ช่วงเวลาของ timeout : **EstimatedRTT** บวกกับระยะเวลาที่เผื่อเอาไว้
  - ยิ่งค่าของ EstimatedRTT ที่แปรปรวนมากเท่าไร -> ระยะเวลาที่เผื่อความมากขึ้นตาม
- มีประเมินค่าการเปลี่ยนแปลงของ SampleRTT จาก EstimatedRTT:

$$DevRTT = (1-\beta)*DevRTT + \beta*|SampleRTT-EstimatedRTT|$$

(typically, $\beta = 0.25$)

$$TimeoutInterval = EstimatedRTT + 4*DevRTT$$

estimated RTT

“ช่วงเวลาที่เผื่อ”

Chapter 3 Outline

3.1 บริการในขั้น Transport

3.2 การรวมและการแยกข้อมูล (multiplexing and demultiplexing)

3.3 การส่งข้อมูลที่ไม่ต้องมีการเชื่อมต่อ (connectionless transport): UDP

3.4 หลักการต่าง ๆ ของการส่งข้อมูลที่ไว้ใจได้ (principles of reliable data transfer)

3.5 การส่งข้อมูลที่ต้องมีการเชื่อมต่อก่อน (connection-oriented transport): TCP

- โครงสร้างส่วนข้อมูล (segment)
- การส่งข้อมูลที่น่าเชื่อถือ/น่าไว้ใจได้ (reliable data transfer)
- การควบคุมการไหล (flow control)
- การจัดการการเชื่อมต่อ (connection management)

3.6 หลักการของการควบคุมความคับคั่ง (congestion control)

3.7 congestion control ของ TCP

TCP reliable data transfer

- TCP สร้างบริการแบบ reliable อยู่บนบริการของ IP ที่เป็นแบบ unreliable
  - segments ถูกส่งแบบ pipeline
  - cumulative acks
  - มี timer สำหรับส่งใหม่ค่าตัวเดียว
- TCP ส่ง segment ใหม่เมื่อ:
  - เหตุการณ์ timeout
  - ได้รับ acks ซ้ำ ๆ กัน

เรามาเริ่มต้นพิจารณาทางด้านผู้ส่งของ TCP อย่างง่ายก่อน:

- ยังไม่ต้องสนใจ acks ซ้ำ
- ยังไม่ต้องสนใจการควบคุม flow และความคับคั่ง

เหตุการณ์ของ TCP sender:

เมื่อได้ข้อมูลจาก app:

- สร้าง segment ด้วยหมายเลข seq
- เลข seq คือหมายเลขของไบต์แรกของข้อมูลในแต่ละ segment
- หาก timer ยังไม่เคยถูกตั้ง จะเริ่มตั้งเวลา
  - การตั้งเวลาเพื่อจับเวลาสำหรับ segment เก่าที่สุดยังไม่ได้รับ ack
  - ช่วงเวลาที่หมดอายุ: TimeOutInterval

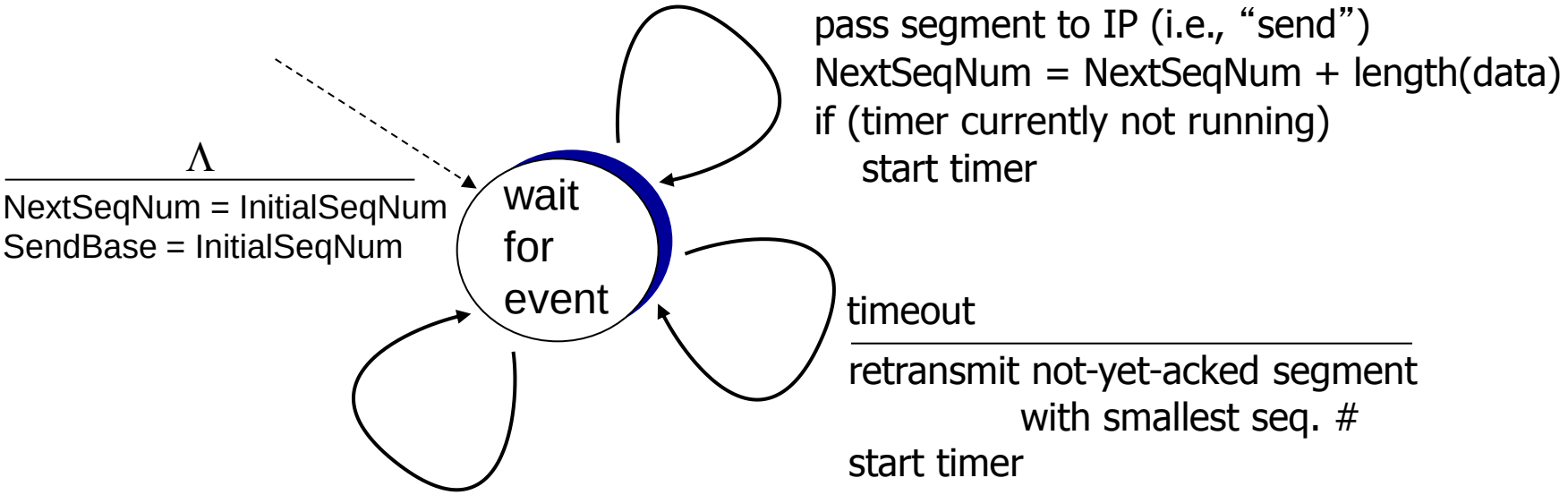
timeout:

- ส่งใหม่เฉพาะ segment ที่เกิด timeout
- ตั้งเวลานับใหม่

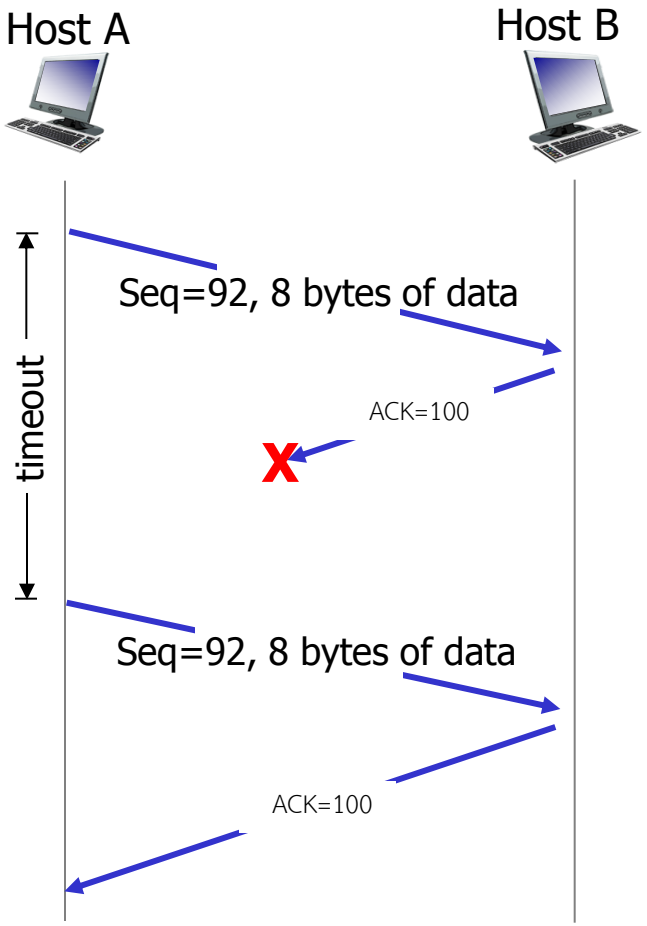
เมื่อได้รับ ack:

- ถ้า ack เป็น ack ของ segment ก่อนหน้ายังไม่ได้รับ ack
  - Update สิ่งที่รู้ว่า ถูก ACK แล้ว
  - เริ่มตั้งเวลานับ ถ้ายังคงมี segment ที่ยังไม่ได้รับ acked

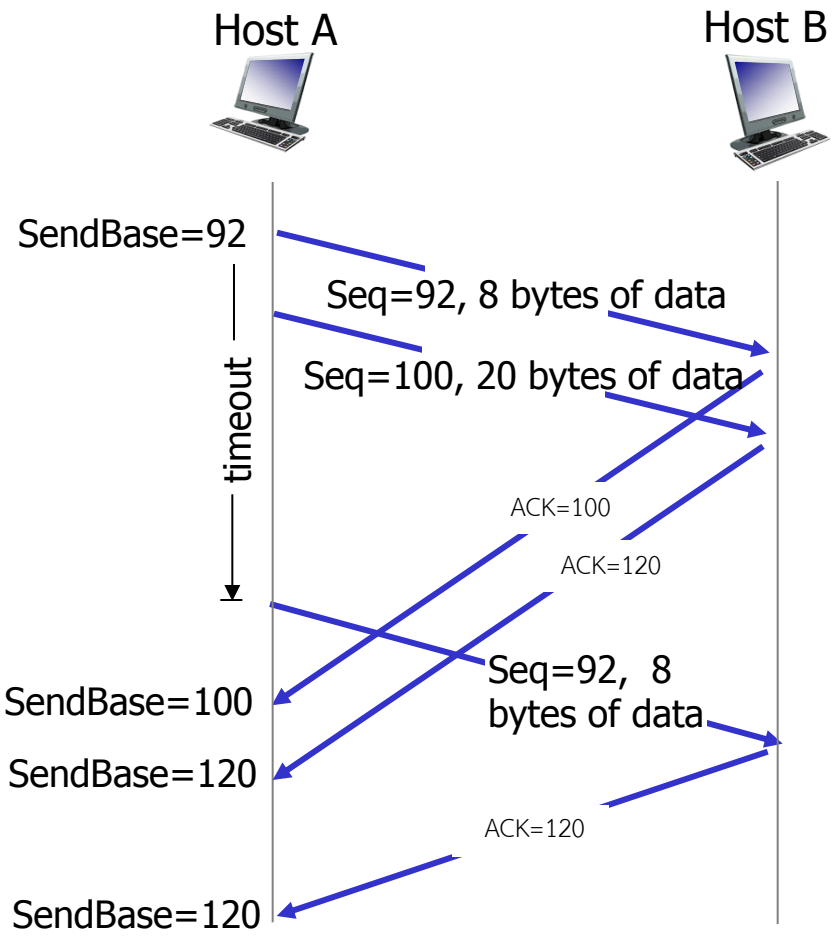
ด้านผู้ส่งของ TCP (อย่างง่าย)



TCP: retransmission scenarios

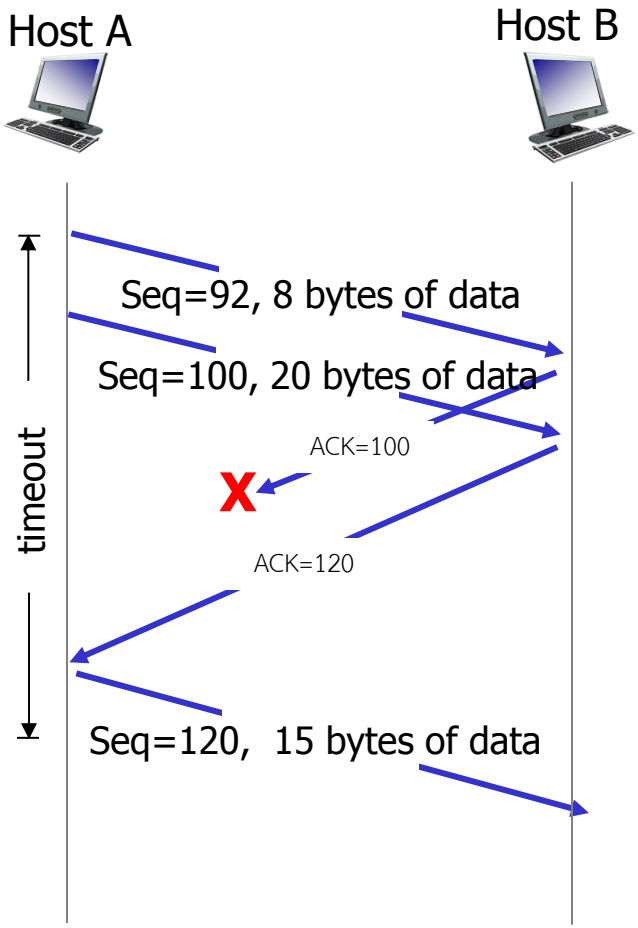


lost ACK scenario



premature timeout

TCP: retransmission scenarios



cumulative ACK

การสร้าง ACK ของ TCP [RFC 1122, RFC 2581]

เหตุการณ์ที่ผู้รับ	*TCP receiver action*
เมื่อ Segment ที่มาถึงเรียงตามลำดับ (นั่นคือ มีหมายเลข seq ที่ต้องการ) โดยข้อมูลทั้งหมดจนถึงเลข seq ที่คาดหวังไว้ ถูก ACK หมดแล้ว	อย่าเพิ่งส่ง ACK. รออีก segment นึง โดยรอสัก 500ms หากยังไม่มี segment มา ก็ส่ง ACK
เมื่อ Segment ที่มาถึงเรียงตามลำดับ (นั่นคือ มีหมายเลข seq ที่ต้องการ ทำให้ ไม่มี gap ใน segment ที่ได้มา) แต่มีบาง segment ยังไม่ได้ ACK	ส่ง ACK แบบรวม (cumulative) ไปทันทีในครั้งเดียว ซึ่งจะเป็นการ Ack segment ที่มาแบบเรียงตามลำดับทั้งสอง segments
Segment มาถึงแบบไม่เรียงตามลำดับ โดยมีเลข seq สูงกว่าที่ต้องการ ทำให้มี gap (ช่องโหว่) เกิดขึ้น	ส่ง *ACK ซ้ำ* ไปทันที โดยใส่เลข seq ของ byte ที่ต้องการ
Segment ทำมาถึง ที่จะเติมบางส่วนหรือเติมเต็มช่องโหว่	ส่ง ACK แบบรวมทันที โดย segment ที่จะ Ack เริ่มจากด้านที่น้อยกว่าของ gap

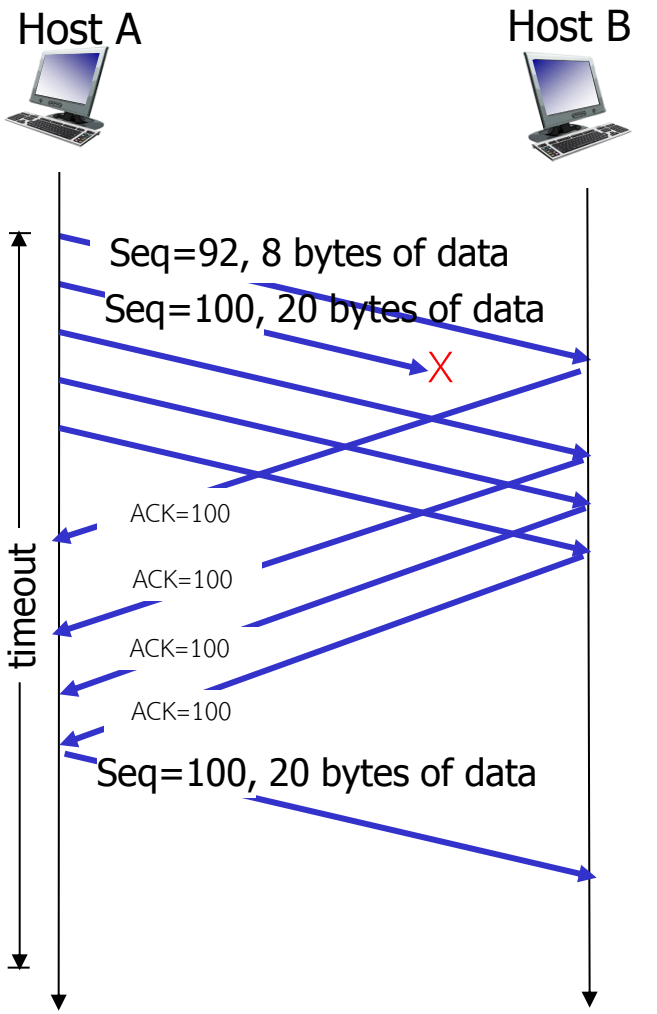
TCP fast retransmit

- ระยะเวลาในการ time-out มักจะค่อนข้างยาว (โดยเปรียบเทียบ):
  - ทำให้ใช้เวลาค่อนข้างนานก่อนที่จะส่ง packet อีกครั้ง
- แก้โดย ตรวจสอบการสูญเสีย pack โดยดูที่ ACKs ที่ซ้ำ ๆ กัน
  - ผู้ส่งมักส่ง segment ติด ๆ กัน
  - ดังนั้น ถ้า segment หนึ่งหายไป, จึงเป็นไปได้ที่จะมี ACKs ซ้ำ ๆ กัน จำนวนมาก

TCP fast retransmit

- ถ้าผู้ส่งได้รับ ACKs สำหรับข้อมูลเดียวกัน 3 ACKS ("ACKs ที่ซ้ำกันสามครั้ง"), ผู้ส่งต้องส่ง seq ที่ยังไม่ถูก acked (ระบุเลข seq ที่เล็กที่สุด)
  - มีแนวโน้มว่า segment ที่ยังไม่ถูก ack จะหายไปเลย, ดังนี้น จึงไม่ต้องรอให้หมดเวลา

TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

Chapter 3 Outline

3.1 บริการในขั้น Transport

3.2 การรวมและการแยกข้อมูล (multiplexing and demultiplexing)

3.3 การส่งข้อมูลที่ไม่ต้องมีการเชื่อมต่อ (connectionless transport): UDP

3.4 หลักการต่าง ๆ ของการส่งข้อมูลที่ไว้ใจได้ (principles of reliable data transfer)

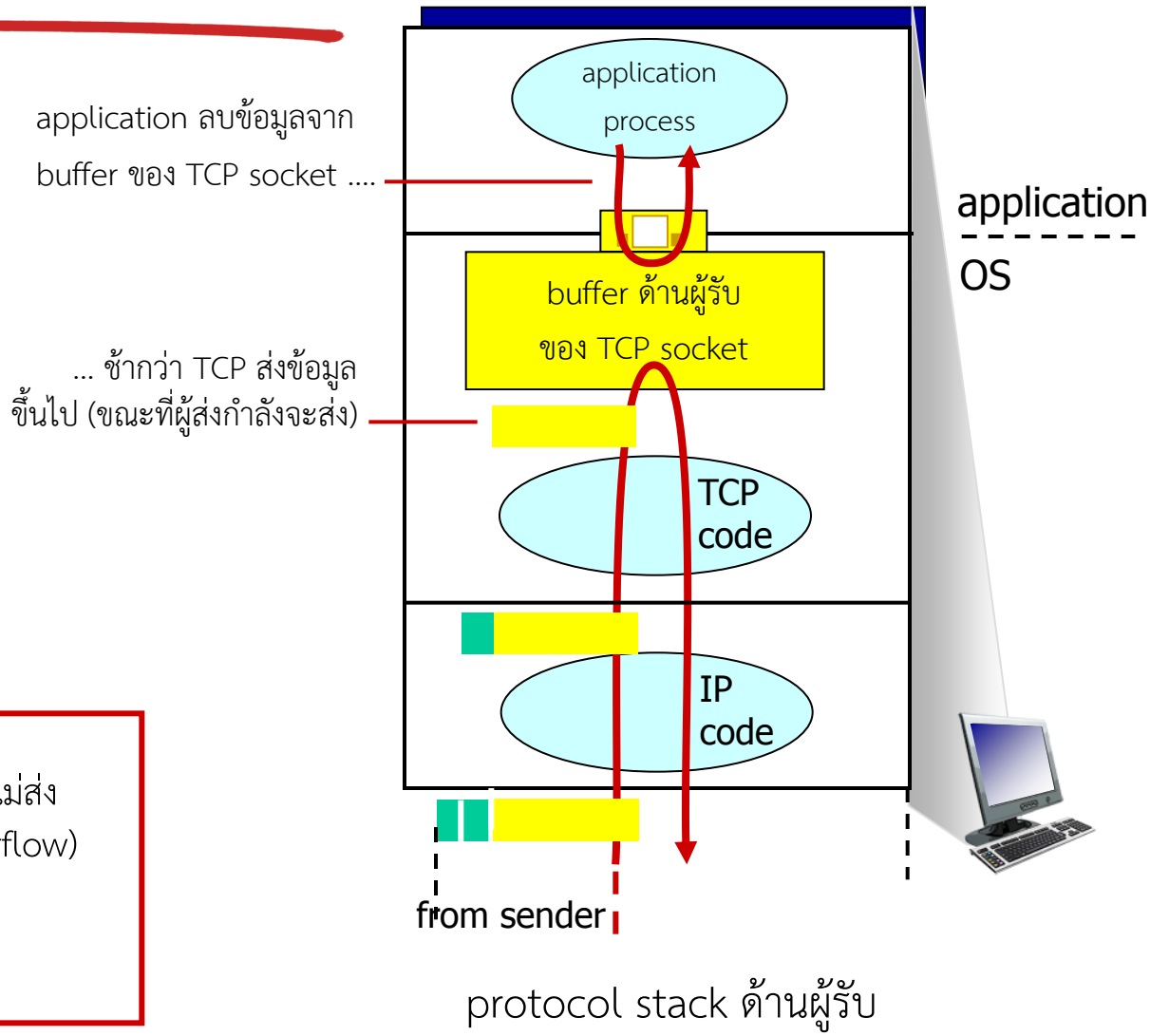
3.5 การส่งข้อมูลที่ต้องมีการเชื่อมต่อก่อน (connection-oriented transport): TCP

- โครงสร้างส่วนข้อมูล (segment)
- การส่งข้อมูลที่น่าเชื่อถือ/น่าไว้ใจได้ (reliable data transfer)
- การควบคุมการไหล (flow control)
- การจัดการการเชื่อมต่อ (connection management)

3.6 หลักการของการควบคุมความคับคั่ง (congestion control)

3.7 congestion control ของ TCP

TCP flow control

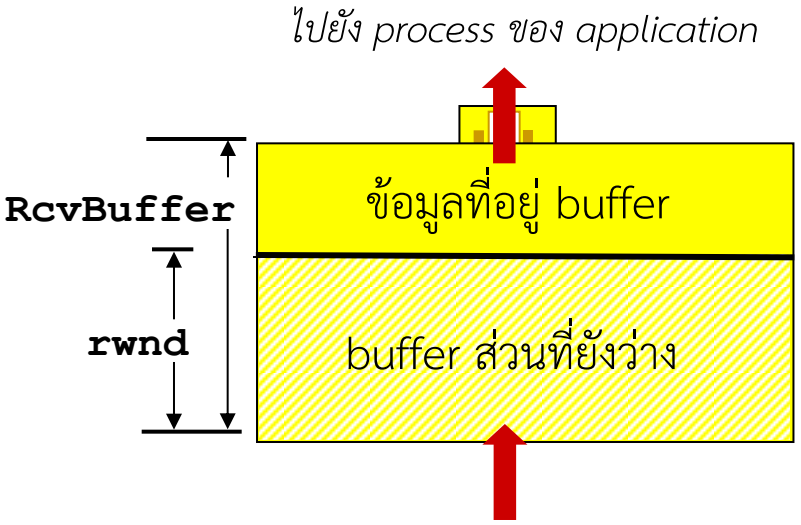


flow control

ผู้รับสามารถควบคุมผู้ส่งได้ ดังนั้น ผู้ส่งจะไม่ส่งข้อมูลเร็วไป หรือ มากเกินไปจนล้น (Overflow) buffer ของผู้รับ

การควบคุมการไหลข้อมูลของ TCP

- ผู้รับระบุขนาดของ buffer ที่ว่างไว้ใน field rwnd ที่อยู่ในส่วนหัวของ TCP ที่จะส่งไปยังผู้ส่ง
  - ผู้เขียนโปรแกรมระบุขนาดของ RcvBuffer ได้โดยใช้ API ตั้งค่า option ของ socket (ค่าเริ่มต้นทั่วไปคือ 4096 ไบต์)
  - ทุกระบบปฏิบัติการจะปรับขนาด RcvBuffer อัตโนมัติ
- ผู้ส่งจำกัดจำนวนของข้อมูลที่ยังไม่ถูก acked (“อยู่ในระหว่างการเดินทาง”) ให้เท่ากับค่า rwnd ของข้อมูลผู้รับ
- เป็นการรับประกันว่า buffer ของผู้รับจะไม่ล้น



การbuffer ในด้านของผู้รับ

Chapter 3 Outline

3.1 บริการในขั้น Transport

3.2 การรวมและการแยกข้อมูล (multiplexing and demultiplexing)

3.3 การส่งข้อมูลที่ไม่ต้องมีการเชื่อมต่อ (connectionless transport): UDP

3.4 หลักการต่าง ๆ ของการส่งข้อมูลที่ไว้ใจได้ (principles of reliable data transfer)

3.5 การส่งข้อมูลที่ต้องมีการเชื่อมต่อก่อน (connection-oriented transport): TCP

- โครงสร้างส่วนข้อมูล (segment)
- การส่งข้อมูลที่น่าเชื่อถือ/น่าไว้ใจได้ (reliable data transfer)
- การควบคุมการไหล (flow control)
- การจัดการการเชื่อมต่อ (connection management)

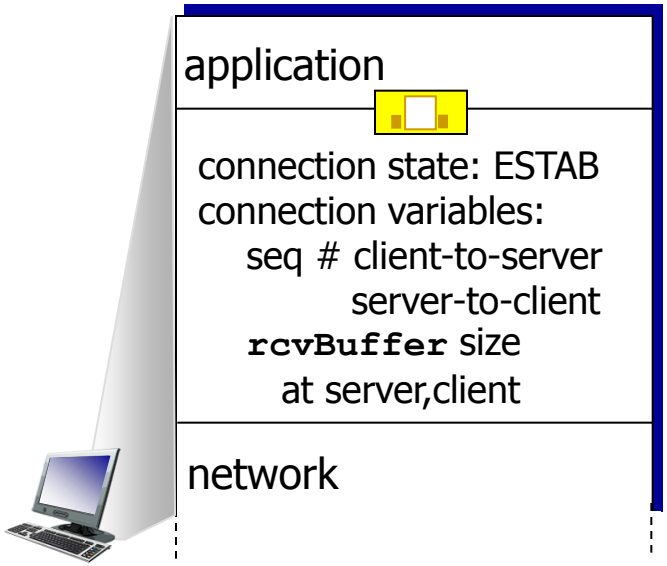
3.6 หลักการของการควบคุมความคับคั่ง (congestion control)

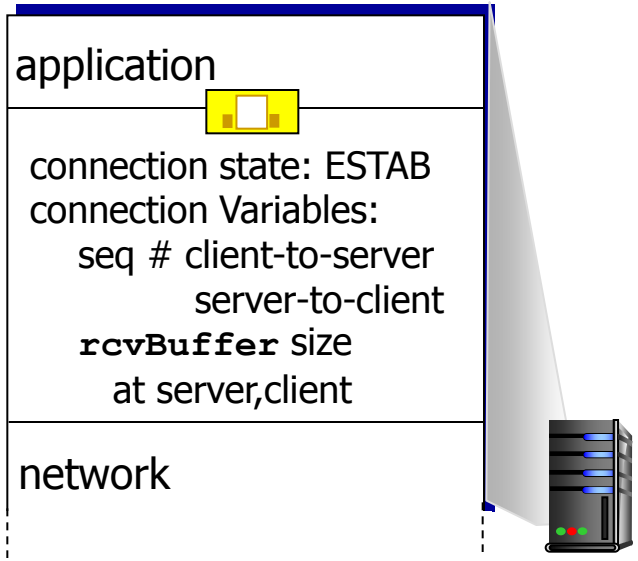
3.7 congestion control ของ TCP

การจัดการกับการเชื่อมต่อ

ก่อนที่จะมีการแลกเปลี่ยนข้อมูลกัน, ผู้ส่งและผู้รับต้องตกลงกันก่อน :

- ตกลงร่วมในการสร้างการเชื่อมต่อ (แต่ละคนจะรู้ว่าอีกฝ่ายอยากจะเชื่อมต่อหรือไม่)
- ตกลงค่าพารามิเตอร์สำหรับการเชื่อมต่อ



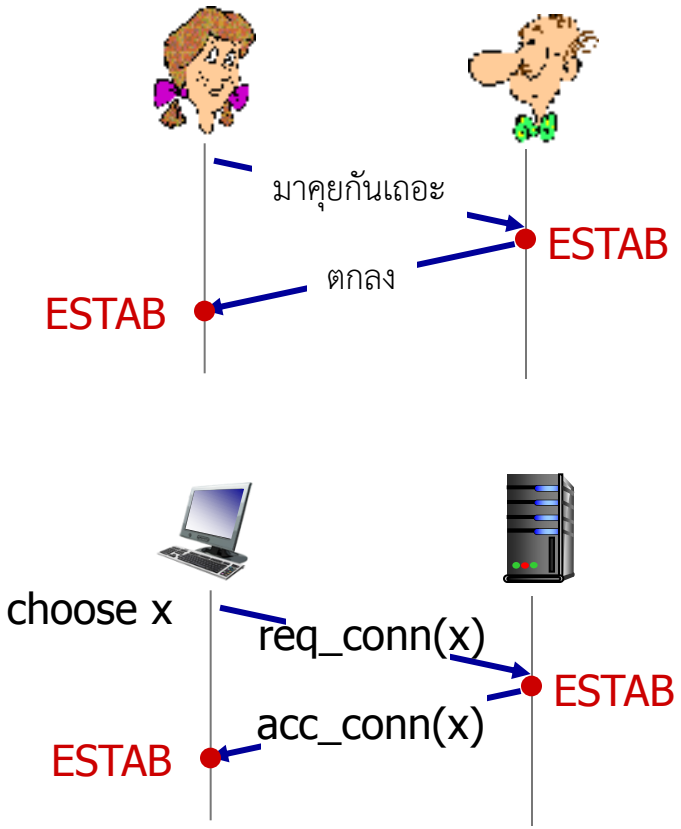



```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

```
Socket connectionSocket =
   welcomeSocket.accept();
```

ตกลงที่จะสร้างการเชื่อมต่อ

2-way handshake:

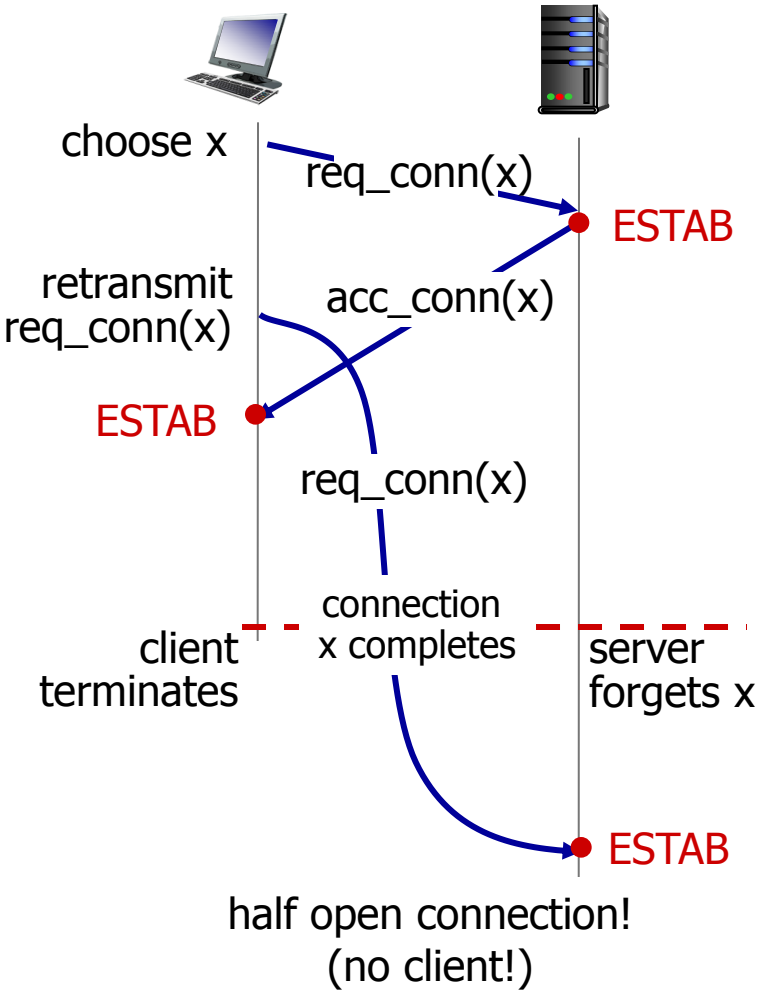


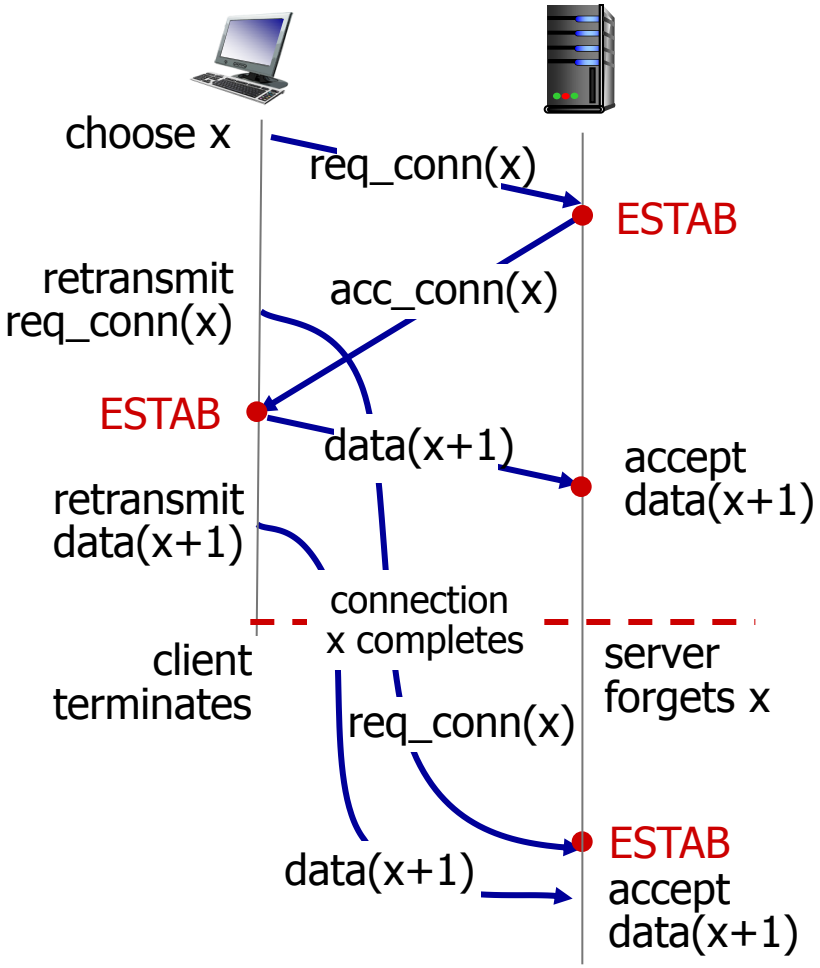
Q: 2-way handshake จะทำงานได้เสมอไปหรือไม่บน network?

- ความล่าช้าอาจเปลี่ยนไป
- ข้อความอาจถูกส่งซ้ำตัวอย่าง (เช่น req_conn(x)) เนื่องจากการสูญหายของข้อมูล
- ข้อมูลอาจสลับกัน
- ไม่สามารถรู้ได้ว่าอีกฝั่งได้รับหรือไม่ได้รับข้อมูลอะไร

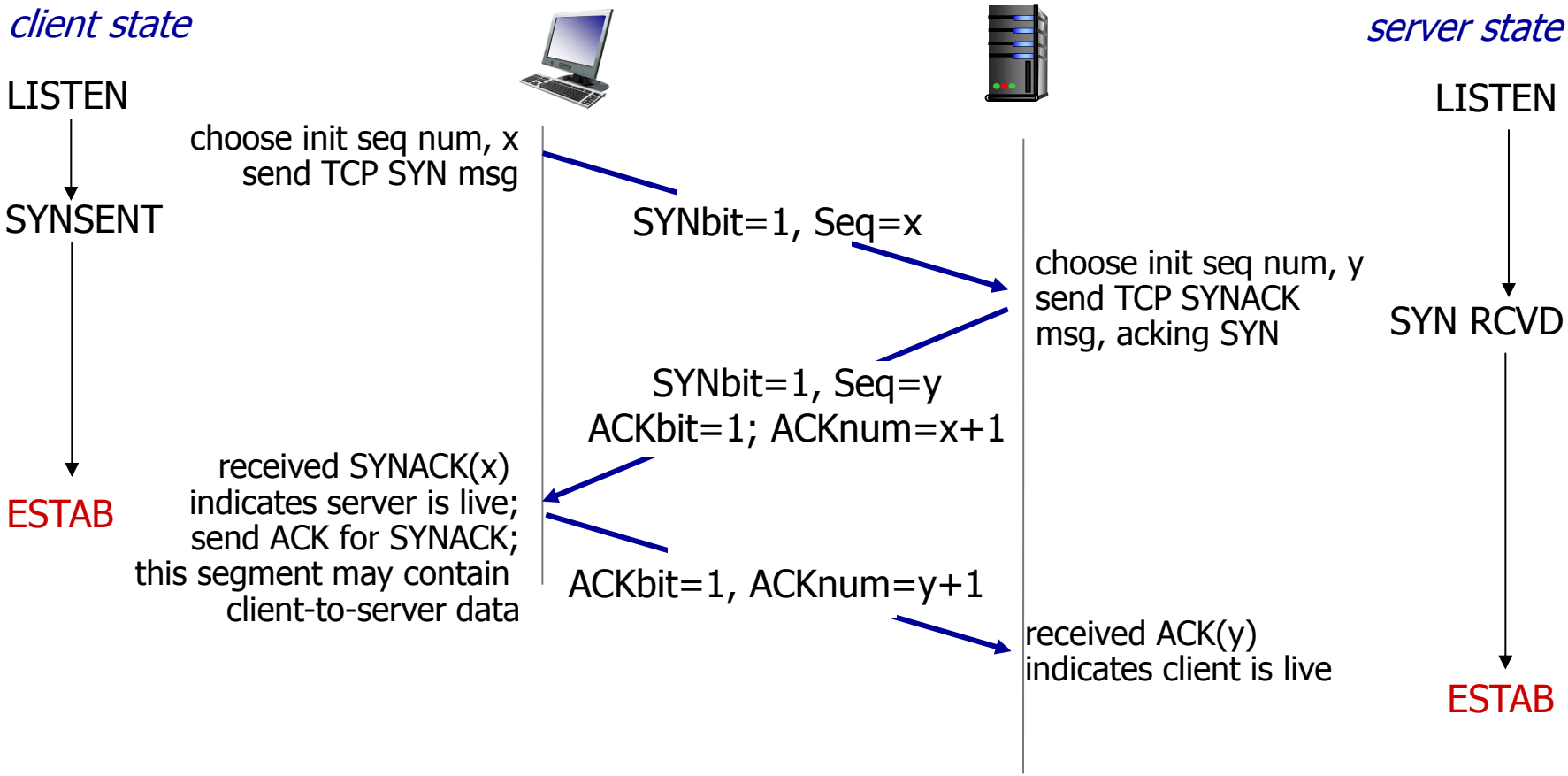
ตกลงที่จะสร้างการเชื่อมต่อ

เหตุการณ์ผิดพลาดของ 2-way handshake:

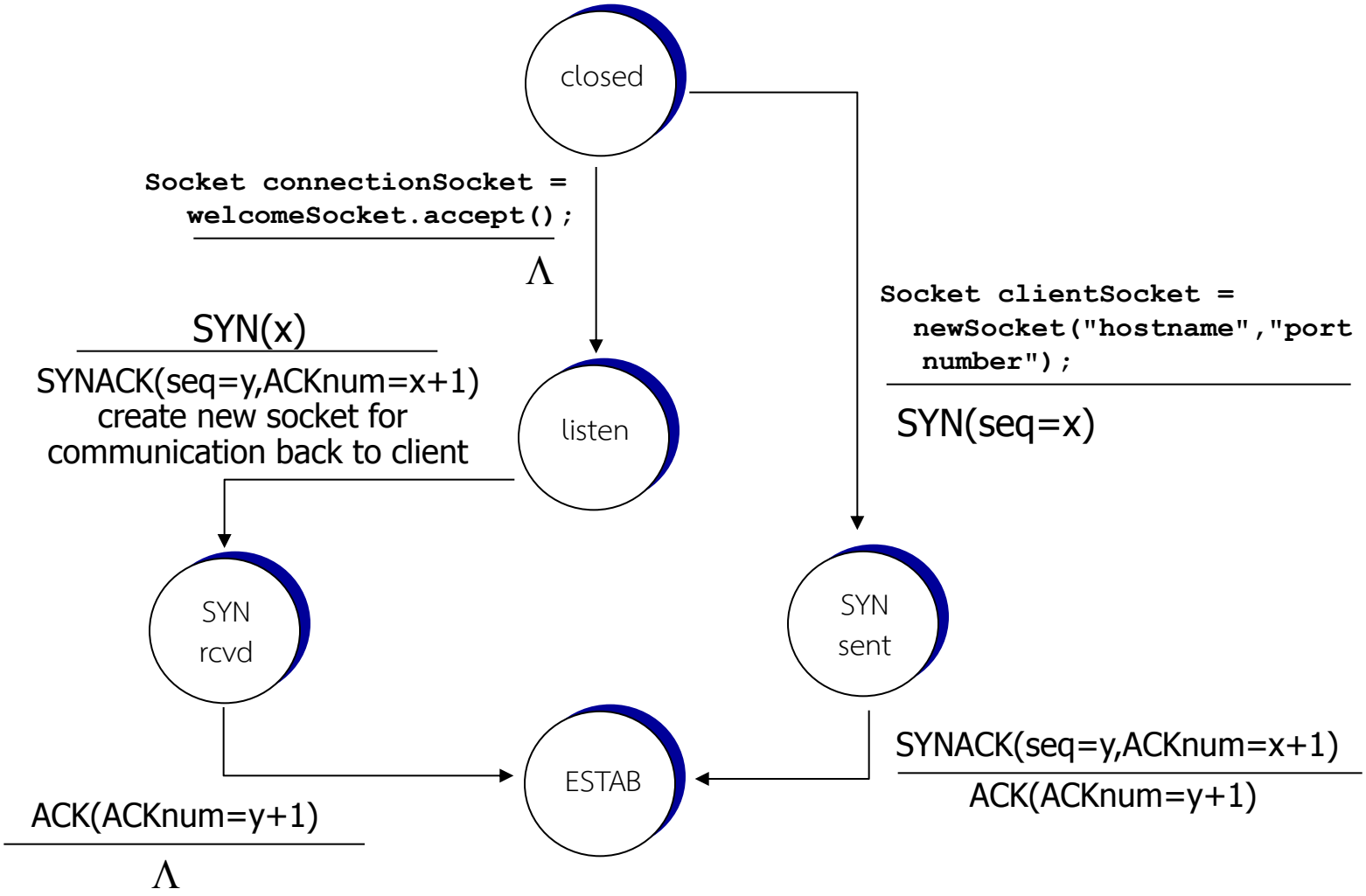




3-way handshake ของ TCP



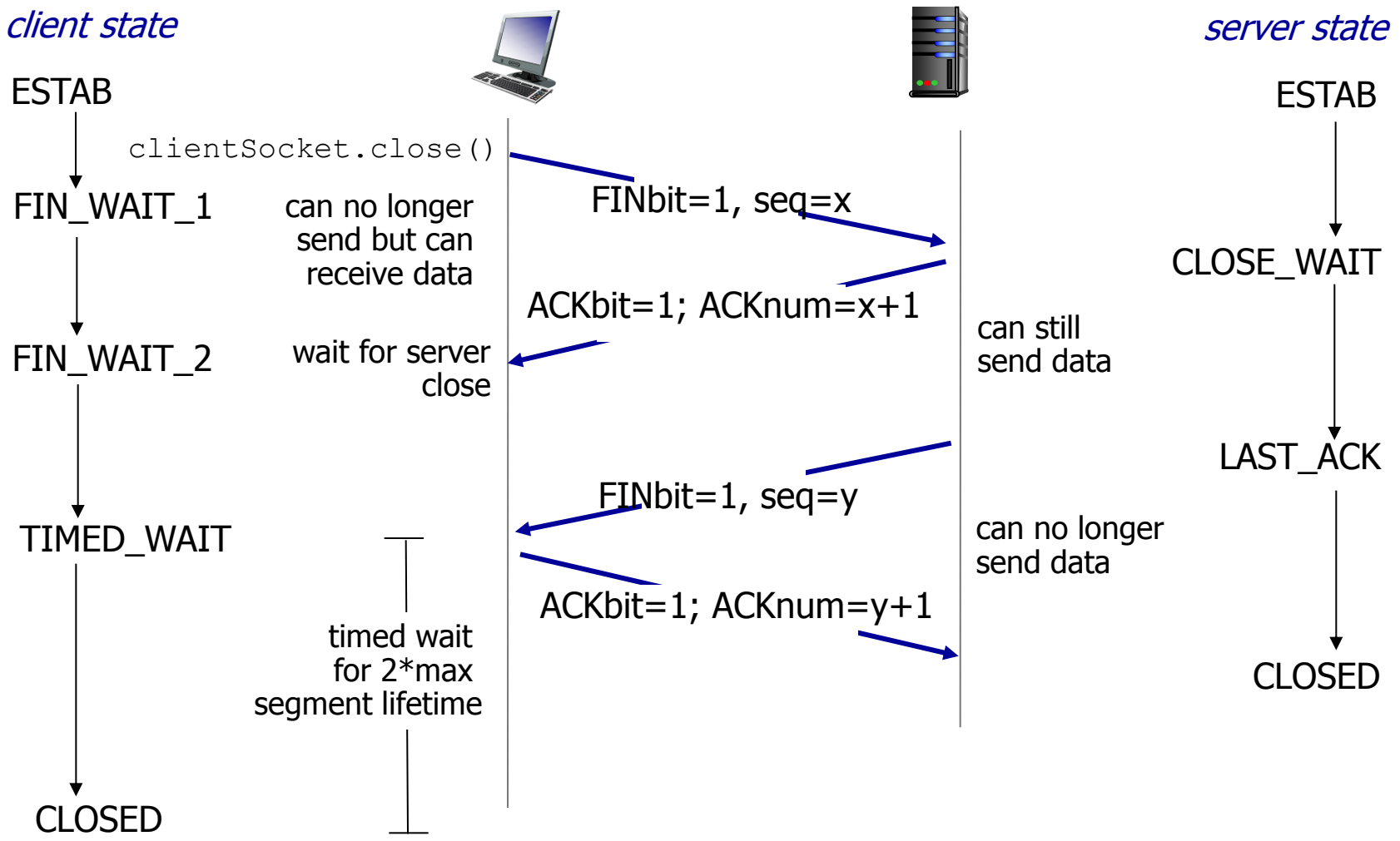
FSM ของ 3-way handshake ของ TCP



TCP: การหยุดการเชื่อมต่อ

- client, server เลิกเชื่อมต่อที่แต่ละด้านของการเชื่อมต่อ
  - โดยแต่ละด้านจะส่ง TCP segment ด้วย FIN bit = 1
- ตอบกลับ FIN ด้วย ACK
  - ในขณะที่รับ FIN, ACK ก็สามารถรวมอยู่ในแพคเกจที่ส่ง FIN มาได้ด้วย
- แต่ละด้านสามารถส่ง FIN ได้พร้อม ๆ กัน

TCP: การหยุดการเชื่อมต่อ



Chapter 3 Outline

3.1 บริการในขั้น Transport

3.2 การรวมและการแยกข้อมูล (multiplexing and demultiplexing)

3.3 การส่งข้อมูลที่ไม่ต้องมีการเชื่อมต่อ (connectionless transport): UDP

3.4 หลักการต่าง ๆ ของการส่งข้อมูลที่ไว้ใจได้ (principles of reliable data transfer)

3.5 การส่งข้อมูลที่ต้องมีการเชื่อมต่อก่อน (connection-oriented transport): TCP

- โครงสร้างส่วนข้อมูล (segment)
- การส่งข้อมูลที่น่าเชื่อถือ/น่าไว้ใจได้ (reliable data transfer)
- การควบคุมการไหล (flow control)
- การจัดการการเชื่อมต่อ (connection management)

3.6 หลักการของการควบคุมความคับคั่ง (congestion control)

3.7 congestion control ของ TCP

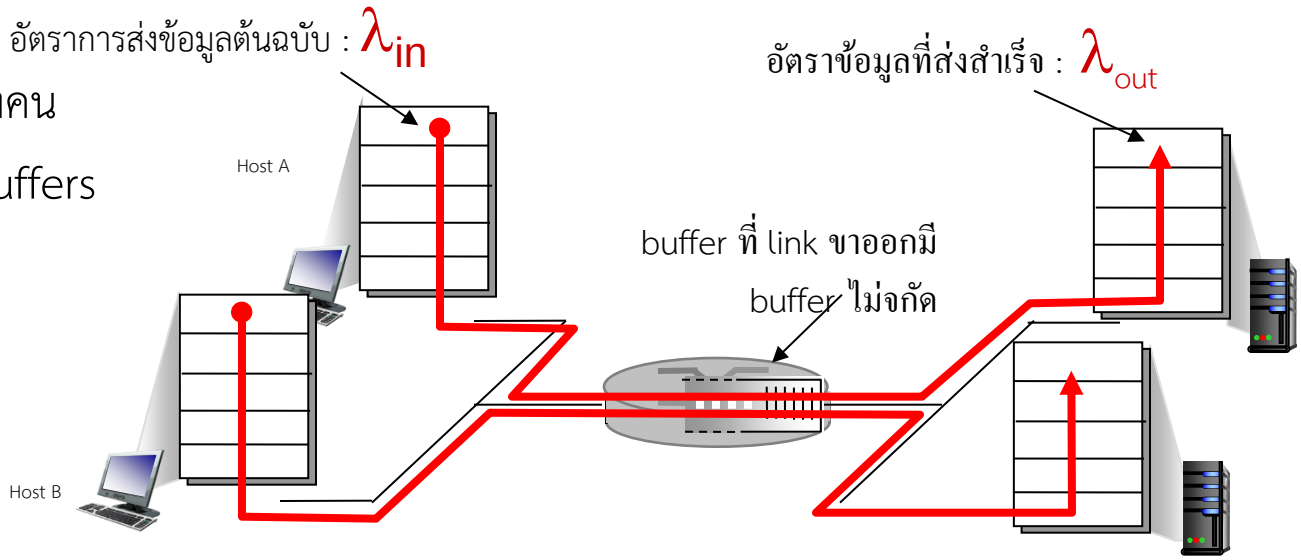
หลักการของการควบคุมความคับคั่งของเครือข่าย

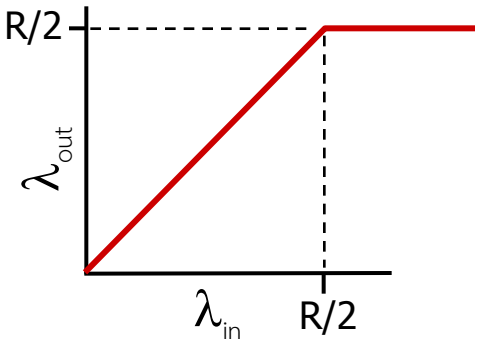
ความคับคั่ง:

- ความหมายไม่เป็นทางการ: “เมื่อจำนวนคนส่งมากเกินไป กำลังส่งข้อมูลมากเกินไปด้วยอัตราการส่งเร็วเกินกว่าที่ Network จะรับไหว”
- แตกต่างจาก flow control !
- ผลที่อาจที่เกิดขึ้นจากความคับคั่ง :
  - packets หาย (buffer ล้นที่ routers)
  - ล่าช้ามาก (เข้าคิวใน buffers ของ router)
- เป็นปัญหาที่คนสนใจติด 10 อันดับต้นๆ

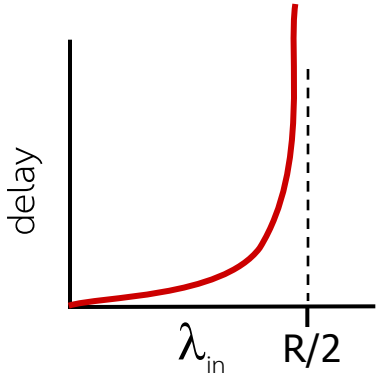
ต้นเหตุและผลของความคับคั่ง: เหตุการณ์ 1

- มีผู้ส่งและผู้รับอย่างละสองคน
- Router 1 ตัว, ไม่จำกัด buffers
- ความจุของ link: R
- ไม่มีการส่งข้อมูลซ้ำ





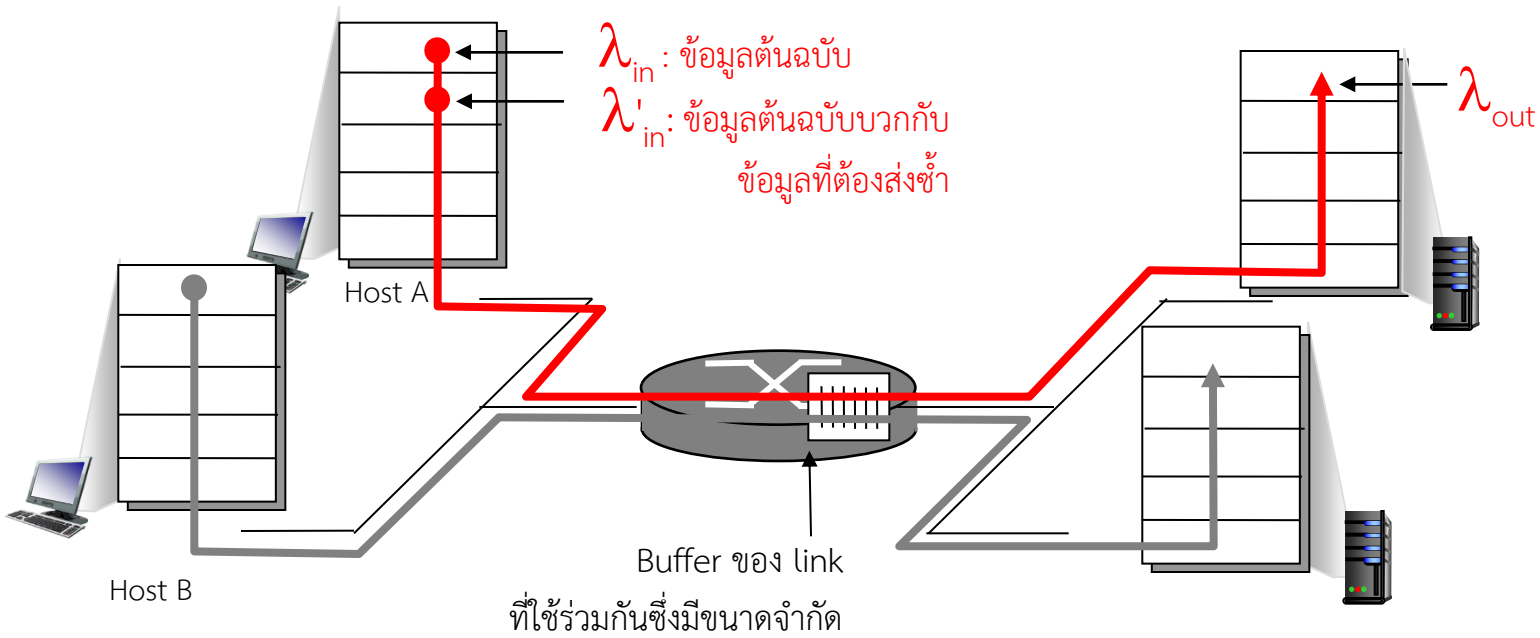
- อัตราข้อมูลที่ส่งสำเร็จต่อหนึ่งการเชื่อมต่อ: R/2



- ดีเลย์จะมากขึ้นเมื่ออัตราส่งข้อมูลต้นฉบับเข้าใกล้ความจุของ link

ต้นเหตุและผลของความคับคั่ง: เหตุการณ์ 2

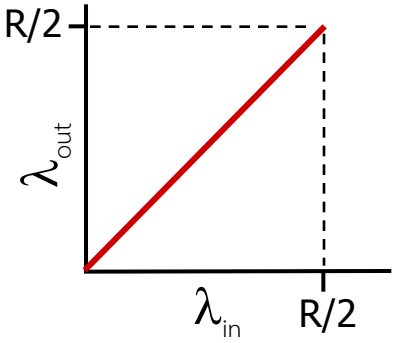
- Router 1 ตัว Buffer แต่มีขนาด buffer จำกัด
- ต้องส่ง pkt ไปซ้ำถ้า pkt นั้น timeout
  - ข้อมูลเข้าของชั้น application = ข้อมูลออกจากชั้น application: $\lambda_{in} = \lambda_{out}$
  - ข้อมูลเข้าของชั้น transport จะรวมที่ต้องส่งซ้ำด้วย: $\lambda'_{in} \geq \lambda_{in}$

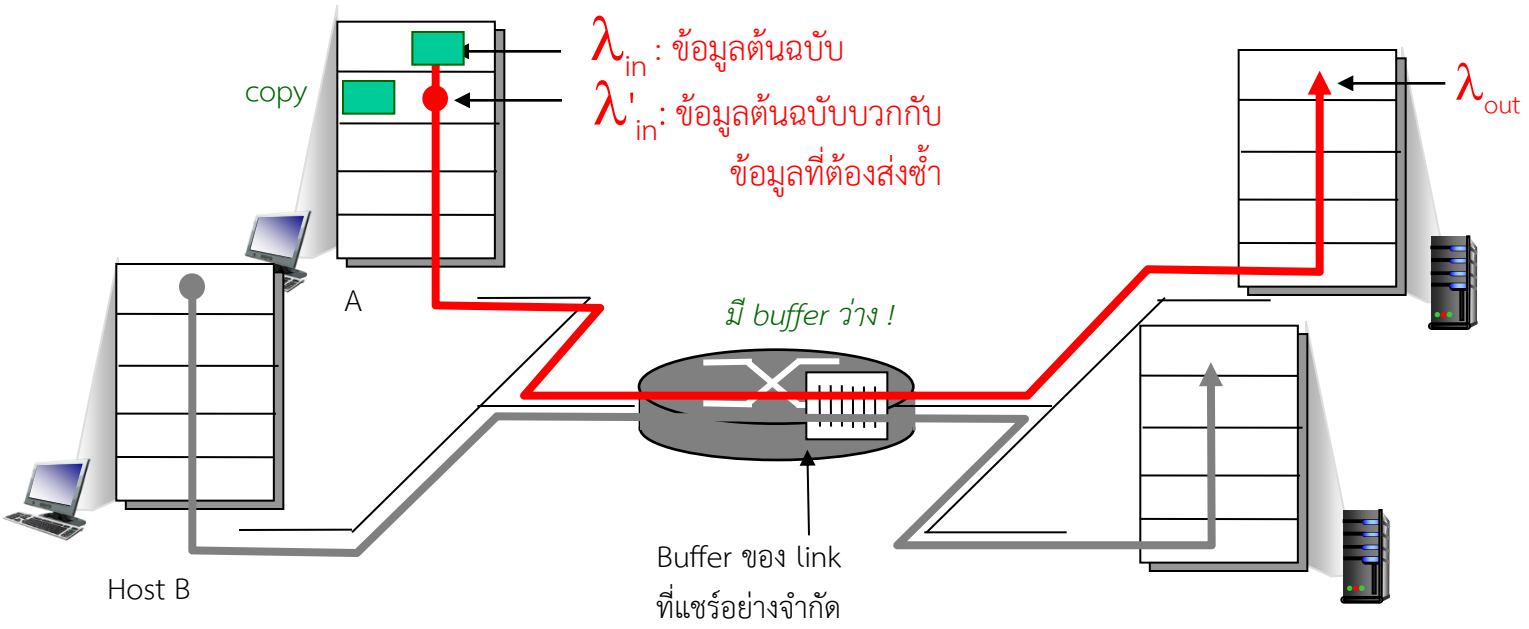


ต้นเหตุและผลของความคับคั่ง: เหตุการณ์ 2

ในอุดมคติ : สมมติผู้ส่งรู้ความเป็นไปในเครือข่าย

- ผู้ส่งส่งเฉพาะตอนที่เร้าเตอร์มี buffer เหลืออยู่

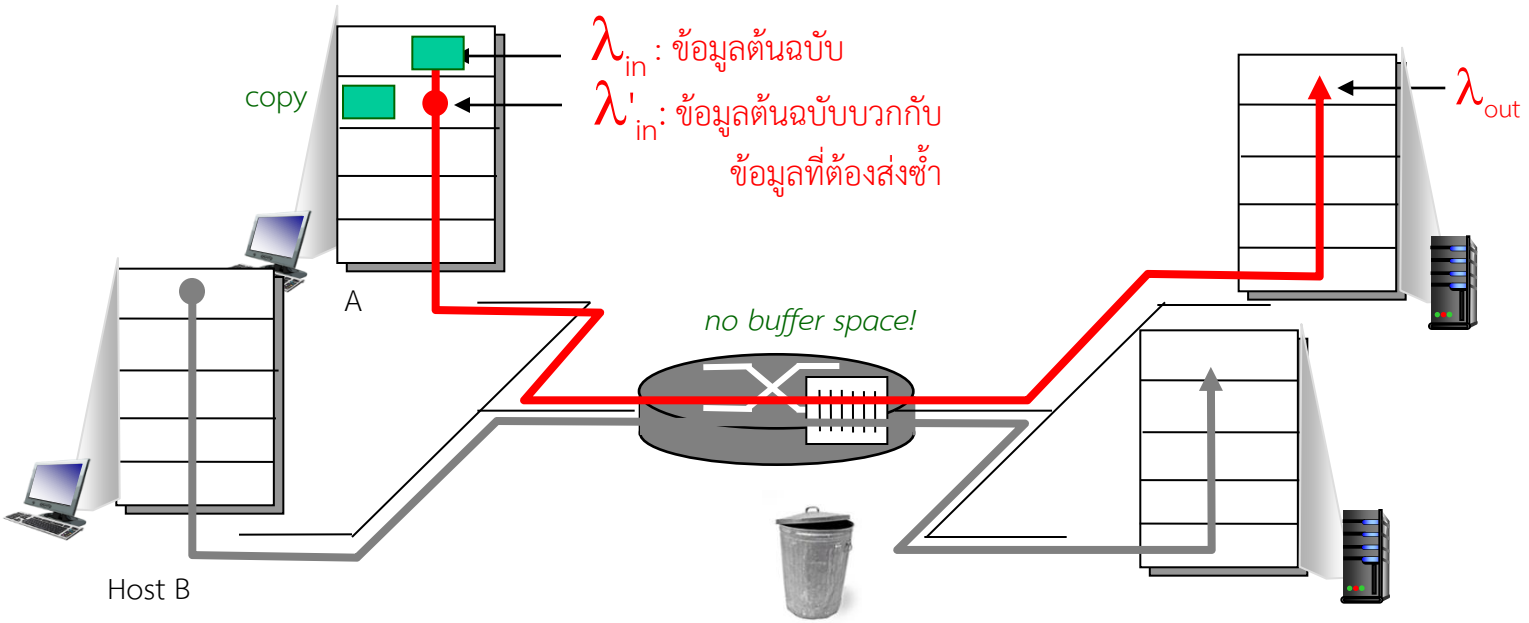




ต้นเหตุและผลของความคับคั่ง: เหตุการณ์ 2

ในอุดมคติ: สมมติว่าผู้ส่งรู้ว่าข้อมูล
*ใดหาย*จากการถูก drop ที่ router
เมื่อ buffer เต็ม

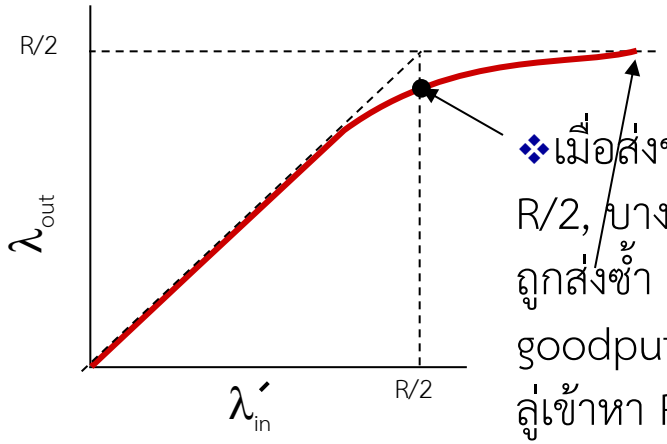
- ผู้ส่งส่งข้อมูลซ้ำก็ต่อเมื่อรู้ว่ามี Packet สูญหาย



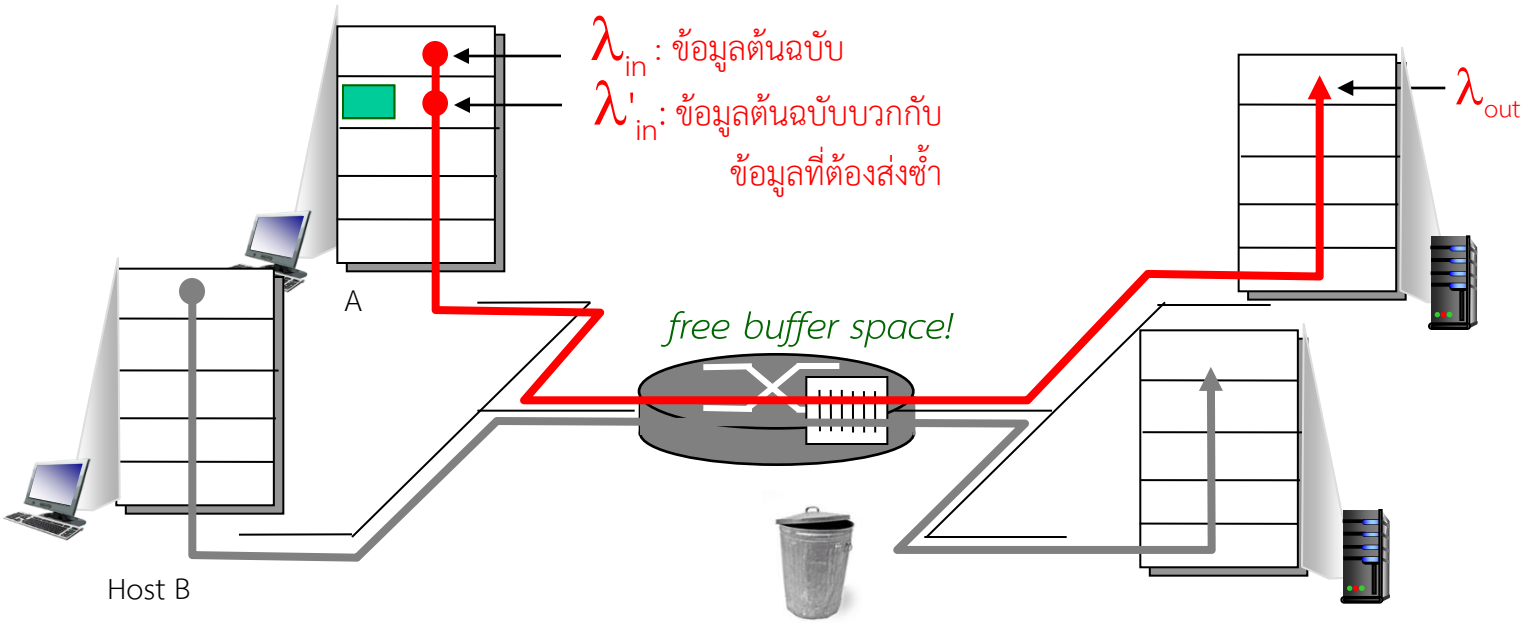
ต้นเหตุและผลของความคับคั่ง: เหตุการณ์ 2

*ในอุดมคติ: สมมติว่าผู้ส่งรู้ว่าข้อมูลใดหาย*จากการถูก drop ที่ router เมื่อ buffer เต็ม

- ผู้ส่งส่งข้อมูลซ้ำก็ต่อเมื่อรู้ว่ามี Packet สูญหาย



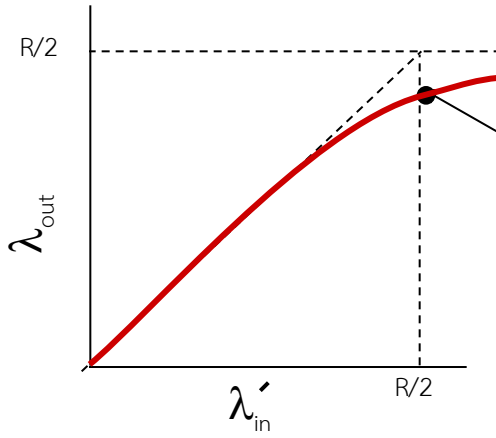
- เมื่อส่งข้อมูลที่ R/2, บาง packets ถูกส่งซ้ำ แต่ goodput ยังคงมีค่าลู่เข้าหา R/2 (ทำไม?)



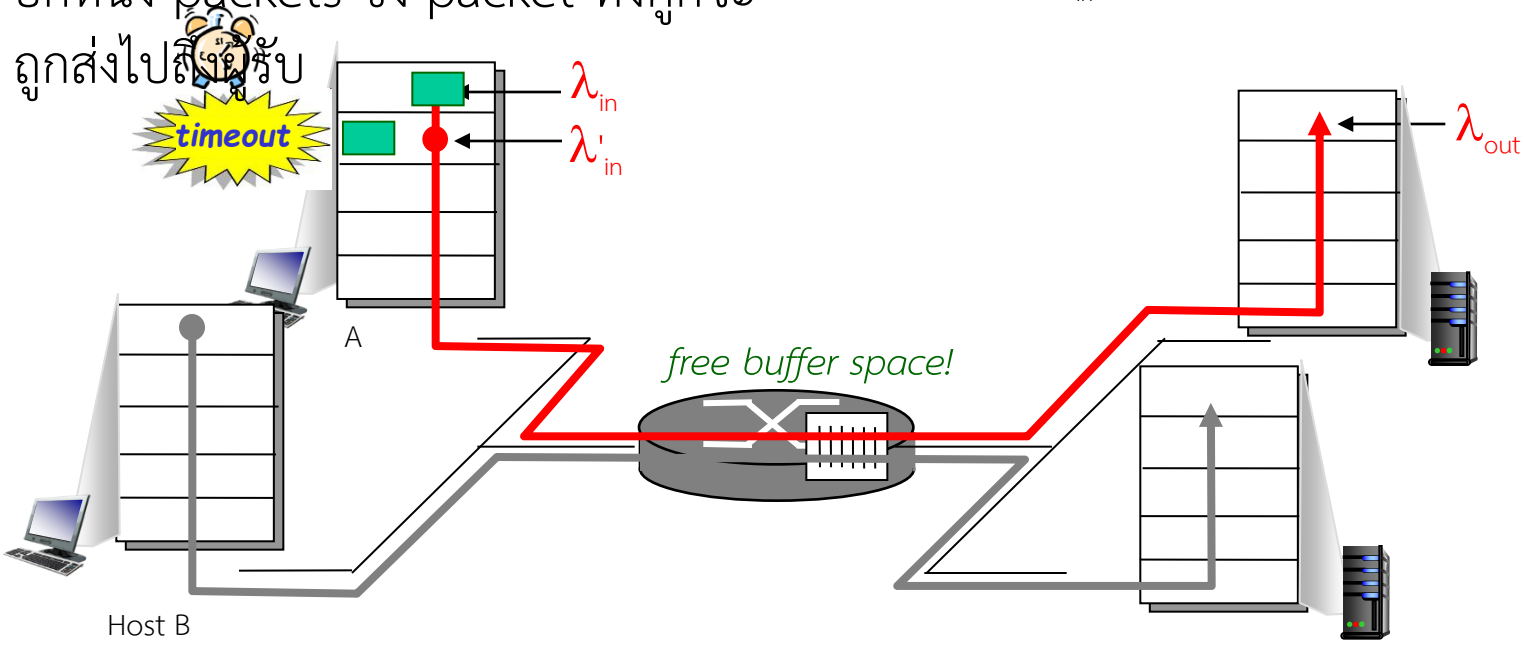
ต้นเหตุและผลของความคับคั่ง: เหตุการณ์ 2

ในความเป็นจริง: มีการส่งซ้ำ

- packets หายหรือตกหล่นระหว่างเส้นทางเนื่องจาก buffers เต็ม
- แต่บางครั้ง Timer ของผู้ส่งหมดเวลาลงก่อนเวลาอันควร, ผู้ส่งจึงส่งข้อมูลซ้ำไปอีกหนึ่ง packets ซึ่ง packet ทั้งคู่ก็จะถูกส่งไปถึงผู้รับ



เมื่อกำลังส่งที่อัตรา R/2, บาง packet จะถูกส่งซ้ำ ในที่นี้รวมการส่งข้อมูลซ้ำ ที่ข้อมูลไม่ได้หาย แต่ timeout ก่อนเวลาด้วย !

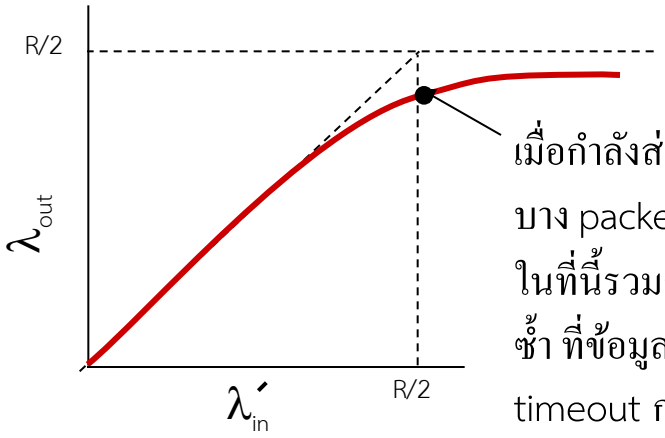


ต้นเหตุและผลของความคับคั่ง: เหตุการณ์ 2

ในความเป็นจริง: มีการส่งซ้ำ

1. packets หายหรือตกหล่นระหว่างเส้นทาง เนื่องจาก buffers เต็ม

2. แต่บางครั้ง Timer ของผู้ส่งหมดเวลาลงก่อนเวลาอันควร, ผู้ส่งจึงส่งข้อมูลซ้ำไปอีกหนึ่ง packets ซึ่ง packet ทั้งคู่ก็จะถูกส่งไปถึงผู้รับ



"ผล" ของความคับคั่ง:

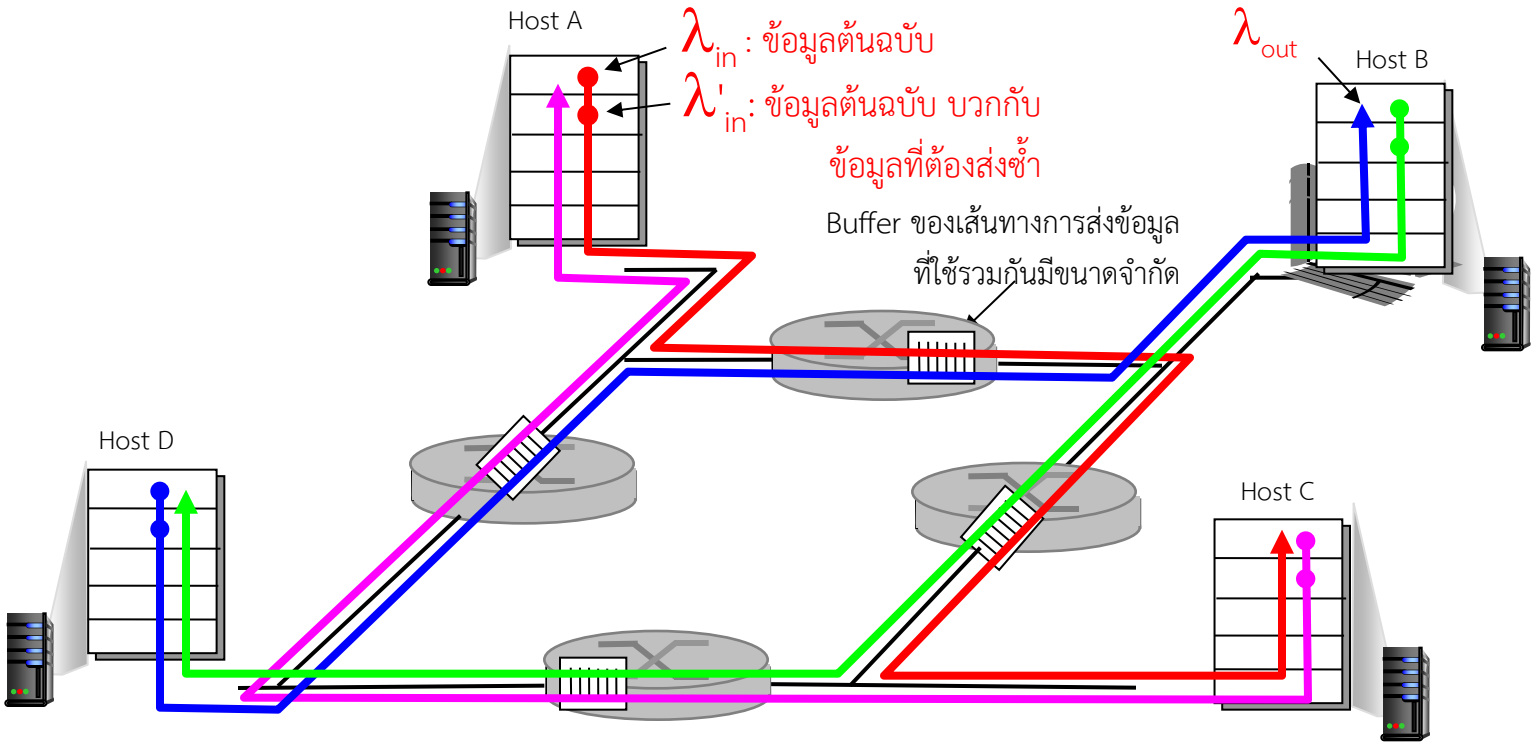
- เพื่อให้ได้ข้อมูลในอัตราที่ต้องการ (goodput) ต้องทำงานมากขึ้น (ส่งซ้ำ)
- เมื่อต้องส่ง packets ซ้ำโดยที่ไม่จำเป็น : link จะต้องลำเลียงหลาย copy ของข้อมูลเดียวกัน
  - ทำให้ goodput ลดลง

ต้นเหตุและผลของความคับคั่ง: เหตุการณ์จำลองที่ 3

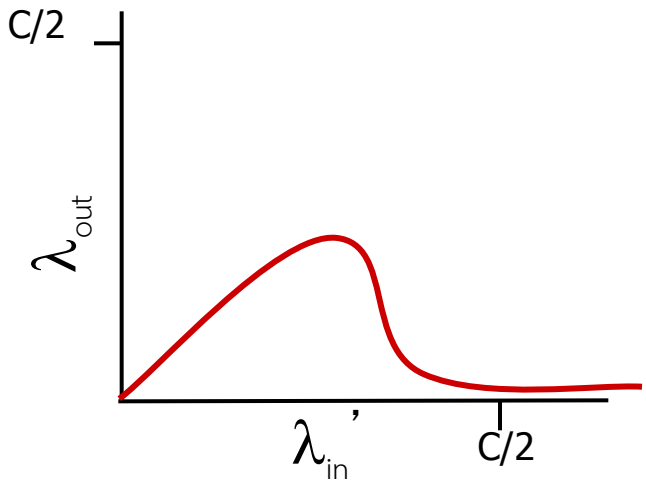
- มีผู้ส่งข้อมูลทั้ง 4 โฮสต์
- เส้นทางต้องผ่านหลายจุด
- หมดเวลา/ส่งไปข้อมูลซ้ำ

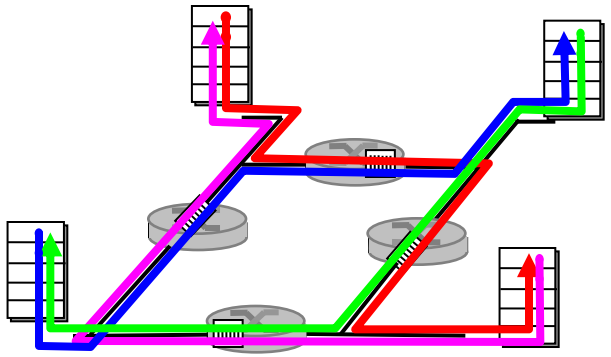
Q: เกิดอะไรขึ้นเมื่อ λ_{in} และ λ_{in}' เพิ่มขึ้น?

A: เมื่อ λ_{in}' ของข้อมูลเส้นสีแดงมีขนาดเพิ่มขึ้น, ข้อมูลในเส้นสีน้ำเงินก็จะถูก drop ทำให้ throughput ของ connection สีน้ำเงินเหลือ 0



ต้นเหตุและผลของความคับคั่ง: เหตุการณ์จำลองที่ 3





อีกหนึ่ง “ผลเสีย” ของความคับคั่ง :

เมื่อแพกเกจถูกทิ้งไป นั่นก็เท่ากับว่า ทรัยากรเครือข่าย (ที่ใช้ส่งข้อมูลก่อนหน้าที่ข้อมูลนี้จะถูกทิ้ง) ถูกใช้ไปอย่างไร้ค่า

แนวทางในการควบคุมความคับคั่งของข้อมูล

2 แนวทางในการควบคุมความคับคั่ง:

การควบคุมความคับคั่งโดยใช้แค่เครื่องปลายทาง (end-end) :

- ไม่มีการแจ้งกลับ (feedback) จากเครือข่าย
- ความคับคั่งของเครือข่ายจะถูกเดาโดยเครื่องที่อยู่ปลายทางที่คอยสังเกต loss, delay
- เป็นวิธีของ TCP

การควบคุมความคับคั่งโดยให้อุปกรณ์เครือข่ายช่วย:

- routers จะทำการแจ้ง feedback ไปที่ระบบปลายทาง
  - ใช้ bit 1 bit เพื่อระบุถึงความแออัด (SNA, DECbit, TCP/IP ECN, ATM)
  - หรือ ระบุอัตราการส่งที่ชัดเจนสำหรับผู้ส่ง

กรณีศึกษา : การควบคุมความคับคั่งของ ATM แบบ ABR

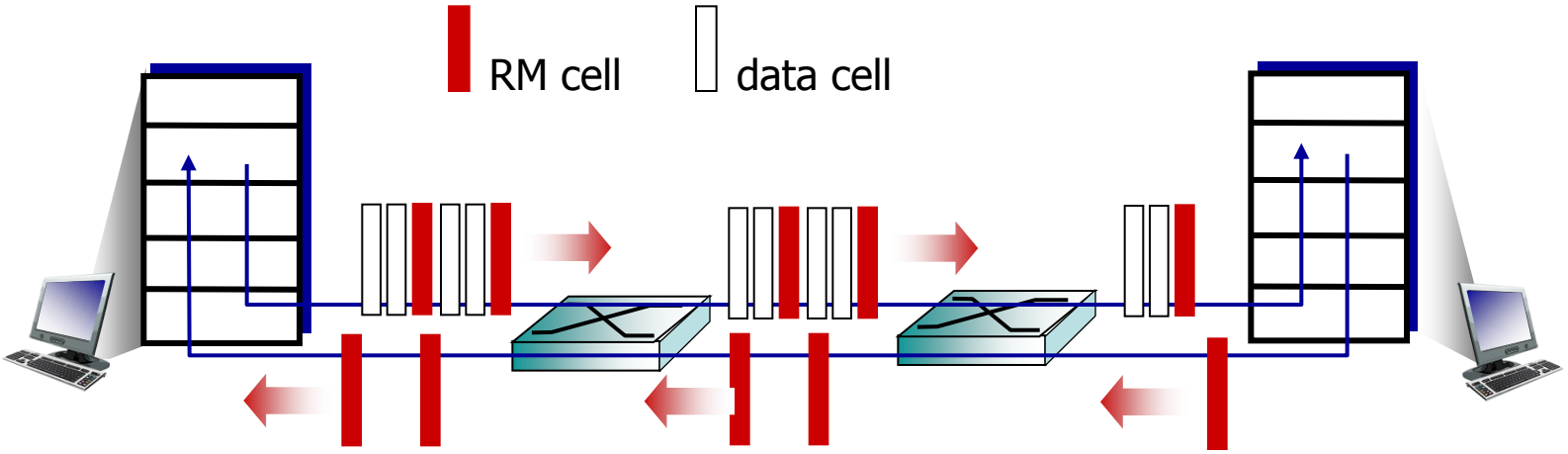
ABR: available bit rate:

- “บริการที่มีความยืดหยุ่น”
- ถ้าเส้นทางของผู้ส่ง “มีข้อมูลน้อยกว่าความจะมาก”:
  - ผู้ส่งจะใช้ประโยชน์จาก bandwidth ที่เหลืออยู่
- ถ้า path ของผู้ส่งแออัด:
  - ผู้ส่งจะค่อย ๆ ลดอัตราส่งให้เหลือเท่าอัตราการส่งขั้นต่ำที่รับประกันไว้

RM (resource management) cells:

- ส่งโดยผู้ส่ง, แทรก ๆ ไปกับ cell ที่เป็นข้อมูล
- bits ใน RM cell ถูกตั้งค่าโดย switches (“เครือข่ายคอยช่วย”)
  - *NI bit:* ไม่เพิ่มอัตราการส่ง (ความแออัดระดับกลาง)
  - *CI bit:* การบ่งบอกถึงความแออัด
- ผู้รับไม่แก้ค่าใน bit นี้ของ RM cells และส่งไปให้ผู้ส่ง

กรณีศึกษา : การควบคุมความคับคั่งของ ATM แบบ ABR



- field ข้อมูล ER (explicit rate) ขนาด 2 byte ใน cell RM
  - switch ที่แออัดอาจลดค่า ER ใน cell
  - ดังนั้น อัตราที่ผู้ส่งจะส่งข้อมูลจะเป็นอัตราการส่งมากที่สุดที่ link ในเส้นทางจะรองรับได้
- bit EFCI ใน cell ข้อมูล: ถูกตั้งค่าไว้ที่ 1 ใน switch ที่มีข้อมูลแออัด
  - ถ้า cell ข้อมูลก่อนหน้า RM cell ถูกตั้งค่า EFCI, ผู้รับจะตั้งค่า bit CI ใน cell RM ที่ส่งกลับให้ผู้ส่ง

Chapter 3 Outline

3.1 บริการในขั้น Transport

3.2 การรวมและการแยกข้อมูล (multiplexing and demultiplexing)

3.3 การส่งข้อมูลที่ไม่ต้องมีการเชื่อมต่อ (connectionless transport): UDP

3.4 หลักการต่าง ๆ ของการส่งข้อมูลที่ไว้ใจได้ (principles of reliable data transfer)

3.5 การส่งข้อมูลที่ต้องมีการเชื่อมต่อก่อน (connection-oriented transport): TCP

- โครงสร้างส่วนข้อมูล (segment)
- การส่งข้อมูลที่น่าเชื่อถือ/น่าไว้ใจได้ (reliable data transfer)
- การควบคุมการไหล (flow control)
- การจัดการการเชื่อมต่อ (connection management)

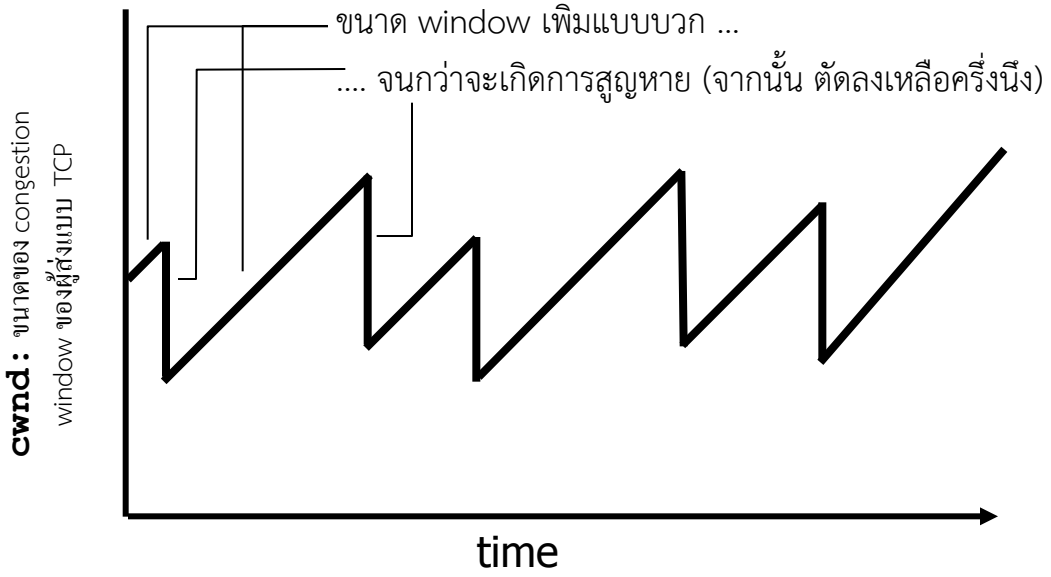
3.6 หลักการของการควบคุมความคับคั่ง (congestion control)

3.7 congestion control ของ TCP

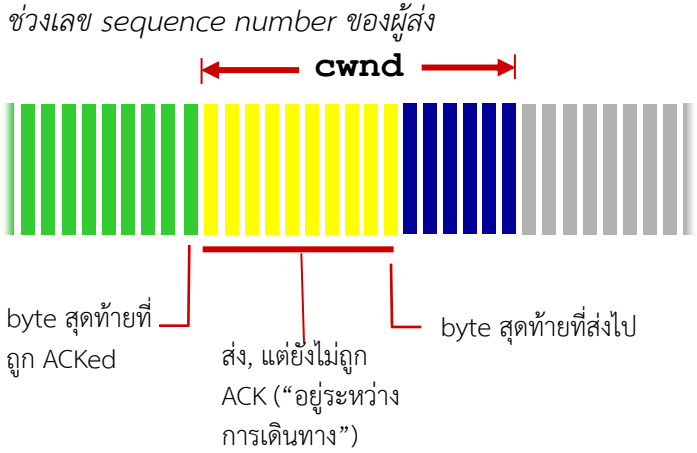
การควบคุมความคับคั่งของ TCP : additive increase (เพิ่มแบบบวก) multiplicative decrease (ลดแบบเท่าตัว)

- *วิธี:* ผู้ส่งเพิ่มอัตราส่ง (window size), ควานหา bandwidth ที่มีให้ใช้ได้, จนกว่าการสูญหายจะเกิดขึ้น
  - *additive increase:* เพิ่ม **cwnd** ทีละ 1 MSS ทุก ๆ RTT จนกว่าจะ pkt จะหาย
  - *multiplicative decrease*: ตัดลด **cwnd** เหลือครึ่งเดียวหลังจากการสูญหาย

พฤติกรรมของ AIMD เป็นลักษณะฟันเลื่อย: หา bandwidth



การควบคุมความคับคั่งของ TCP : รายละเอียด



- ผู้ส่งจำกัดการส่ง:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}$$

- ขนาด **cwnd** จะเปลี่ยนแปลงไปมา ขึ้นกับความคับคั่งของเครือข่ายที่ผู้ส่งได้รู้มา

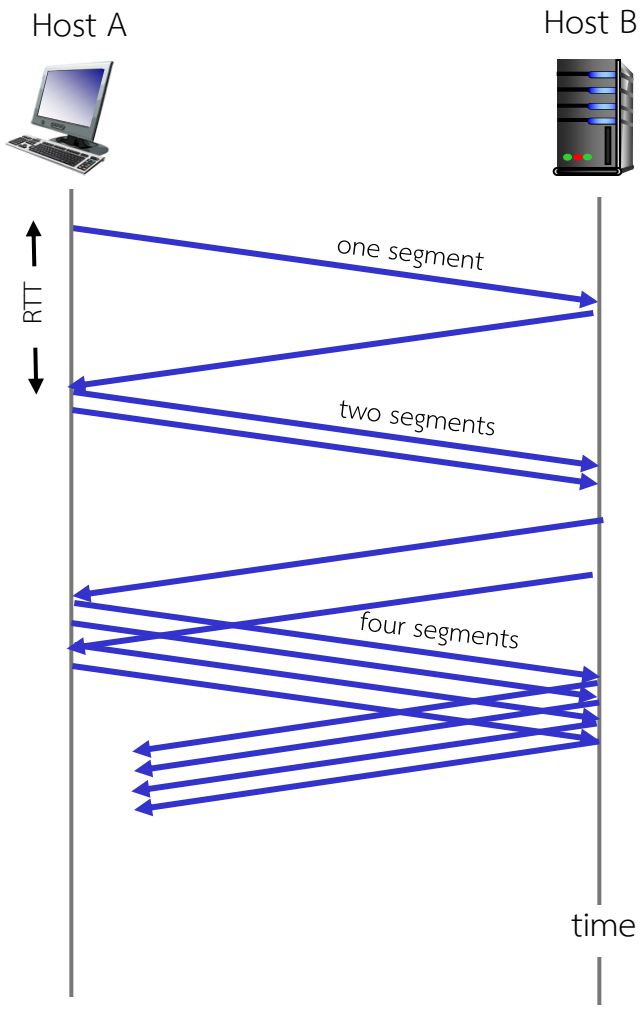
อัตราการส่ง TCP:

- *อย่างคร่าว ๆ:* ส่งข้อมูลจำนวน cwnd bytes, รอ ACKS เพื่อประมาณค่า RTT, จากนั้นส่งไบต์ข้อมูลเพิ่ม

$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

TCP Slow Start

- เมื่อเริ่มการเชื่อมต่อ, เพิ่มอัตราการส่งแบบยกกำลัง (exponentially) จนกว่าจะเจอกับ loss ครั้งแรก:
  - เริ่มต้นขนาด **cwnd** = 1 MSS
  - เพิ่มขนาด **cwnd** หนึ่งเท่าตัวทุก ๆ RTT
  - นั่นคือ โดยเพิ่มขนาด **cwnd** ไป 1 MSS สำหรับทุก ๆ ACK ที่ได้รับ
- *สรุป:* อัตราส่งเริ่มแรกจะช้าแต่ค่าจะเพิ่มแบบกระโดดขึ้นอย่างรวดเร็ว



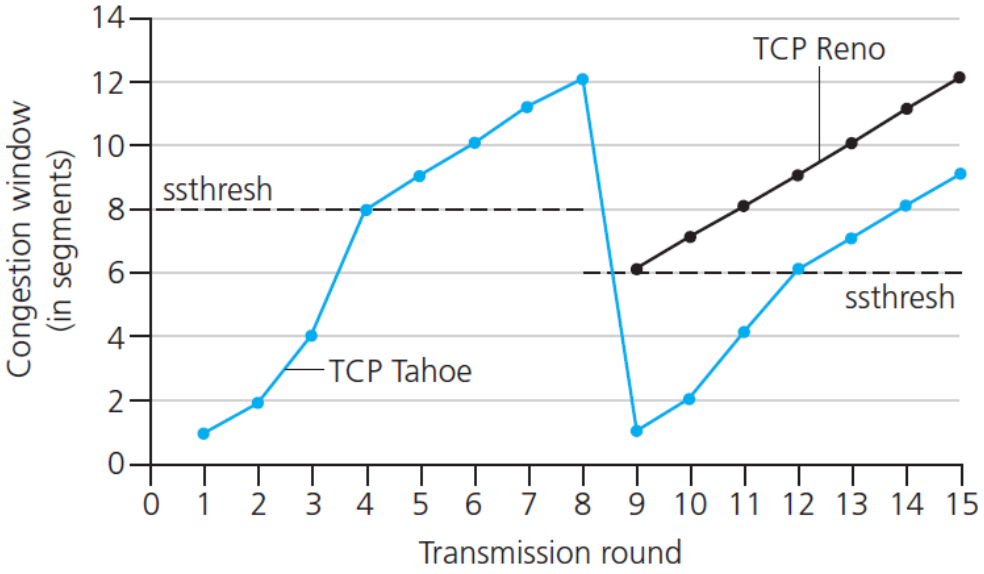
TCP: การหา loss และการตอบสนอง

- ถ้า loss ถูกระบุโดย timeout:
  - **cwnd** ถูกตั้งไว้ที่ 1 MSS;
  - จากนั้นหน้าต่างจะเพิ่มขึ้นเป็นเลขยกกำลัง (exponential เหมือนใน slow start) จนไปถึงค่า threshold, จากนั้นจะเพิ่มขึ้นเป็นเชิงเส้น
- ถ้า loss ถูกระบุโดยได้รับ ACK ซ้ำ 3 ACKs (3 duplicate acks):
  - TCP RENO
    - ACKs ซ้ำ ระบุถึงความสามารถการส่ง segments ของเครือข่าย
    - **cwnd** ถูกตัดเหลือครึ่งนึงจากนั้นเพิ่มขึ้นแบบเชิงเส้น
  - TCP Tahoe จะตั้งค่า **cwnd** เป็น 1 เสมอ (ไม่ว่าจะ timeout หรือ 3 dup acks)

TCP: จาก slow start เป็น CA (Congestion Avoidance)

Q: เมื่อไร TCP ควรจะเปลี่ยนจากการเพิ่มขึ้นแบบ exponential เป็นแบบเชิงเส้น ?

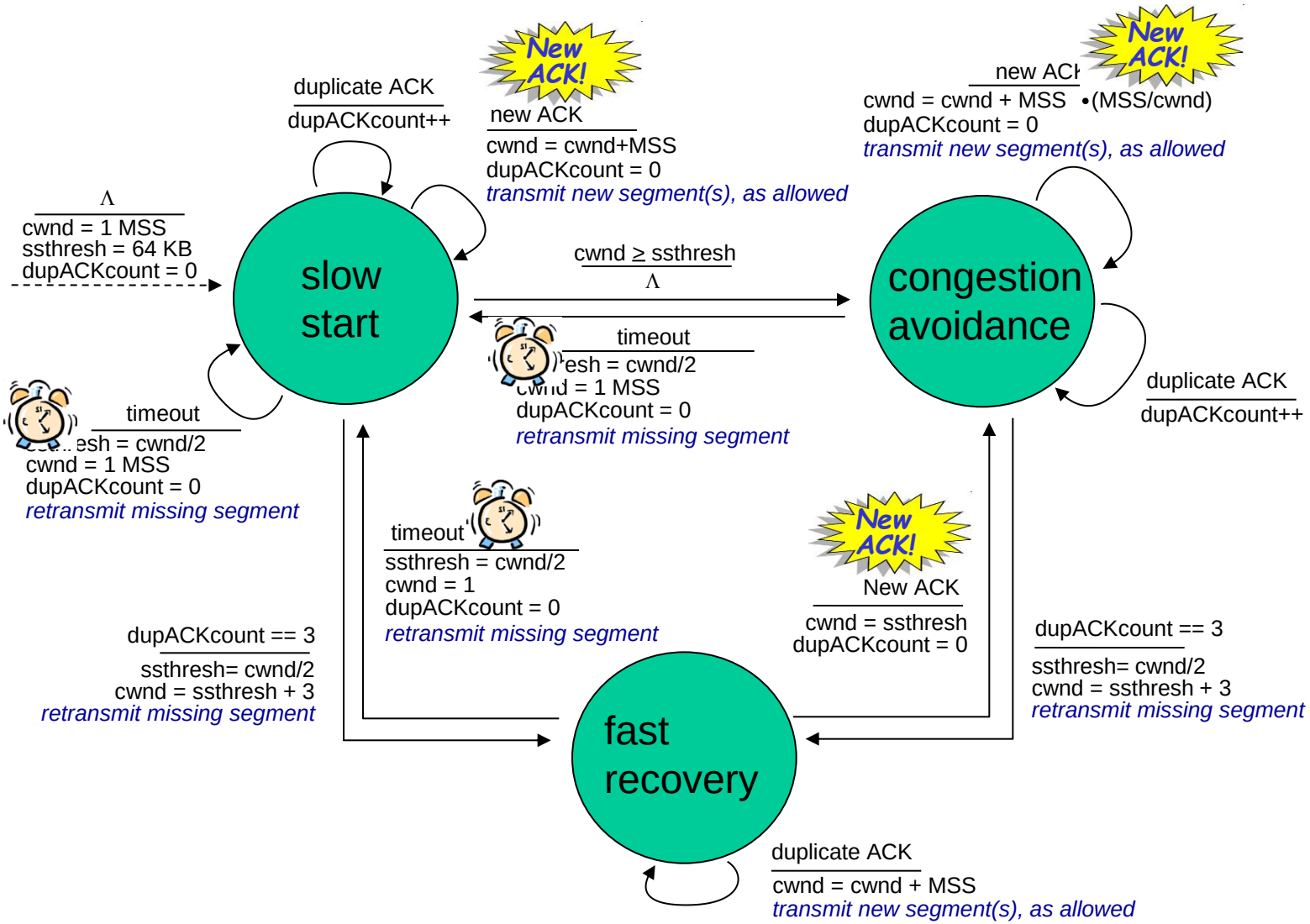
A: เมื่อ cwnd ถูกเพิ่มเป็น 1/2 ของ “ค่าที่มากที่สุด” ของมันก่อนจะเกิด timeout



การดำเนินการ:

- ตัวแปร **ssthresh**
- ขณะเกิด Loss , **ssthresh** จะถูกตั้งค่าที่ 1/2 ของค่า **cwnd** ก่อนเกิดเหตุแป๊ปเดียว (ค่า cwnd ที่มากที่สุดก่อนเกิดเหตุการณ์)

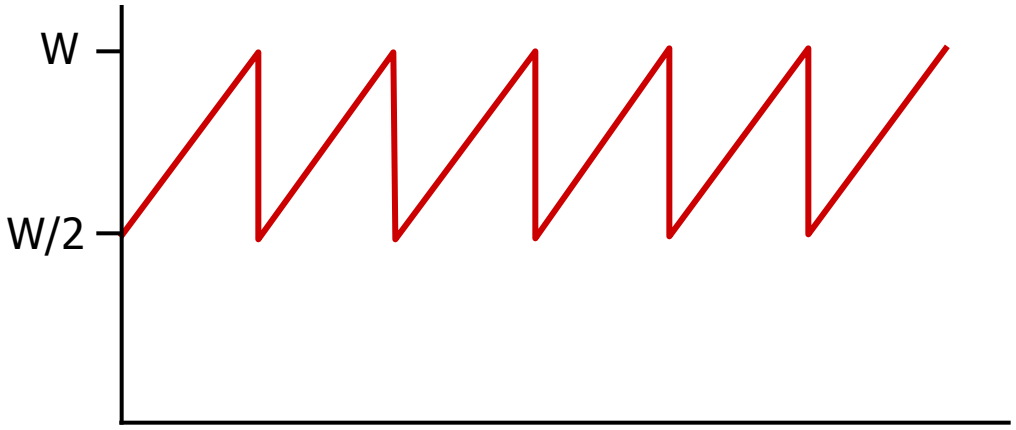
สรุป: TCP Congestion Control



Throughput ของ TCP

- ค่าเฉลี่ย throughput ของ TCP เป็น function ของขนาดของ window และ RTT?
  - ไม่สนใจ slow start, สมมติว่าข้อมูลถูกส่งตลอด
- W: ขนาดของ window (หน่วยเป็น bytes) การสูญเสียที่เกิดขึ้น
  - ค่าเฉลี่ยขนาดของ window (จำนวน byte ที่เดินทางอยู่) คือ ¾ W
  - ค่าเฉลี่ยของ throughput คือ 3/4W ต่อ RTT

$$\text{avg TCP thruput} = \frac{3}{4}\frac{W}{RTT} \text{ bytes/sec}$$



TCP ในอนาคต: TCP ใน “pipes ที่ยาวและมีความจุสูง”

- ตัวอย่าง : 1500 byte segments, 100ms RTT, ต้องการช่องทางการส่งขนาด 10 Gbps
- requires W = 83,333 in-flight segments
- ความเป็นไปได้ของการสูญหายของในช่องทางการส่งข้อมูล, L [Mathis 1997]:

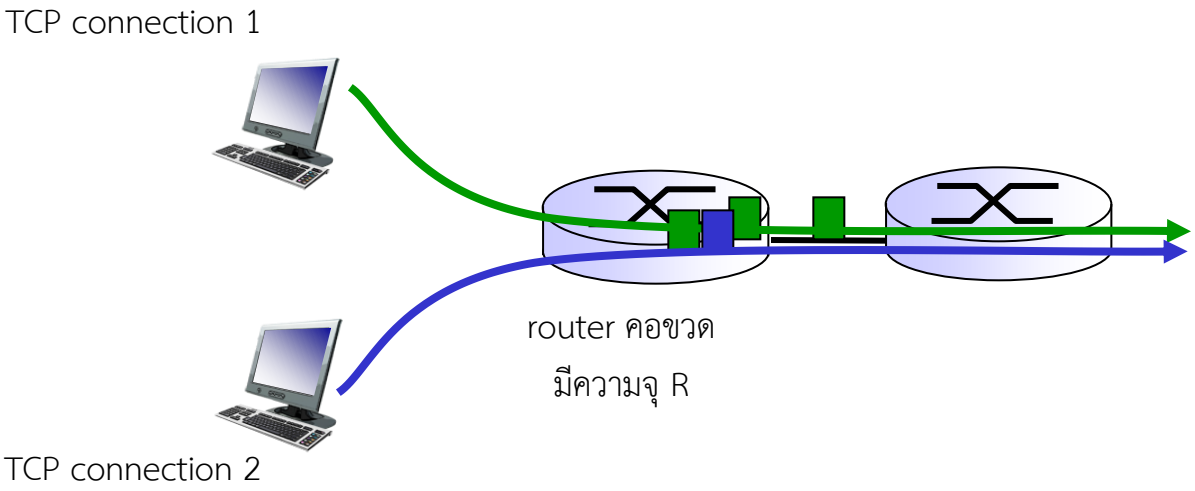
$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT}\sqrt{L}}$$

➜ เพื่อที่จะมีช่องทางการส่งขนาด 10Gbps, ต้องมีการสูญหายของข้อมูลเพียงแค่ = $2 \cdot 10^{-10}$ *– น้อยมาก !!*

- ดังนั้น ต้องมี TCP รุ่นใหม่สำหรับการส่งแบบความเร็วสูง

ความเสมอภาค (Fairness) ของ TCP

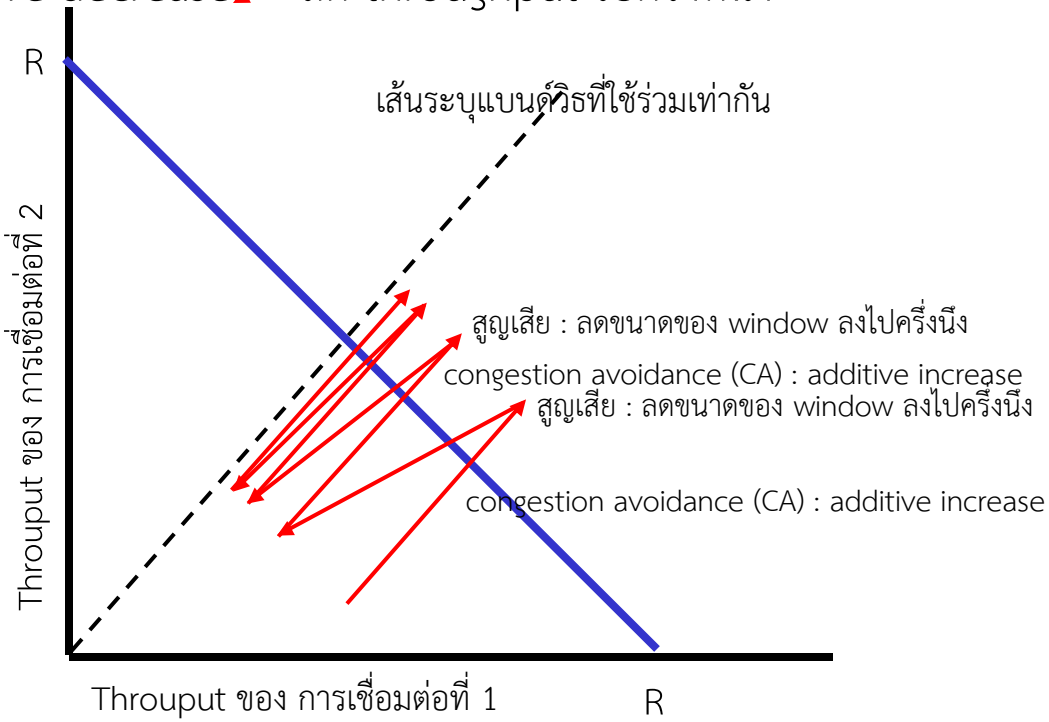
- เป้าหมายของ *fairness:* ถ้ามี TCP session จำนวน K session ใช้ทรัพยากร bandwidth R ร่วมกัน, แต่ละ session ควรจะเฉลี่ยทรัพยากรเหมือนกันที่ R/K



ความเสมอภาคของ TCP คืออะไร?

session 2 session แข่งกันใช้ทรัพยากร :

- additive increase ↗ ให้ความชันเส้นเท่ากับ 1 (session ทั้ง 2 ค่อย ๆ ได้ throughput เพิ่มทีละนิด)
- multiplicative decrease ↙ ลด throughput ไปครึ่งหนึ่ง



ความเสมอภาค (ต่อ)

ความเสมอภาค และ UDP

- มัลติมีเดียแอพพลิเคชั่นบ่อยครั้งไม่ใช้ TCP
  - ไม่ต้องการให้อัตราการส่งลดลงเพราะการควบคุมความคับคั่ง
- มัลติมีเดียแอพพลิเคชั่นจึงใช้ UDP แทน:
  - ส่งเสียง/วีดีโอด้วยความเร็วคงที่, ทนต่อการสูญเสียข้อมูล

ความเสมอภาค , การเชื่อมต่อ TCP แบบขนาน

- แอพพลิเคชั่นสามารถที่จะเปิดหลายการเชื่อมต่อระหว่างเครื่อง 2 เครื่องแบบขนานได้
- เว็บบราวเซอร์ทำแบบนี้
- ตัวอย่างเช่น link ที่มีความจุ R ซึ่งมี 9 การเชื่อมต่อ:
  - ถ้าแอพใหม่ต้องการ 1 TCP, จะได้รับที่ rate R/10
  - ถ้าแอพใหม่ต้องการ 11 TCPs, จะได้รับที่ R/2

Chapter 3: บทสรุป

- หลักการเบื้องหลังการให้บริการชั้น transport :
  - multiplexing, demultiplexing
  - การส่งข้อมูลแบบไว้ใจได้
  - การควบคุมการไหลของข้อมูล
  - การควบคุมความคับคั่งของข้อมูล
- กรณีศึกษาและการปฏิบัติบน internet
  - UDP
  - TCP

ถัดไป:

- ออกจากปลายทางของระบบเครือข่าย (application, transport layers)
- เข้าไปยังสู่แกนของเครือข่าย