3장 Transport Layer

3장 Transport Layer

이 장의 목적

- Transport layer의 배경과 원리 이해
  - multiplexing/demultiplexing
  - 신뢰적인 data transfer
  - flow control
  - congestion control
- 인터넷 transport layer 이해
  - UDP : connectionless transport
  - TCP : connection oriented transport
  - TCP congestion control

3장 Transport Layer

3.1 Transport Layer 서비스 개요

3.2 다중화와 역다중화

3.3 Connectionless transport : UDP

3.4 신뢰적 data 전송의 원리

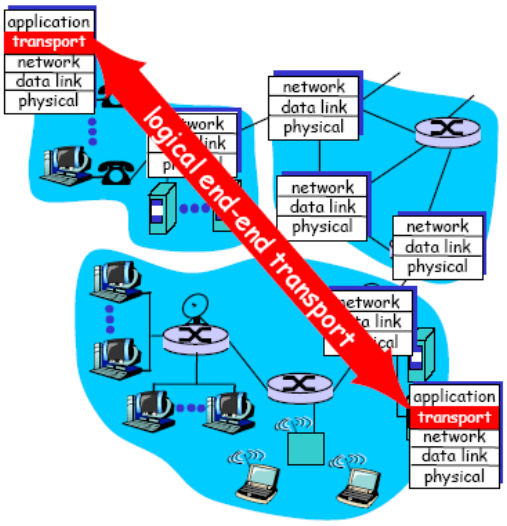
3.5 Connection oriented transport : TCP
- segment structure
- reliable data transfer
- flow control
- connection control

3.6 혼잡제어의 원리

3.7 TCP의 혼잡제어

Transport services and protocol

- 서로 다른 host간에 작동하는 app process간에 논리적인 통신 (logical communication)을 제공한다.
- transport protocol은 end system에서 작동한다.
  - 송신측 : app message를 segment로 변환 network layer에 전달한다.
  - 수신측 : network layer로 부터 전달받은 segment로 부터 message를 추출 app layer에 전달한다.
- 네트워크 app는 하나이상의 transport protocol을 사용 할 수 있다
  - 인터넷 : TCP, UDP

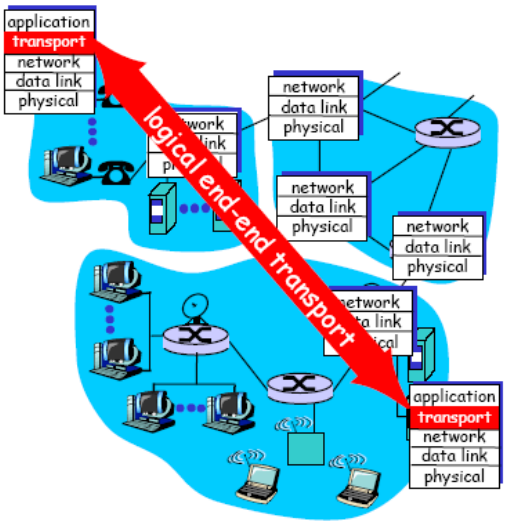


Transport vs Network layer

- network layer : host 간에 logical communication을 제공
- transport layer : process간에 logical communication을 제공
  - Network layer가 제공하지 못하는 신뢰적인 전송을 제공 할 수 있다.
  - 그러나 network layer의 제약을 받는 경우도 있다.
    : 지연이나 대역폭에 대한 보장은 불가능하다.

인터넷에서 trans-layer의 개요

- 신뢰적인 연결지향 서비스(TCP)
  - 혼잡제어
  - 흐름제어
  - 연결지향
- 비신뢰적인 비연결지향 서비스 (UDP)
  - best-effort delivery service : IP
    : data 전달에 최선을 다하지만 어떠한 보장도 하지 않는다.
- 보장하지 않는것
  - 최대 지연시간
  - 전달 대역폭



3장 Transport Layer

3.1 Transport Layer 서비스 개요

3.2 다중화와 역다중화

3.3 Connectionless transport : UDP

3.4 신뢰적 data 전송의 원리

3.5 Connection oriented transport : TCP
- segment structure
- reliable data transfer
- flow control
- connection control

3.6 혼잡제어의 원리

3.7 TCP의 혼잡제어

Multiplexing/demultiplexing

Demultiplexing at rcv host

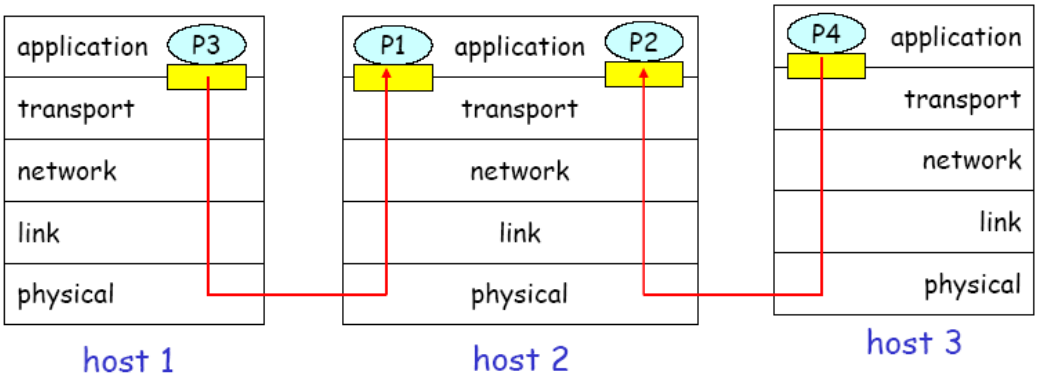
segment의 data를 올바른 socket으로 전달하는 작업

Multiplexing at send host

socket들로 부터 data를 모으고 각 data의 header정보로 캡슐화 하고 하위 layer로 전달하는 작업

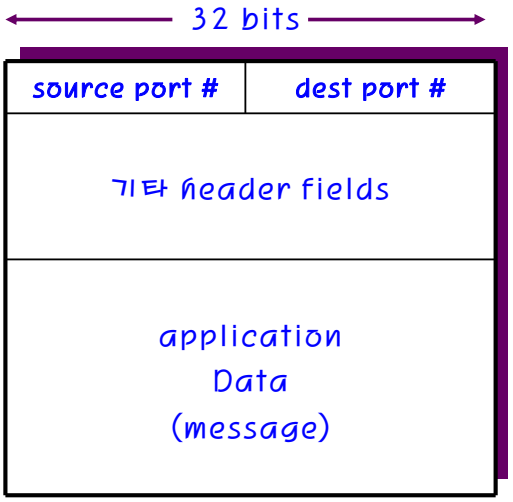
= process

= socket



Transport layer에서 demultiplexing

- Transport layer에서 demultiplexing을 위한 요구 사항
  - 각 socket은 유일한 식별자를 갖는다.
  - 각 segement는 segment가 전달될 적절한 socket을 가르키는 특별한 field를 갖는다.
  - 이들 특별한 field는 source와 destination의 port 번호이다. (0~1023 까지의 port를 well-know port라고 한다.: RFC 1700에 명시되어있다.)
- Segment가 host에 도착하면 T/L는 segment안에 목적지의 port #를 검사하고 이에 상응하는 socket으로 data를 전달한다.



UDP/TCP segment format

port# : 0 ~ 65534

Connectionless demultiplexing

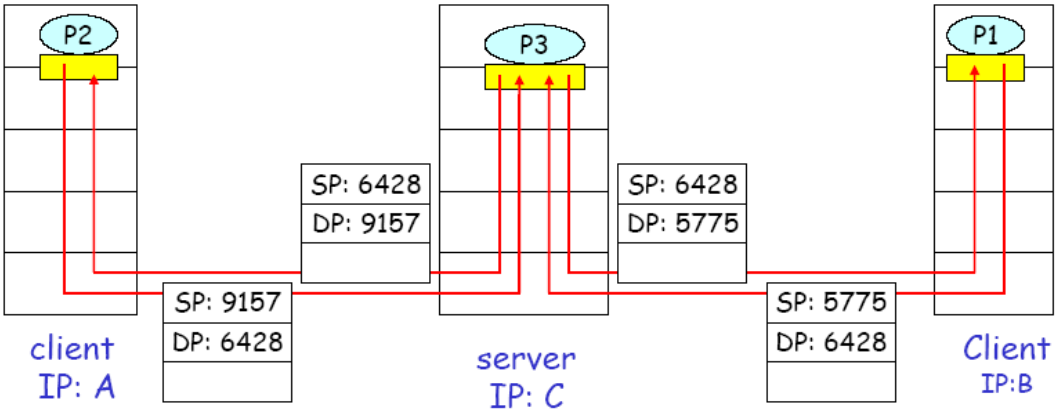
- Port #를 이용 socket을 생성한다.

```
DatagramSocket mysocket1=new
DatagramSocket(59111);
DatagramSocket mysocket1=new
DatagramSocket(59222);
```

- UDP socket은 목적지 IP와 목적지 port #로 구성된 두 요소로 된 집합에 의해 식별된다.
- UDP segment 전달
  - Segment의 dest port #를 검사한다.
  - Segment를 적절한 socket으로 전달한다.
- 두개의 UDP segment들이 출발지 IP나 port # 가 둘다 다르거나 둘 중 하나가 다르더라도 동일한 목적지 IP와 port #를 갖는다면 동일한 socket을 통해 process에 전달된다.

Connectionless demultiplexing

DatagramSocket serverSocket = new DatagramSocket(6428);

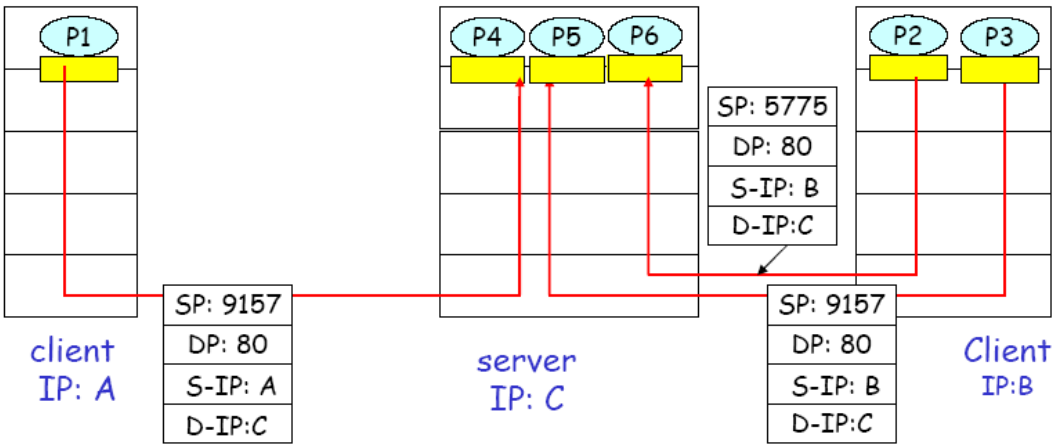


SP provides "return address"

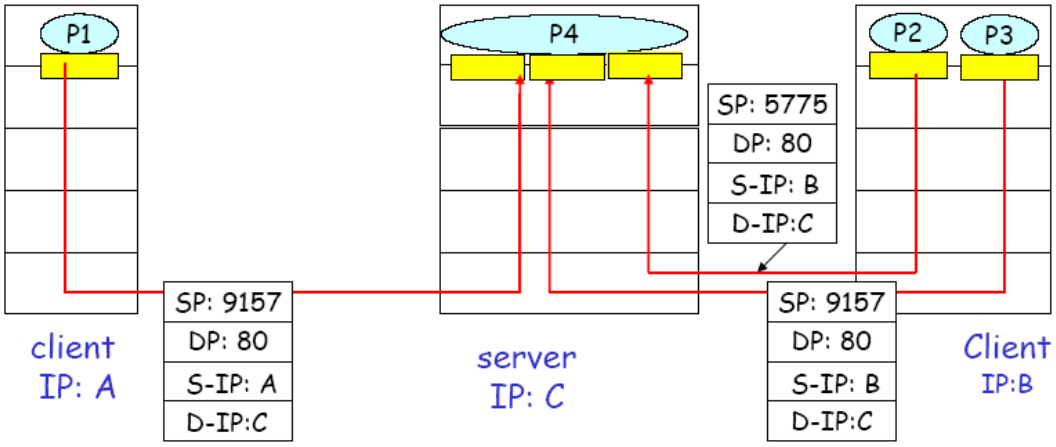
Connection oriented demux

- TCP socket는 4개의 tuple로 구성된 집합에 의해 식별된다.
  - source IP
  - source port #
  - dest IP
  - dest port #
- Host에 segment가 도착하면 host는 적절한 socket으로 segment를 전달하기 위해 네개의 값을 모두 사용한다.
- Server host는 동시에 여러 개의 TCP socket을 지원한다.
  - 각 socket들은 앞에본 4개의 tuple에 의해 구별된다.
- Web server는 각각의 client의 접속에 대해 서로 다른 socket을 이용한다.
  - non-persistent HTTP의 경우 모든 요청마다 다른 socket을 이용한다.
    각각의 object마다 새로운 TCP connection을 이용한다.

Connection oriented demux



Connect oriented demux
Thread Web server



3장 Transport Layer

3.1 Transport Layer 서비스 개요

3.2 다중화와 역다중화

3.3 Connectionless transport : UDP

3.4 신뢰적 data 전송의 원리

3.5 Connection oriented transport : TCP
- segment structure
- reliable data transfer
- flow control
- connection control

3.6 혼잡제어의 원리

3.7 TCP의 혼잡제어

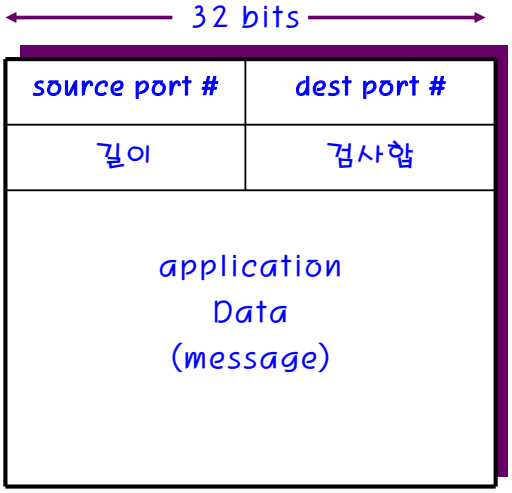
UDP : User Datagram Protocol [RFC 768]

- UDP는 Transport layer에서 할 수 있는 최소한의 기능으로 동작한다.
  - App가 거의 IP와 직접 통신하는 것이다.
  - mulx/demulx 기능만을 제공
- "best effort" (보장하지 않는다.)
  - 손실가능
  - 전송순서가 바뀔 가능
- connectionless
  - UDP sender와 receiver간에 handshaking하지 않는다.
  - 각 UDP segment들은 서로 독립적으로 전달된다.

UDP를 이용하는 이유
- 연결설정이 없다.
  예비동작 없이 전송함으로 설정에 따른 어떠한 지연도 없다 (DNS)
- 연결상태가 없다.
  연결상태 유지를 위한 변수가 없음으로 TCP보다 많은 app를 수용할 수 있다.
- header가 작다.
  8 byte (TCP : 20 byte)
- 혼잡제어하지 않는다.
  네트워크의 혼잡도를 고려하지 않음으로 app가 요구하는 전송을 전송량을 제한없이 전송한다. (SNMP)

UDP

- often used for streaming multimedia apps
  - 손실 허용
  - Rate sensitive
- UDP를 사용하는 app
  - DNS
  - SNMP
- app가 신뢰성을 제공한다면 UDP 상에서도 신뢰성있는 통신이 가능하다.
  - app이 신뢰성있는 통신을 위해 확인응답이나 재전송등의 기능을 제공해야한다.



UDP/TCP segment format

UDP checksum

목적 : 전송된 segment에 "errors"를 검출한다.

Sender

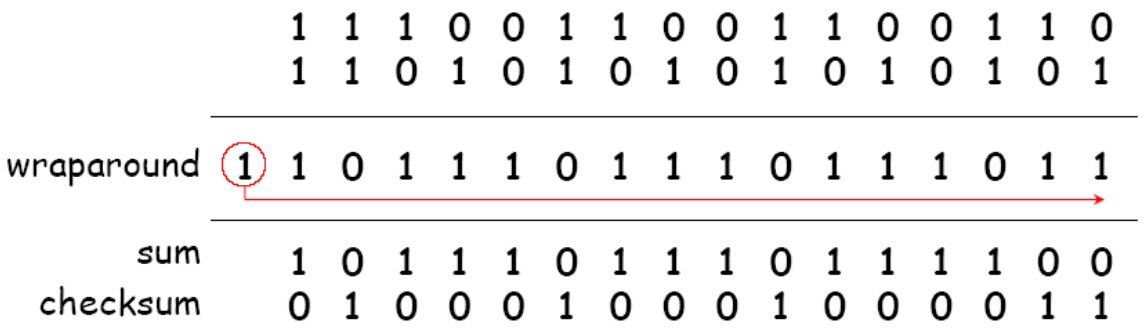
- Segment의 content는 순서에 따라 16bit integers로 간주한다.
- 모든 16bit(word)의 합을 가지고 1의 보수를 수행한다. (이때 오버플로우는 버린다.)
- 결과는 UDP segment checksum field에 삽입한다.
- overflow bit를 wraparound로 이용 하는 경우도 있음

receiver

- Segment를 수신후 checksum을 을 포함한 모든 16bit word를 더한다.
- 값이 모두 1이면 에러가 없지만 하나라도 0이 나오면 오류가 있다.
- 참고 (checksum을 사용하는 방법과 bit sum에서 발생하는 overflow를 처리하는 방법이 다름. 과정은 다르지만 판단 결과는 동일)

Internet checksum

◈ Example : 두개의 16bit integers



3장 Transport Layer

3.1 Transport Layer 서비스 개요

3.2 다중화와 역다중화

3.3 Connectionless transport : UDP

3.4 신뢰적 data 전송의 원리

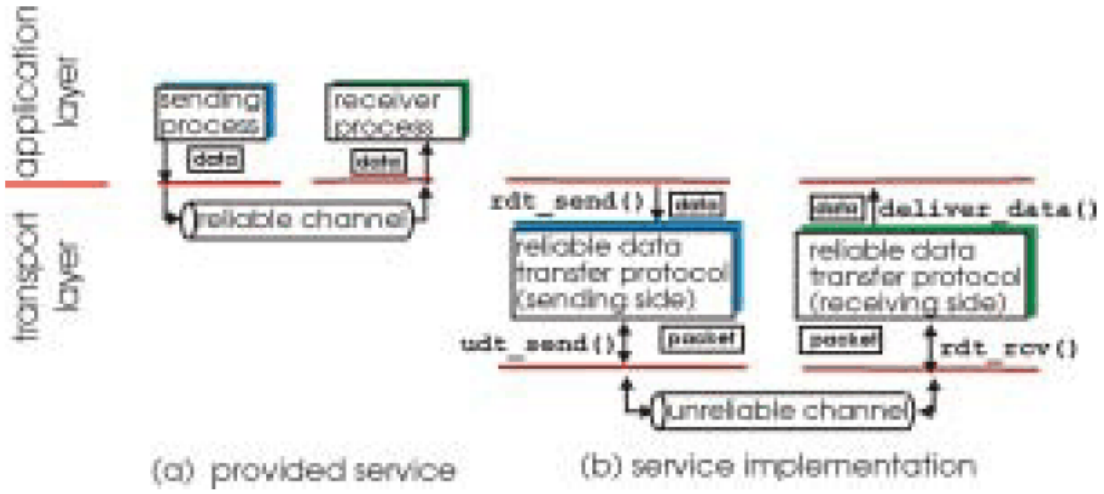
3.5 Connection oriented transport : TCP
- ❍ segment structure
- ❍ reliable data transfer
- ❍ flow control
- ❍ connection control

3.6 혼잡제어의 원리

3.7 TCP의 혼잡제어

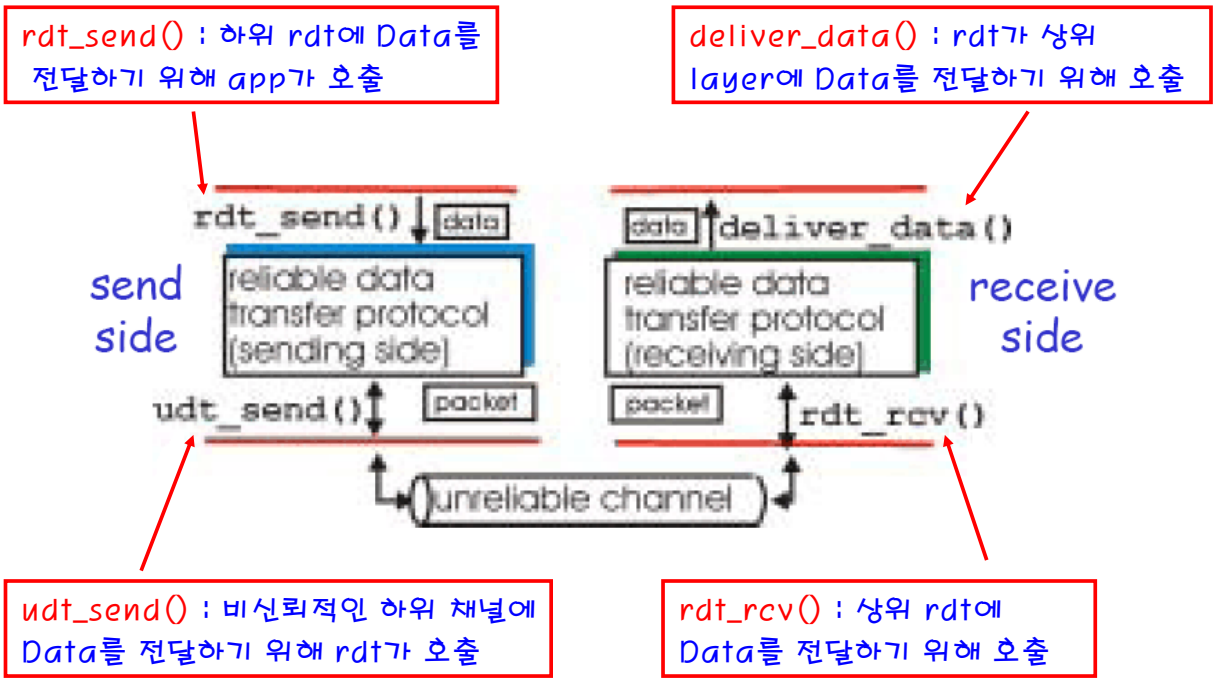
신뢰적인 data 전송의 원리

- T/L 뿐아니라 A/L, L/L에서도 매우 중요한 문제이다.
- 매우 중요한 topics



- 하위계층이 비 신뢰적인 채널인 경우 상위채널에서 신뢰적인 통신을 제공 해야 한다.

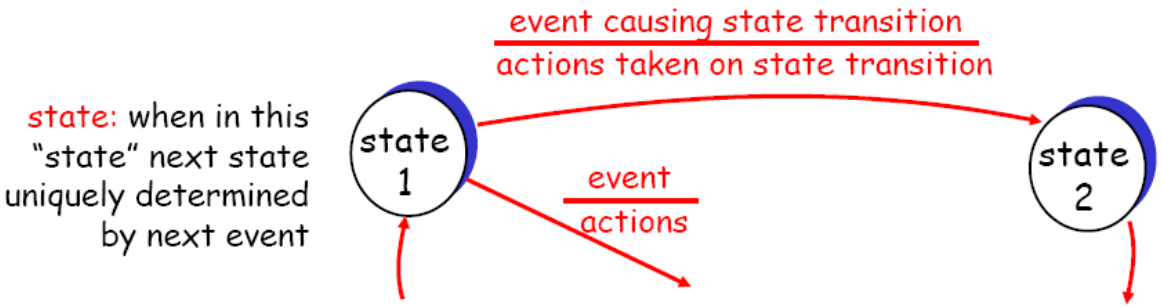
신뢰적 Data transfer



신뢰적 Data transfer

앞으로의 전개

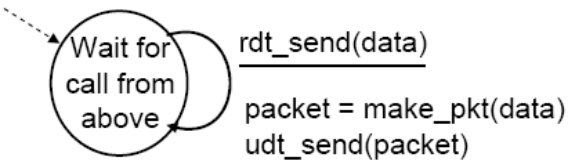
- 점점 복잡해지는 하위채널을 고려 신뢰적 data 전송 protocol의 송신자 측면과 수신자 측면을 전개
- 단 방향 전송만을 고려 : 이 경우에도 제어 패킷은 양방향 전송이 필요
- FSM으로 표현



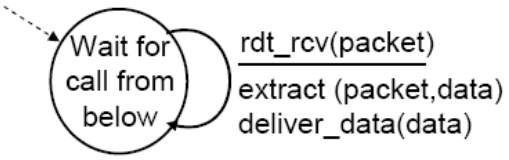
참고 : datagram이라는 용어보다는 일반적인 packet이라는 용어를 사용한다.

rdt 1.0 : 신뢰적인 채널에서 rdt

- 하위 채널이 완전히 신뢰적인 경우
  - no bit error
  - no loss of packets
- FSM
  - 송신측은 상위 layer의 data를 받아 하위 채널로 packet을 전달한다.
  - 수신측은 하위 채널에서 packet을 수신하고 data를 상위 layer로 전달한다.



sender

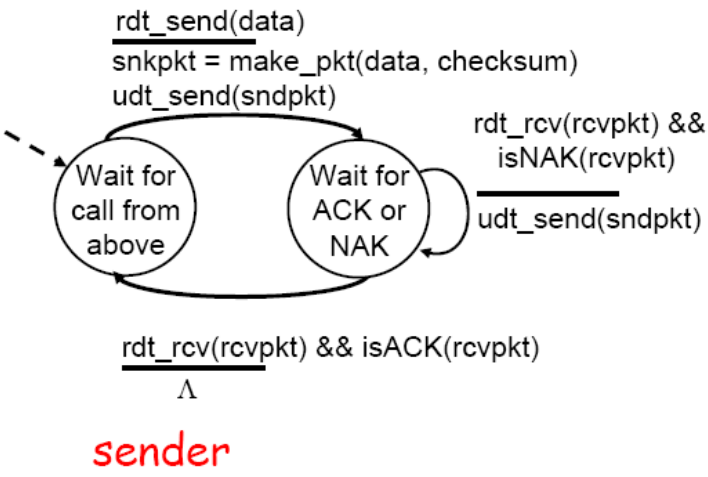


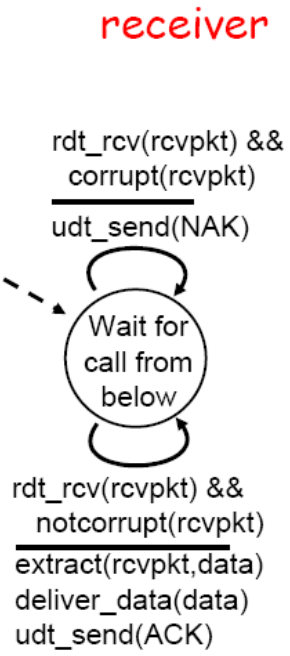
receiver

rdt 2.0 : 비트 오류가 있는 채널

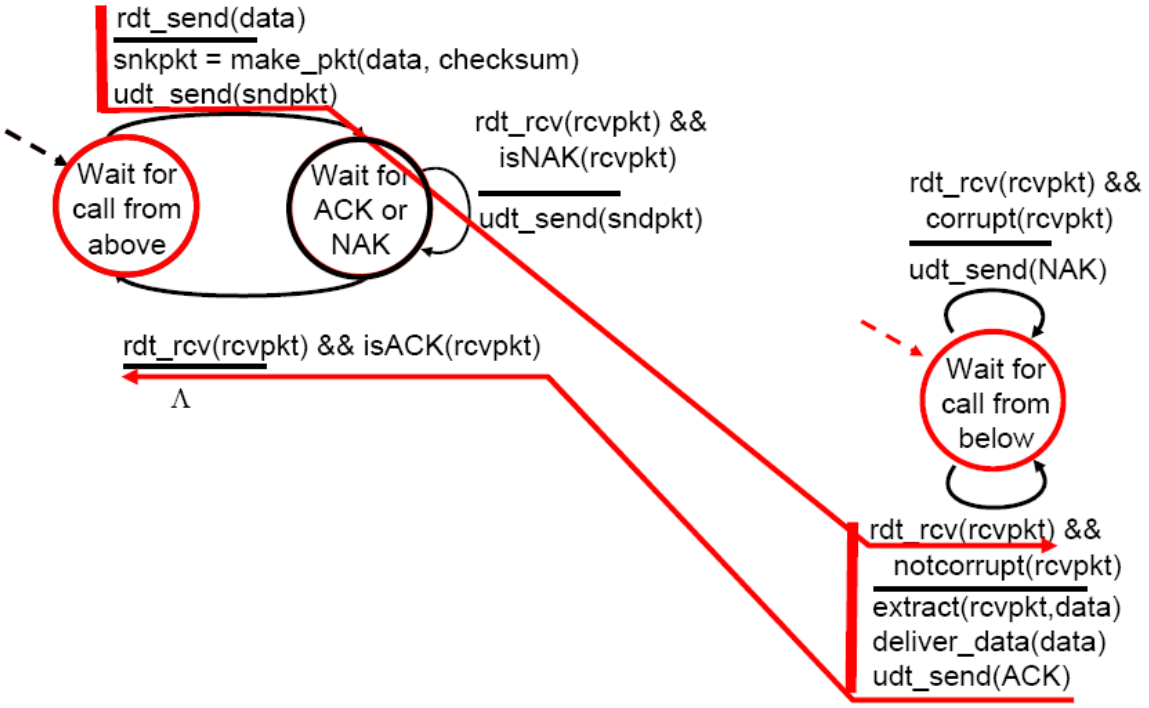
- bit error이 가능한 채널에서 rdt : packet의 손실이나 순서는 보장
  - checksum을 통해 bit error감지
- 수신측의 feedback 필요
  - acknowledgement (ACKs)
    : sender가 receiver에게 pkt를 잘 받았다는 응답
  - negative acknowledgement (NAKs)
    : sender가 receiver에게 pkt에 error또는 장애가 있다는 응답
  - sender는 NAK인 경우 pkt 재전송
- rdt 2.0에 부가적으로 필요한 protocol (rdt 1.0에 비해서)
  - error 검출
  - receiver feedback : 제어 msg (ACK,NAK)
  - 재전송

rdt 2.0 : FSM

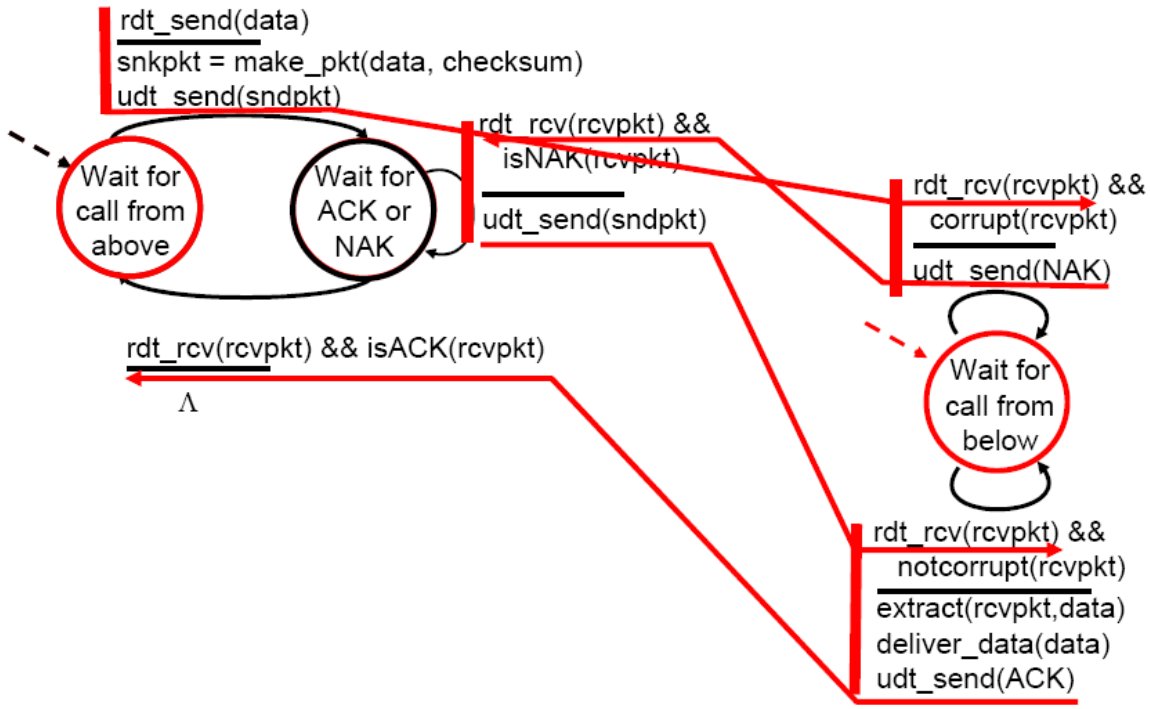




rdt 2.0 : rcvpkt OK



rdt 2.0 : rcvpkt error



rdt 2.0 : 치명적인 결함

ACK/NAK가 회손되는 경우

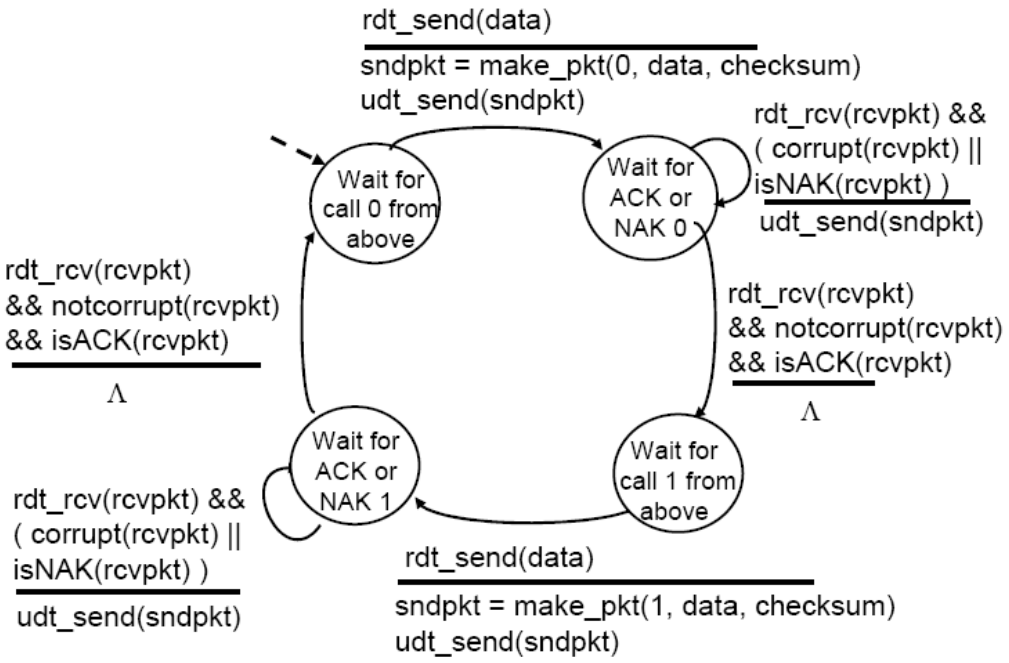
- sender는 receiver쪽에 어떤일이 벌어졌는지 알 수 없다.
- 재전송으로 packet이 중복될 수 있다.

중복을 제어하는 방법

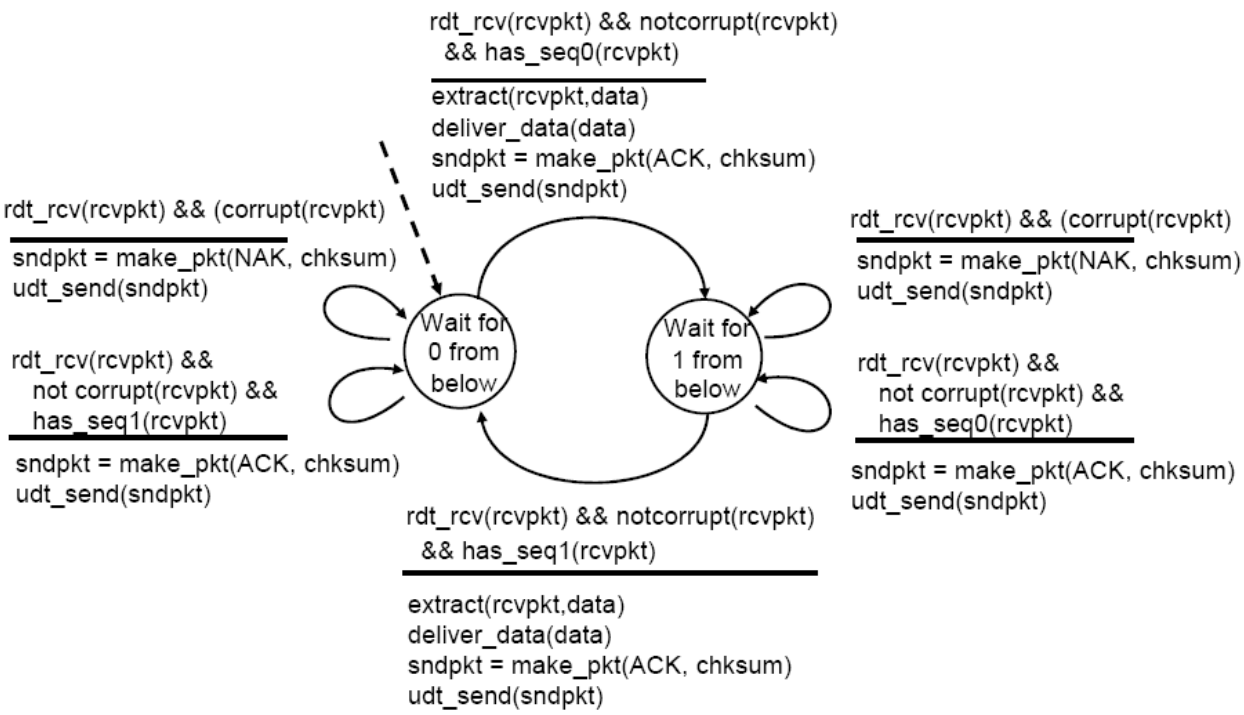
- sender는 pkt에 sequence number를 추가한다.
- ACK나 NAK가 변형되면 sender는 pkt를 재전송한다.
- receiver는 중복된 pkt을 버린다.
- rdt 2.1

stop and wait
Sender는 pkt 하나를 전송한후 receiver 로 부터 response가 올때까지 대기한다.
-> response가 없으면 계속 기다린다.

rdt 2.1 : sender



rdt 2.1 : receiver



rdt 2.1

Sender

- pkt에 seq #를 추가한다.
- 두개의 seq #만으로 충분하다.
  (이전 송신된 pkt와의 중복만을 검사함으로..)
- ACK/NAK의 손상을 검사한다.
- 상태가 rdt 2.0보다 두배 많다.
  - 현재 전송 pkt의 seq #가 0인지 1인지 기억하고 있어야 한다.

receiver

- 수신된 pkt의 중복여부를 검사해야한다.
  - pkt의 seq #가 0/1인지 검사한다.
- receiver는 sender가 ACK/NAK를 정상적으로 수신했는지 알 수 없다.

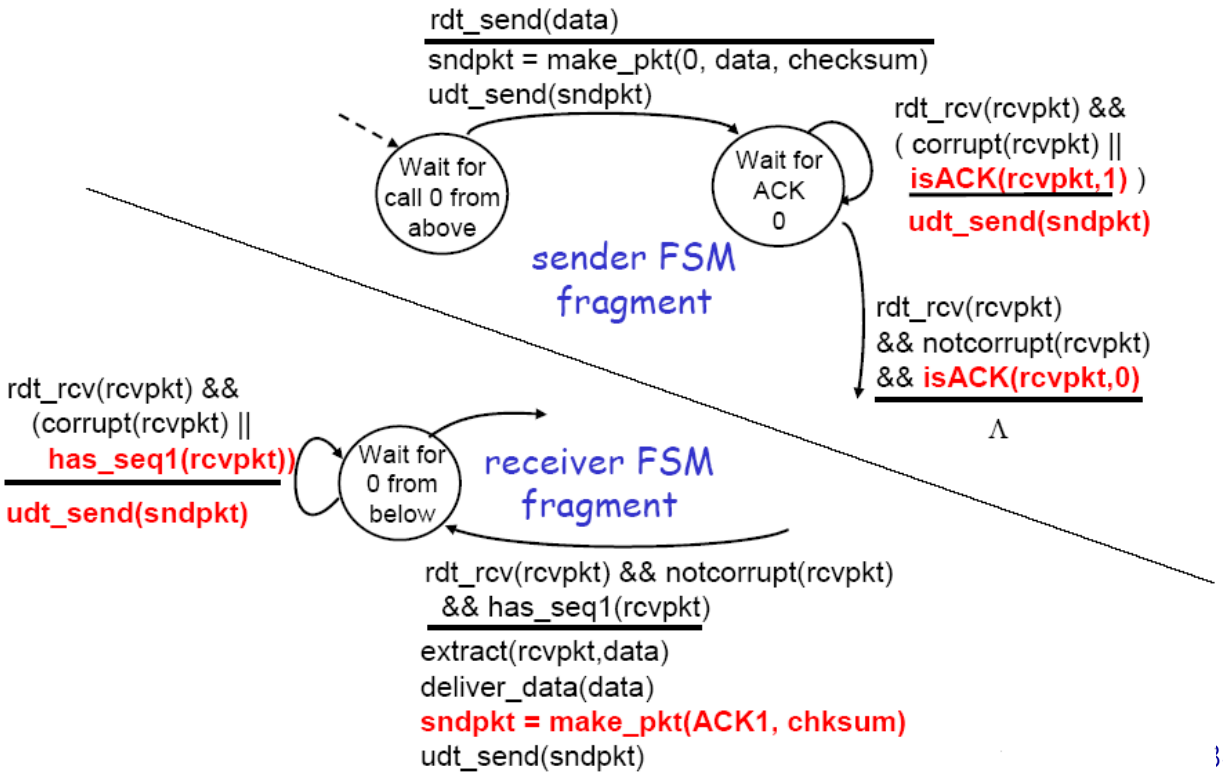
rdt 2.2 : NAK가 없는 rdt

- rdt 2.1과의 차이는 ACKs만을 사용하는 것이다.
- receiver는 NAK 대신 최근에 정확히 수신된 pkt에 대한 ACK를 전달한다.
- Sender는 중복된 ACK를 통해서 중복된 ACK pkt이후에 전송한 pkt이 정상적으로 수신되지 못했음을 안다.

ACK의 누적

누적된 ACK를 사용하는 경우 sender는 마지막 수신한 ACK의 seq # 이전의 pkt는 모두 정상 잔달 되었음을 확인 할 수 있다.

rdt 2.2 : NAK가 없는 rdt



rdt 3.0 : 채널에 error나 loss가 가능

새로운 가정

하위 채널에 packet loss가 일어날 수 있는 경우 (data & ACKs)

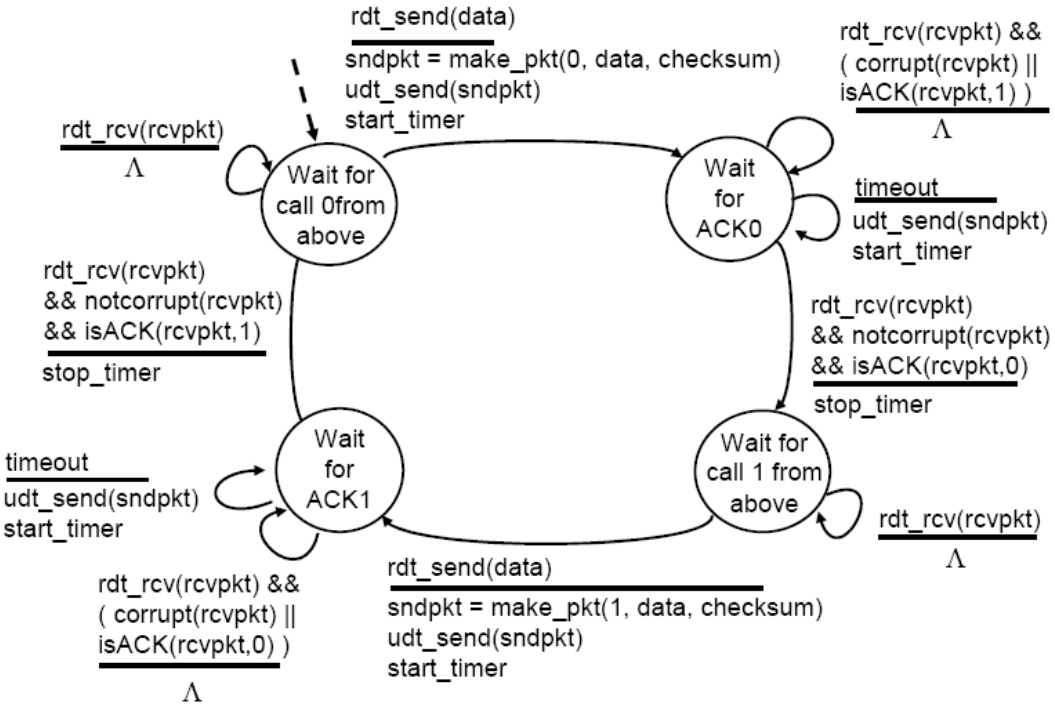
- checksum, seq #, ACKs만으로는 충분하지 않다.

새로운 접근 방식

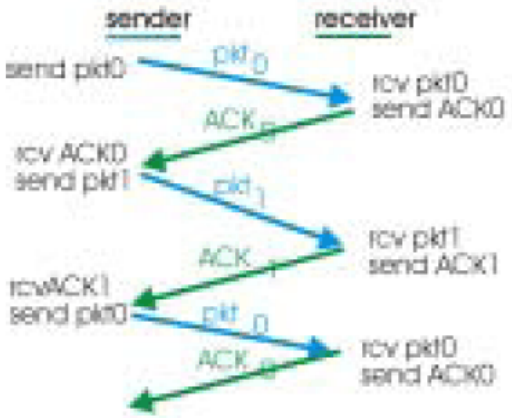
sender는 합리적인 시간 만큼 ack를 기다린다.

- 합리적인 시간만큼 기다려도 ACK가 없으면 재전송한다.
- 만일 ACK가 지연된 것이라면
  - 중복을 유발하는 재전송이 일어난다. – 이미 중복 처리에 대한 준비가 되어있다.
  - receiver는 중복된 pkt에 대한 ACK를 재전송한다.
  - 이러한 작업은 실제로 문제를 일으키지는 않는다.
- 위의 구현을 위해 countdown timer를 필요로 한다.

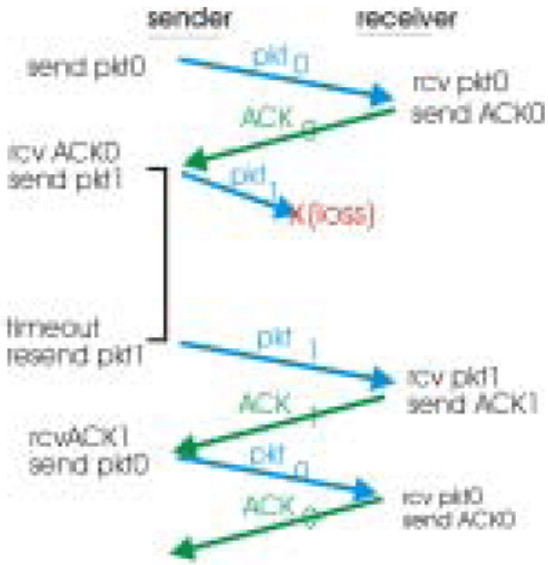
rdt 3.0 : sender



rdt 3.0 in action

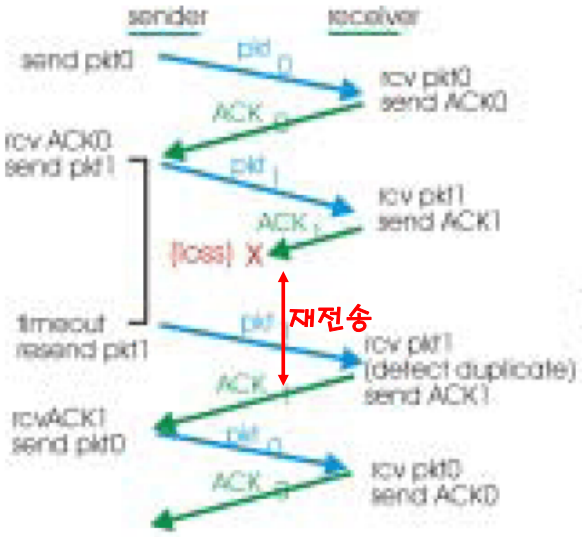


(a) operation with no loss

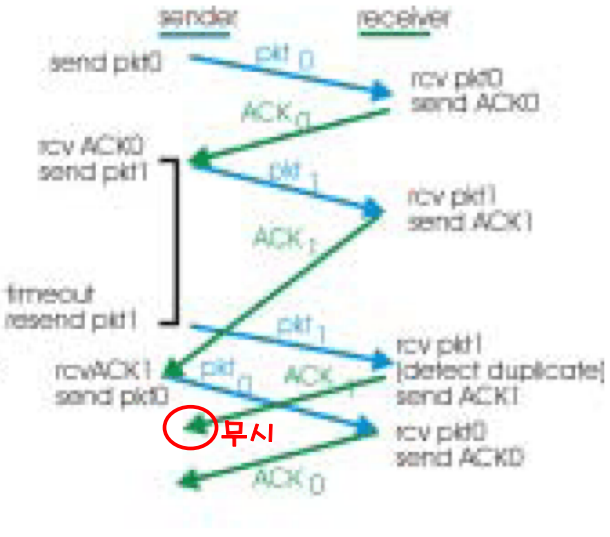


(b) lost packet

rdt 3.0 in action



(c) lost ACK



(d) premature timeout

rdt 3.0 의 성능

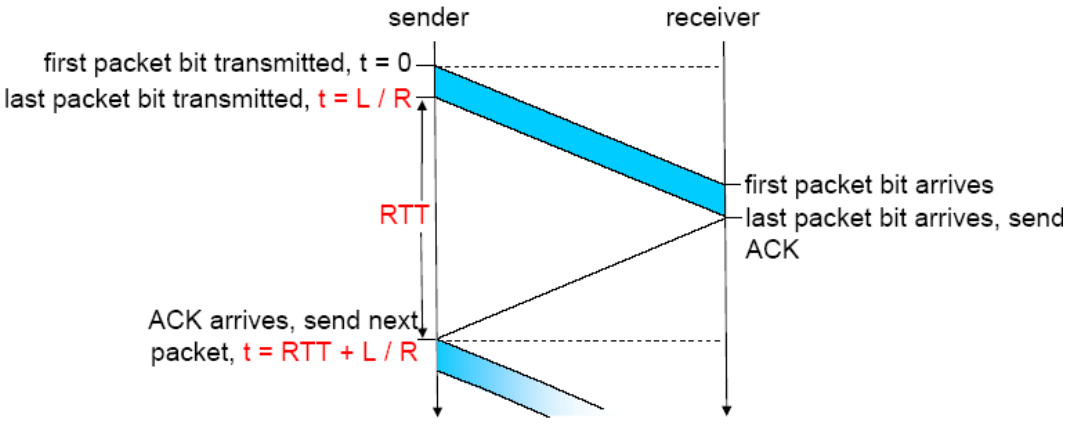
- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8kb/pkt}{10**9 \text{ b/sec}} = 8 \text{ microsec}$$

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

- U_{sender}: utilization - fraction of time sender busy sending
- 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!

rdt 3.0 : stop – and – wait

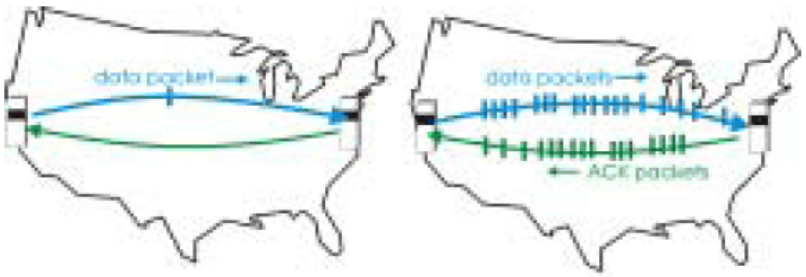


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined protocol

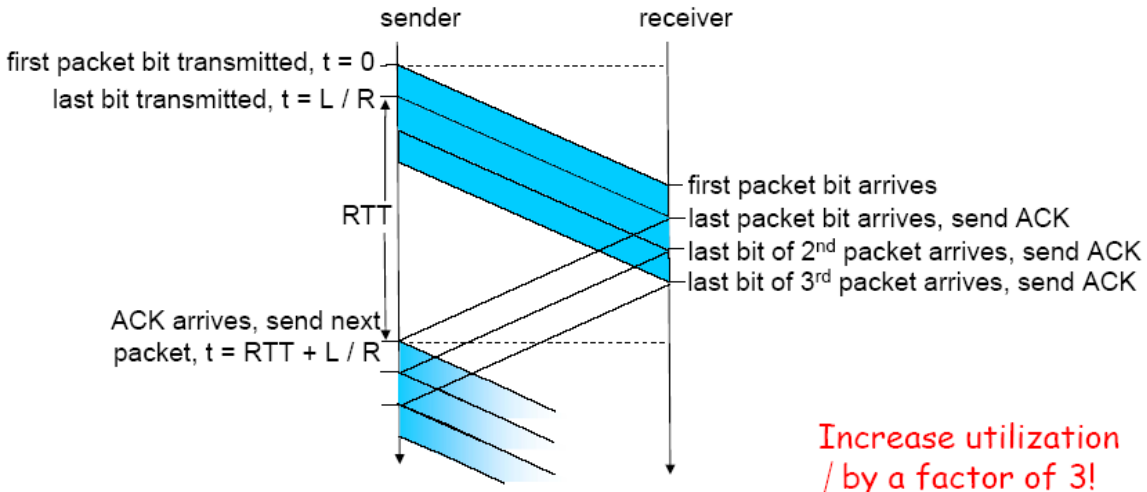
Pipelining : sender에게 ACK를 기다리지 않고 여러 개의 pkt를 전송하도록 하용하는것

- seq #의 범위는 증가되어야 한다.
- sender와 receiver는 하나이상의 pkt를 buffering해야 한다.



◈ piplining protocol : go-Back-N, selective repeat

Pipelining : 성능의 향상



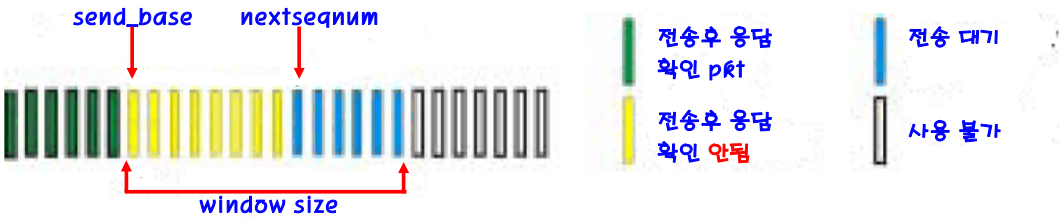
Increase utilization by a factor of 3!

$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Go-Back-N

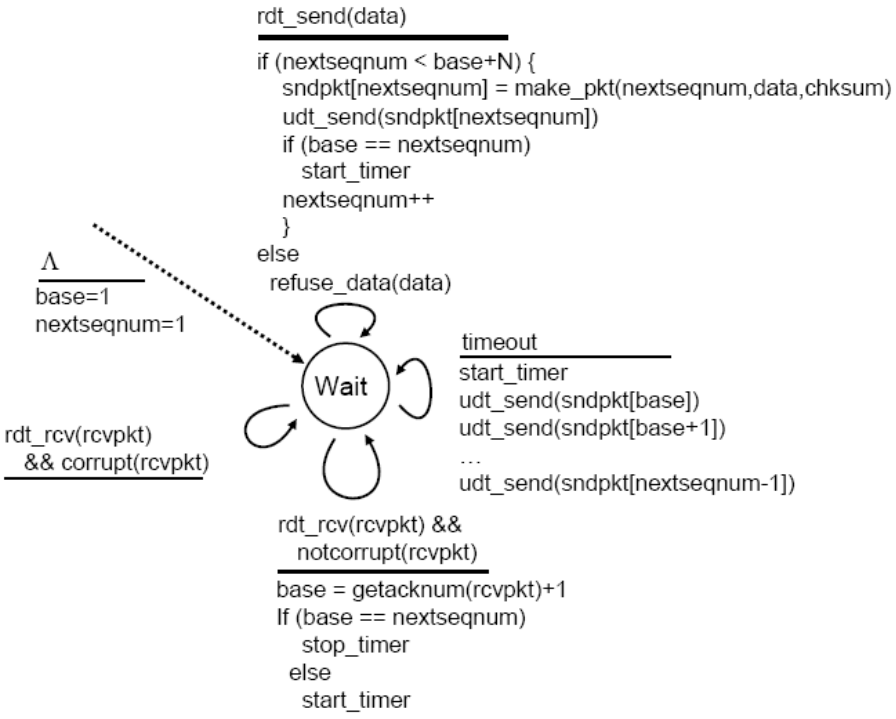
sender

- pkt header에 k-bit seq # 를 갖는다.
- window : 전송되었지만 확인 안된 pkt를 위해 허용 할수있는 seq #의 범위

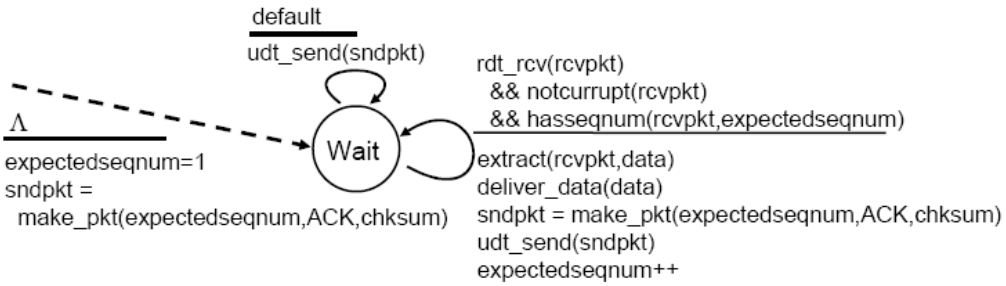


- ACK(n) : seq # n을 가진 ACK를 cumulative ACK로 인식한다.
  - 수신측에서 보면 n을 포함한 n까지의 모든 pkt에 대한 ACK이다.
- 가장 오래된 수신확인 안된 timer를 단일 timer로 사용한다.
- Timeout 발생시 : 송신되었으나 ACK가 없는 모든 pkt를 재전송한다.
  (window내에 전송된 모든 pkt)

GBN : sender FSM



GBN : receiver FSM



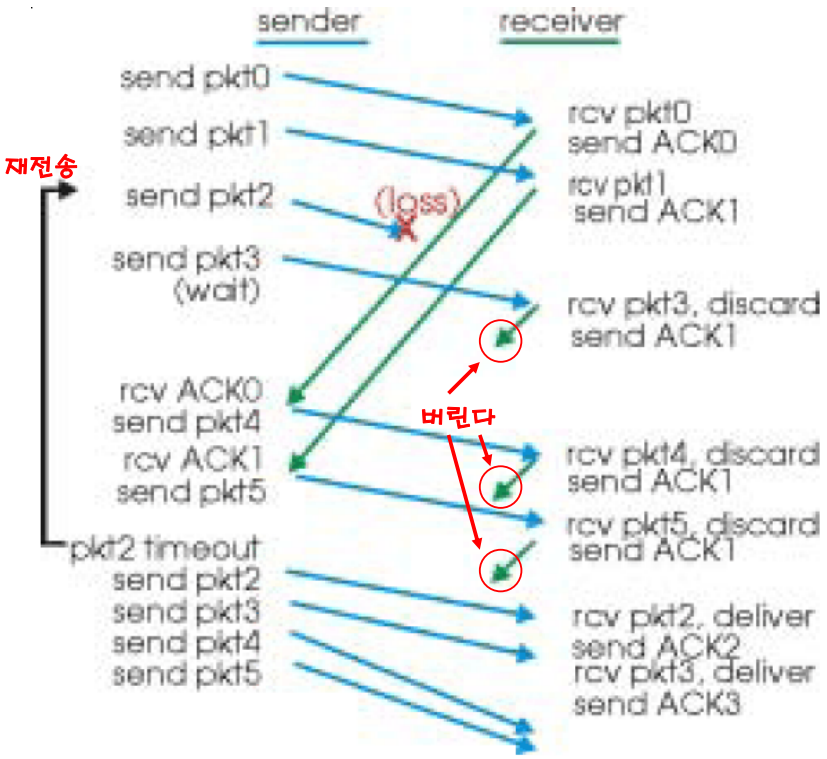
ACK만 사용 : pkt가 수신될때 마다 항상 현재까지 수신된 가장 높은 seq#를 가진 pkt에 대한 ACK를 전송한다.

- 중복된 ACK가 발생 할 수 있다.
- 단지 expectedseqnum만을 유지한다.
  : 현재까지 수신된 pkt의 seq#의 다음 seq#

◈ 순서가 잘못 수신된 pkt의 처리

- 걍 버린다. : 순서가 잘못된 pkt에 대해서 buffering 할 필요가 없다.
- 현재까지 수신된 가장 큰 seq#에 대한 ACK를 재전송한다.

GBN in action



Selective Repeat

- receiver가 수신한 pkt에 대한 개별적인 ACK가 요구 된다.
  - 순서가 틀린 pkt들은 하위 seq#를 가진 pkt가 수신 될때까지 buffering되었다가 seq# 순서대로 상위 layer에 전달된다.
- sender는 ACK가 없는 pkt에 대해서만 재전송한다.
  - sender의 timer는 ACK가 없는 각각의 pkt에 대해서 유지된다.
- sender 의 window
  - send_base pkt에 대한 ACK를 받아야 이동한다.
    : GBN에서는 누적된 ACK를 이용 함으로 send_base pkt에 대한 ACK가 없어도 이동가능하다.

Selective repeat : sender receiver windows

Selective repeat

sender

상위 layer에서 Data가 수신되면

- pkt의 다음 seq#를 검사 window내에 있으면 pkt를 전송한다.

timeout(n)

- pkt n을 재전송하고 timer를 restart한다.

ACK(n) 이 수신

- n이 윈도우에 있다면 pkt가 수신된것을 확인한다.
- n이 send_base와 같다면 send_base는 가장 작은 seq#를 갖는 미확인 pkt로 이동하고 window내에 이 전송 pkt가 있으면 전송한다.

receiver

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

- ACK(n)을 송신
- out-order인경우 buffering한다.
- in-order인 경우 필요하다면 buffer에 저장된 번호가 연속적인 pkt와 함께 상위 layer에 전달하고 rcv_base를 가장 낮은 seq#를 가진 미전송 pkt로 옮긴다.

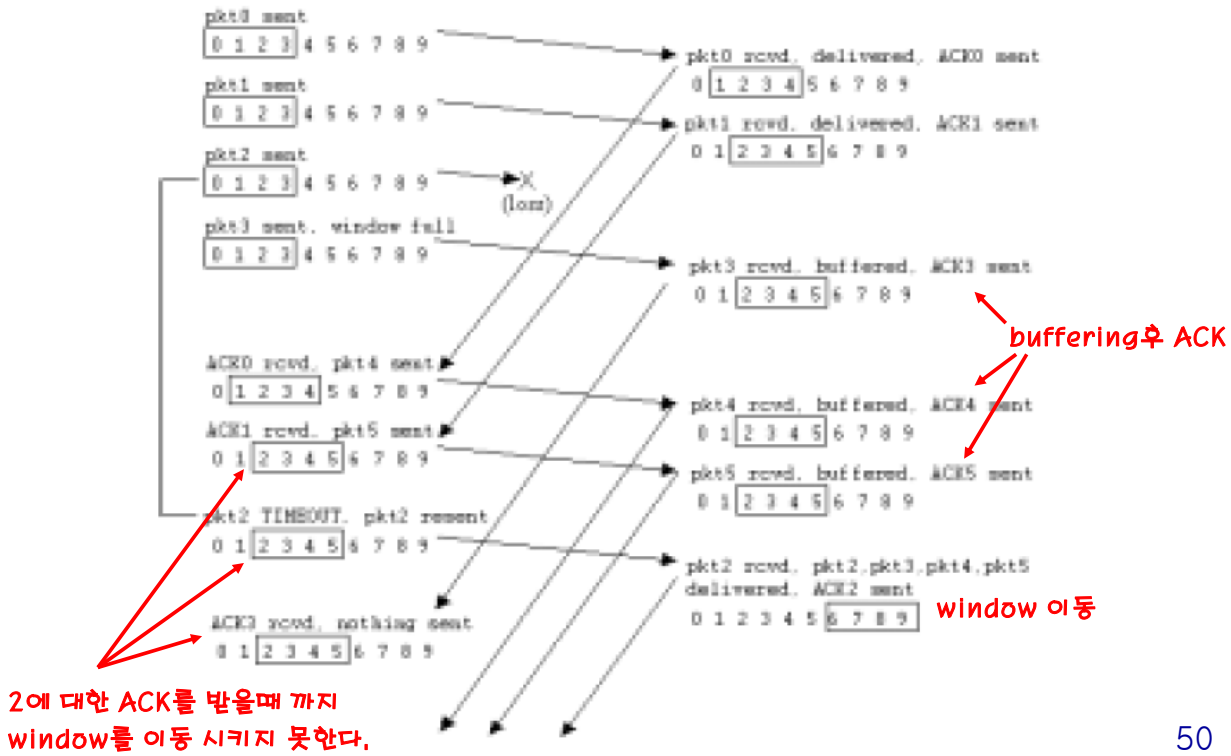
pkt n [n in rcvbase-N, rcvbase-1]

- ACK(n)

이외의 경우

- 무시한다.

Selective repeat in action



Selective repeat 의 문제

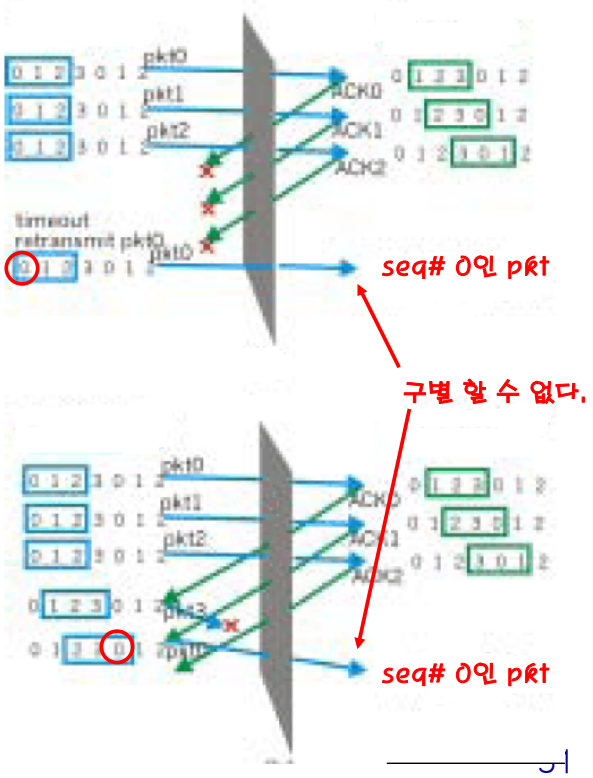
Example

- Seq# : 0, 1, 2, 3
- windows size = 3

- 왼쪽 두개의 경우 모두 seq#가 0 으로 구별이 불가능하다.

Q; windows size와 seq#의 관계를 어떻게 하면 이문제를 해결 할 수 있는가

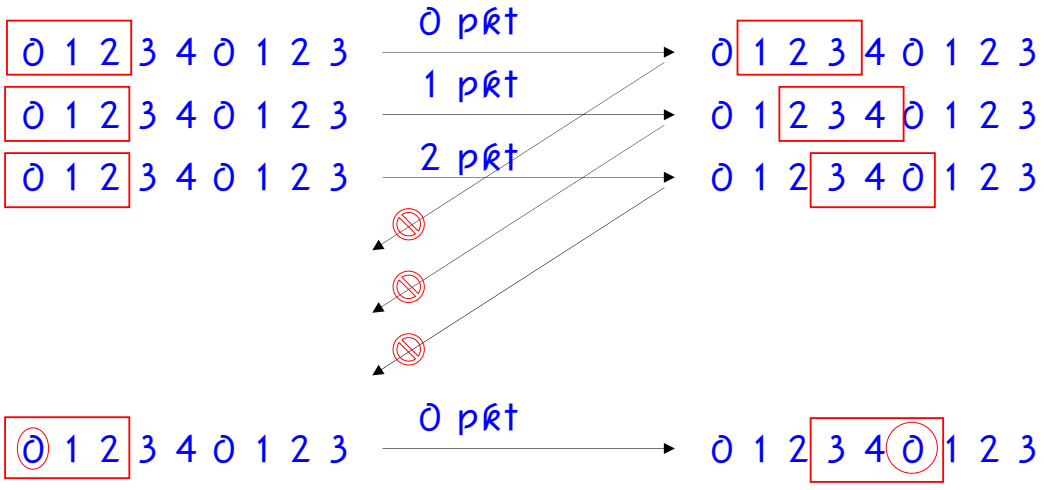
windows size를 seq#의 개수의 절반 이하로 한다.



Selective repeat 의 문제

Example

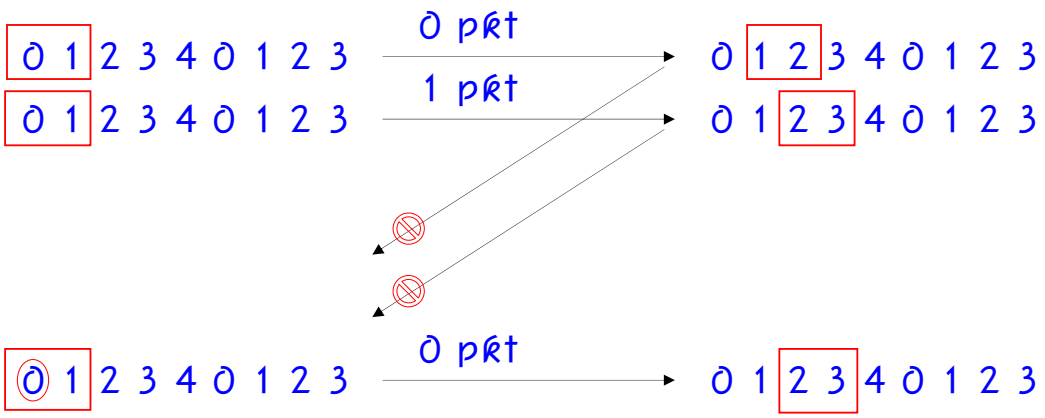
- Seq# : 0, 1, 2, 3, 4
- windows size = 3



Selective repeat 의 문제

Example

- Seq# : 0, 1, 2, 3, 4
- windows size = 2



3장 Transport Layer

3.1 Transport Layer 서비스 개요

3.2 다중화와 역다중화

3.3 Connectionless transport : UDP

3.4 신뢰적 data 전송의 원리

3.5 Connection oriented transport : TCP

- segment structure
- reliable data transfer
- flow control
- connection control

3.6 혼잡제어의 원리

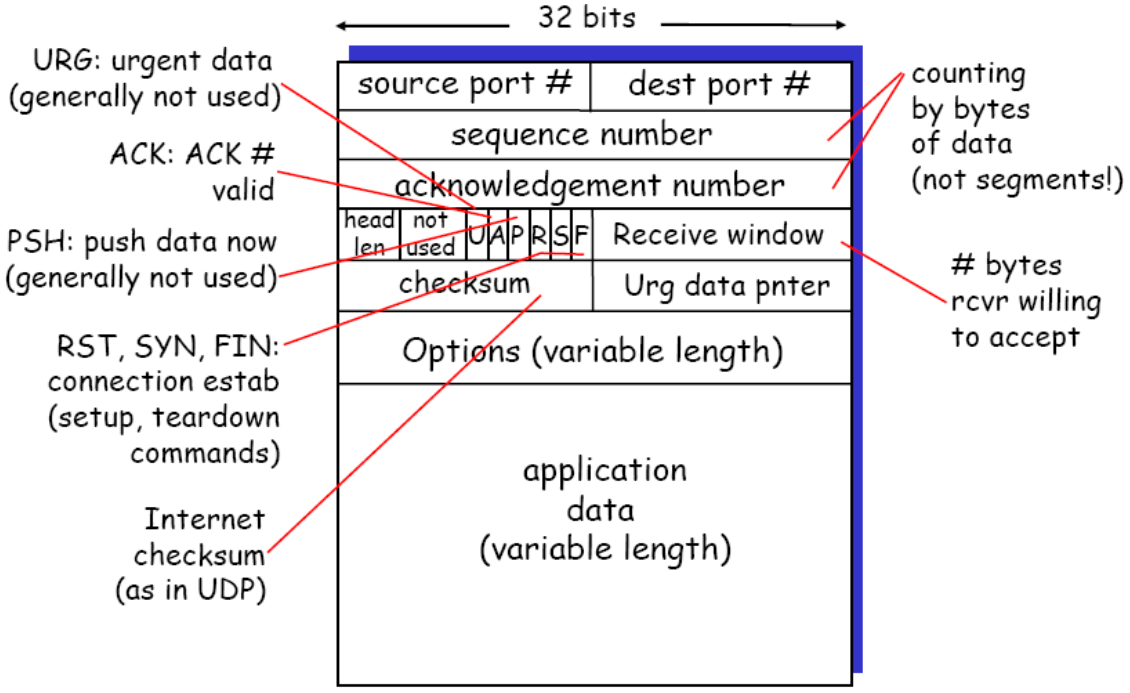
3.7 TCP의 혼잡제어

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

- point-to-point
  - 단일 송,수신자 간에 통신
- 신뢰적인 in-order byte stream
  - Message에 구분이 없다.
- pipelined
  - 혼잡제어나 흐름제어를 통해 window size를 제어한다.
- 송,수신측은 buffer를 갖는다.

- full duplex
  - 동일 connection에 양단이 동시에 data를 전송 할 수 있다.
  - MSS : maximum segment size (segment에서 app layer data의 최대 크기)
- connection oriented
  - handshake를 먼저 수행한다.
- Flow control
  - sender가 receiver를 압박하지 못하도록 receiver가 전송량을 통제한다.

TCP segment의 구조



TCP seq#와 ACK

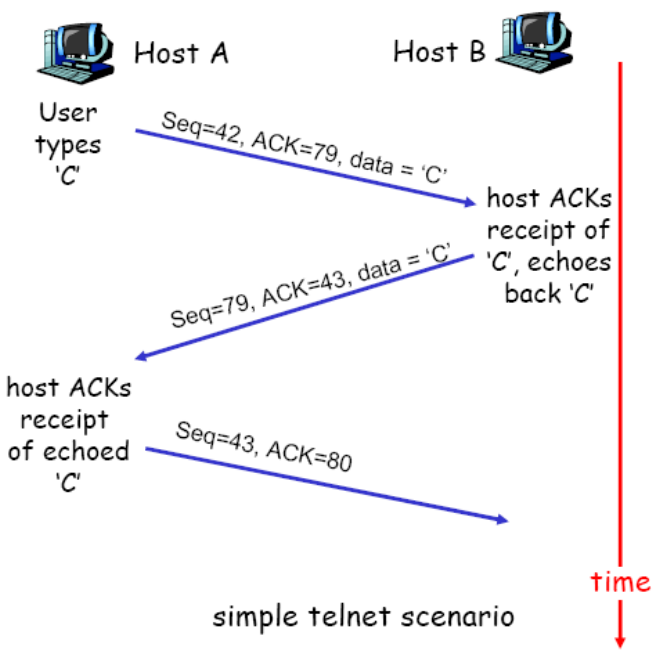
- seq#
  - Segment의 첫번째 byte의 stream에서의 byte 순서 번호
- ACKs:
  - 다음에 받을 첫번째 byte의 순서 번호
  - Cumulative ACK가 가능하다.

Q :순서가 틀린 segment를 수신한 경우 어떻게 할까??

A : RFC에서는 어떤 규칙도 없다.

기본적으로 두가지 방법이 있다.

- 즉시 버린다.
- buffering한다. (중간에 빠진 data를 받기 위해,,)



시작 seq#는 종료된 접속의 남아있는 seg와 유효한 seg와의 혼동을 막기위해 임의로 선택된다. 57

TCP : RTT(Round Trip Time) & Timeout

Q : TCP의 timeout는 어떻게 설정 할까???

- RTT보다 커야한다.
  - but RTT 값은??
- 너무 작은 경우
  - 불필요한 재전송이 일어난다.
- 너무 큰경우
  - Segment 손실에 대한 대응이 늦어진다.

Q : EstimatedRTT는 어떻게 설정 할까???

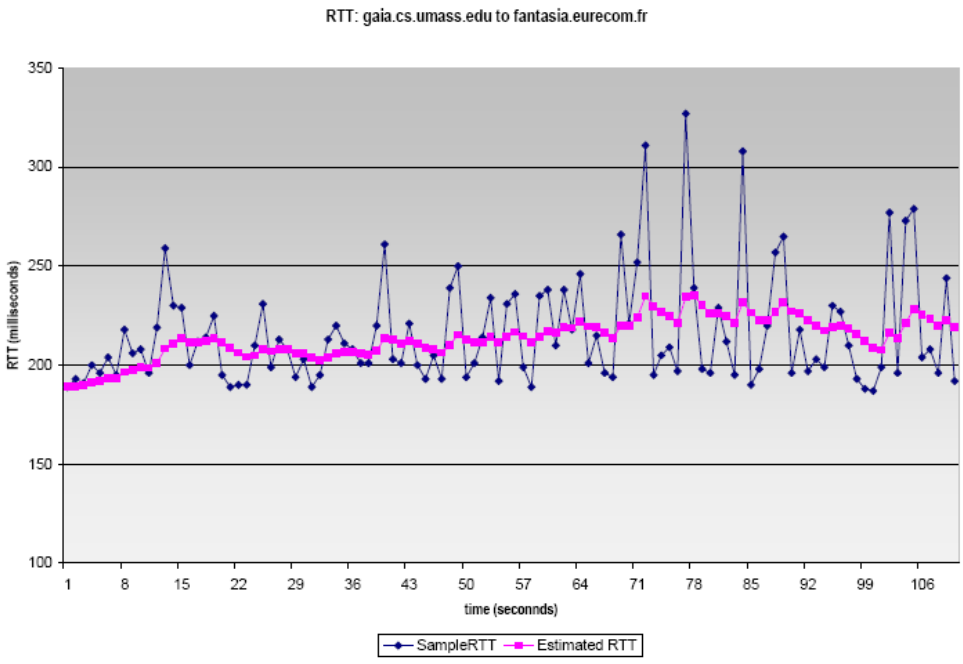
- SampleRTT
  : segment가 송신된 시간으로부터 ACK가 도착한 시간간격
  - 재전송 segment는 무시된다.
- EstiamteRTT는 매우 smooth하게 변경된다.
  - EstiamatedRTT는 SampleRTT의 가중 평균 값이다.

TCP : RTT(Round Trip Time) & Timeout

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

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: α = 0.125

RTT 추정치 예제



TCP : RTT(Round Trip Time) & Timeout

timeout 설정

◈ EstimatedRTT보다 약간의 여유값을 더한 값으로 설정

❍ SampleRTT의 변화량이 큰경우는 따라서 커져야한다. (반대의 경우도 동일)

◈ DevRTT

: SampleRTT가 EstimatedRTT로 부터 얼마나 벗어나느지에 대한 예측

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

(β의 권장값 : 0.25)

실제 timeout 설정

TimeoutInterval = EstimatedRTT + 4*DevRTT

3장 Transport Layer

3.1 Transport Layer 서비스 개요

3.2 다중화와 역다중화

3.3 Connectionless transport : UDP

3.4 신뢰적 data 전송의 원리

3.5 Connection oriented transport : TCP

❍ segment structure

❍ reliable data transfer

❍ flow control

❍ connection control

3.6 혼잡제어의 원리

3.7 TCP의 혼잡제어

TCP reliable data transfer

- TCP는 비신뢰적인 IP 상위에서 신뢰적인 통신 서비스를 제공한다.
- Pipelined segment
- 누적 ACK
- TCP는 단일 재전송 timer를 사용한다.
  - 전송 segment 각각에 대한 timer는 개념적으로는 쉽지만 overhead가 크다.
- 재전송을 일으키는 이벤트
  - timeout
  - 중복 ACK
- 간소화된 TCP sender
  - 중복 ACK 무시
  - 혼잡제어, 흐름제어 무시

TCP sender event

App로 부터 data 수신 :

- seq#를 가진 segement 생성
- seq#는 첫번째 바이트의 바이트스트림 번호이다.
- timer 시작
  (다른 segment에서 의해서 timer가 실행중이 아닐때)
- timer의 만료 주기
  TimeoutInterval

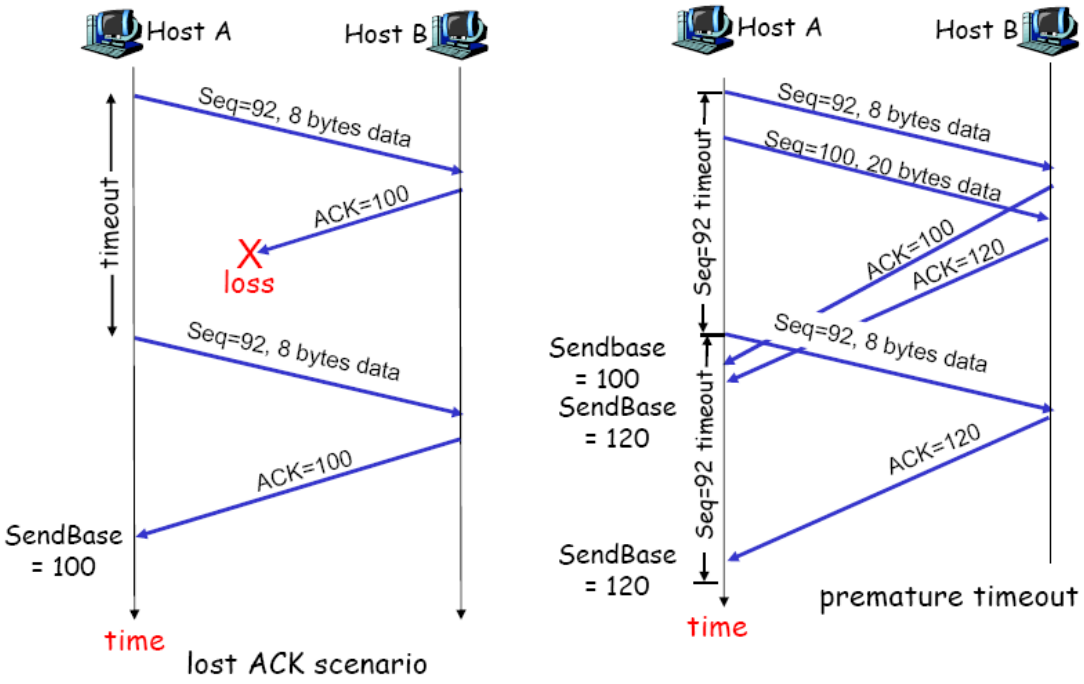
timeout :

- timeout을 일으킨 segment를 재전송
- timer 재시작

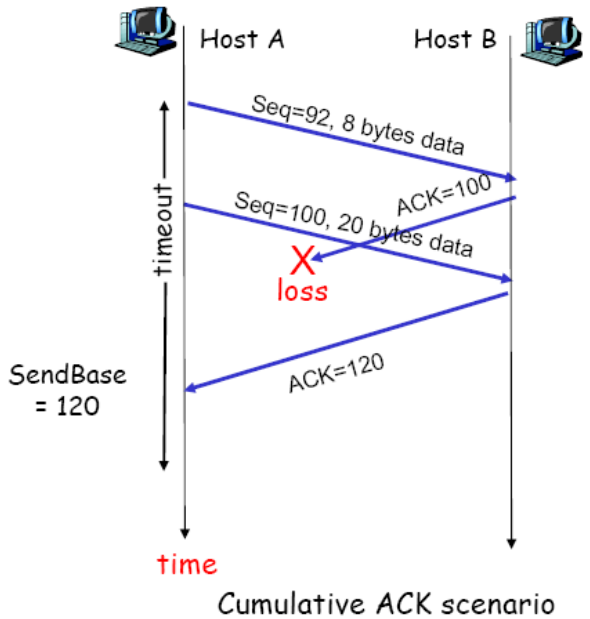
ACK :

- ACK가 확인 안된 segment의 ACK인경우
  - ACK의 seq#가 sendbase보다 크면 이전에 확인 응답안된 segment에 대해 확인하고 sendbase를 갱신한다.
  - 아직 확인 응답 안된 segment가 존재하면 timer를 재시작한다.

TCP의 흥미로운 시나리오



TCP의 흥미로운 시나리오



TCP의 ACK생성 권고

Receiver의 event	Receiver의 동작
기다리는 seq#를 가진 segment도착, 이미 기다리는 seq#까지의 모든 segment 수신	지연된 ACK, 다음 순서의 segment를 500ms 기다렸다가 도착하지 않으면 ACK를 전송한다.
기다리는 seq#를 가진 segment도착, 이전에 수신한 seq에대한 ACK는 아직 전송전	하나의 누적된 ACK를 전송
기다리는 것보다 높은 순서의 seq#를 가진 segment가 도착, Gap이 발견	즉시 중복된 ACK전송 (기다리는 다음 순서의 seq#)
수신 segment의 gap부분에 해당하는 segment 도착	즉시 ACK전송 (segment가 gap의 낮은 쪽에서 시작하도록)

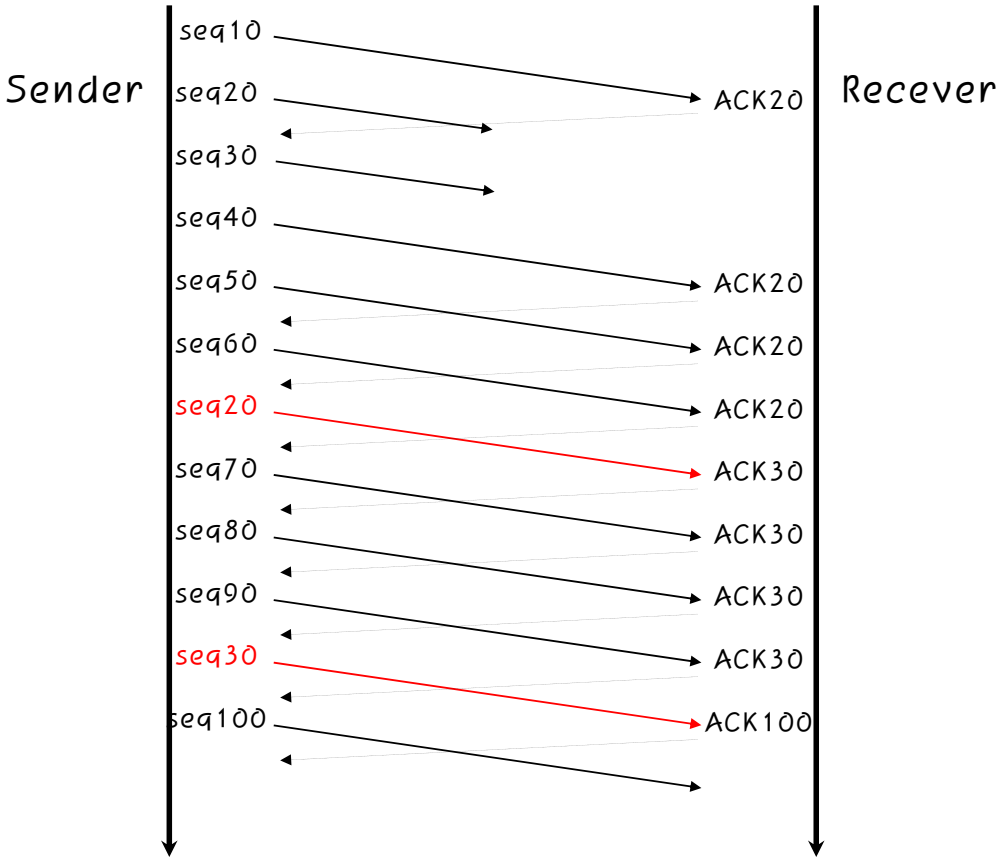
빠른 재전송

- timeout이 유발하는 재전송의 문제
  - 주기가 비교적 길어서 segment를 잃어 버렸을때 송신자를 기다리게 해서 종단간의 지연이 증가한다.
- 중복된 ACK와 segment loss
  - 수신자는 기다리는것보다 큰 seq#의 segment를 수신하면 이를 buffering하고 중복 ACK를 전송한다.
  - segment를 손실하면 많은 중복된 ACK가 전송된다.

Seq#가 10, 20, 30, 40, ...인 경우 20이 loss되면 30, 40, 50, ... 을 보낼때 마다 ACK(20)이 중복해서 sender에게 전달 된다.

- 만일 3번의 중복된 ACK가 sender에게 전송되면 해당 segment가 loss되었다고 판단한다.
  - 이경우 timeout이전이라도 해당 segment를 재전송한다.
    -> fast retransmit

빠른 재전송



Fast retransmit algorithm

```
event: ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase = y
                if (there are currently not-yet-acknowledged segments)
                    start timer
              }
            else {
                increment count of dup ACKs received for y
                if (count of dup ACKs received for y = 3) {
                    resend segment with sequence number y
                }
```

a duplicate ACK for already ACKed segment

fast retransmit

3장 Transport Layer

3.1 Transport Layer 서비스 개요

3.2 다중화와 역다중화

3.3 Connectionless transport : UDP

3.4 신뢰적 data 전송의 원리

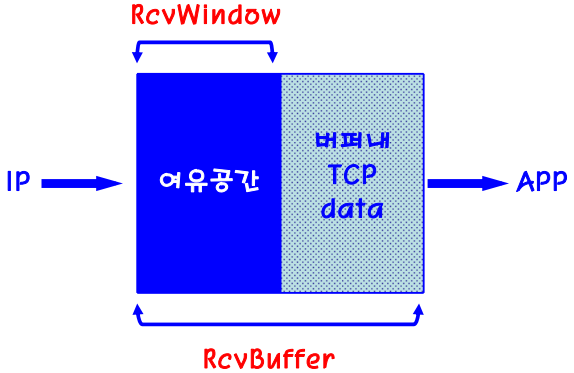
3.5 Connection oriented transport : TCP
- segment structure
- reliable data transfer
- flow control
- connection control

3.6 혼잡제어의 원리

3.7 TCP의 혼잡제어

TCP Flow control

◈ TCP connection에서 수신측이 가용한 버퍼의 크기를 알려주어 흐름을 제어한다.

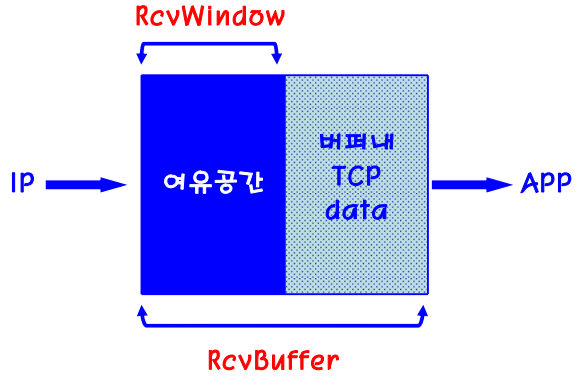


◈ App process가 수신 buffer로부터 data를 늦게 읽으면 RcvWindow가 줄어들수 있다.

flow control
Sender가 receiver의 buffer가 overflow 되도록 너무 빠르게 data를 전송하지 못하도록 하는것

◈ 흐름제어는 속도를 일치시키는 서비스이다.
- Receiver쪽 app이 data를 읽는 속도와 sender의 전송 속도를 일치 시키는 것이다.

TCP Flow control



(out-of-order segment는 버린다고 가정)

◈ 버퍼내에 여유 공간
= RcvWindow
= RcvBuffer – [LastByteRcvd – LastByteRead]

◈ segment내에 RcvWindow size를 저장 receiver가 sender에게 알려줌

◈ Sender는 RcvWindow크기보다 적은 확인 응답된 data량을 유지함으로써 수신 buffer에 overflow가 발생하지 않았음을 확신한다.

3장 Transport Layer

3.1 Transport Layer 서비스 개요

3.2 다중화와 역다중화

3.3 Connectionless transport : UDP

3.4 신뢰적 data 전송의 원리

3.5 Connection oriented transport : TCP
- ❍ segment structure
- ❍ reliable data transfer
- ❍ flow control
- ❍ connection control

3.6 혼잡제어의 원리

3.7 TCP의 혼잡제어

TCP의 connection 관리

Recall: TCP sender와 receiver 는 connection을 초기화한 다음 data segment을 전송한다.

- TCP 초기화 변수
  - Seq#, buffer, flow control 정보들
    ex) RcvWindow
- client : connection initiator
- server : client의 접속을 승인

Three way handshake:

step 1: client가 server에게 TCP SYN segment를 전송
- 초기 seq# 설정
- data는 없다.

step 2: server는 SYN를 수신하고 SYNACK를 전송
- server : 변수및 buffer 할당
- server의 초기 seq# 설정

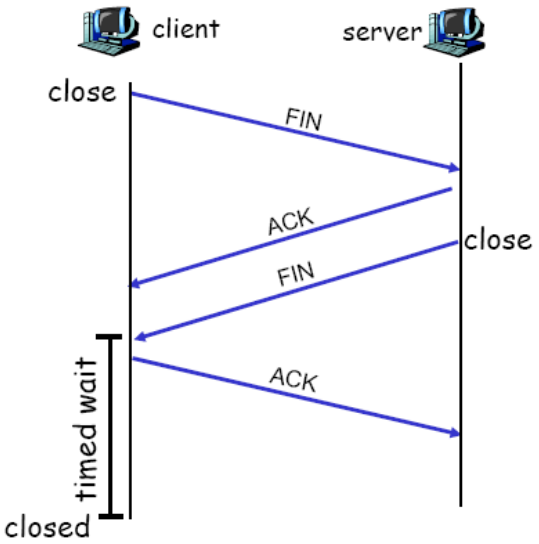
step 3: client는 SYNACK수신 ACK전송 (data를 추가해도 된다.)
- client : 변수및 buffer 할당

TCP의 connection 관리

Closing a connection:

Step 1: client는 connection 을 종료하기위해 FIN bit가 1로 설정된 TCP segment를 server에게 전달

Step 2: server는 FIN을 수신하면 ACK를 응답하고 connection을 종료한다는 FIN를 client에게 전송한다.

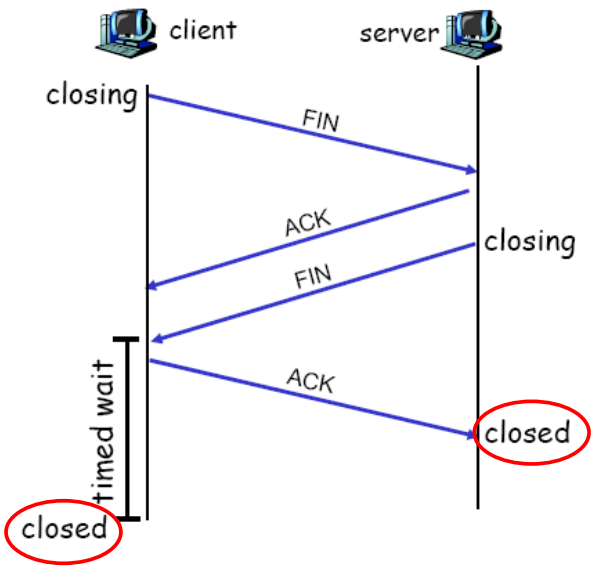


TCP의 connection 관리

Step 3: client는 FIN을 수신하면 ACK를 응답하고 일정시간을 기다린후 (time wait) connection을 종료한다.

Step 4: server는 ACK를 수신하면 connection을 종료한다.

Note : with small modification can handle simultaneous FINs.



3장 Transport Layer

3.1 Transport Layer 서비스 개요

3.2 다중화와 역다중화

3.3 Connectionless transport : UDP

3.4 신뢰적 data 전송의 원리

3.5 Connection oriented transport : TCP
- segment structure
- reliable data transfer
- flow control
- connection control

3.6 혼잡제어의 원리

3.7 TCP의 혼잡제어

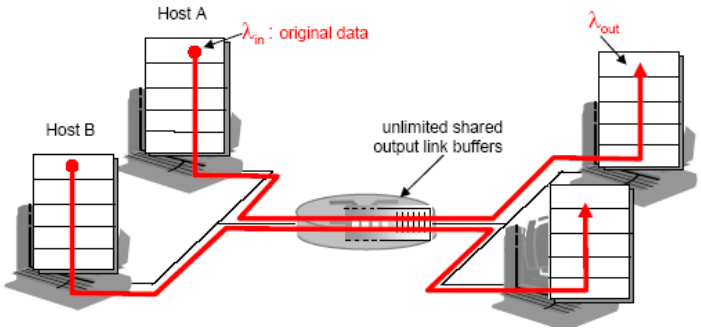
혼잡제어의 원리

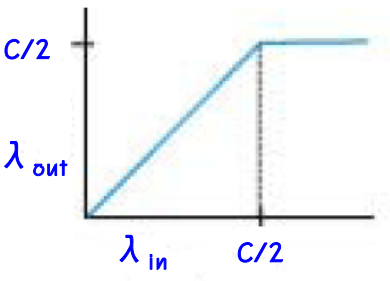
Congestion :

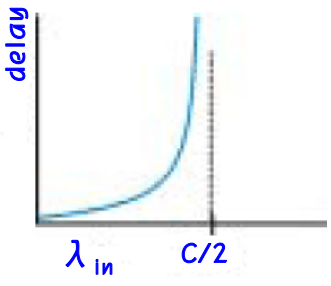
- 높은 전송률로 data를 전송하려는 많은 근원지로 인한 문제
- flow control과는 다르다.
- 현상
  - pkt의 손실 (router buffer의 overflow)
  - 지연의 증가 (router buffer에서 queueing delay 증가)
- Sender에 의해서 조정
  - 실제 혼잡도를 조사하는것은 아니다 (TCP)
  - ATM등가 같이 혼잡도를 조사하기 위해서는 네트워크 장치들의 협조가 필요함
- 다음과 같은 경우 Data 전송량을 줄인다.
  - timeout이 빈번히 일어날때
  - 재전송량이 많아 질때

congestion scenario 1

- 두개의 sender와 reciver
- 무한 버퍼를 가진 하나의 라우터
- 재전송은 없다.



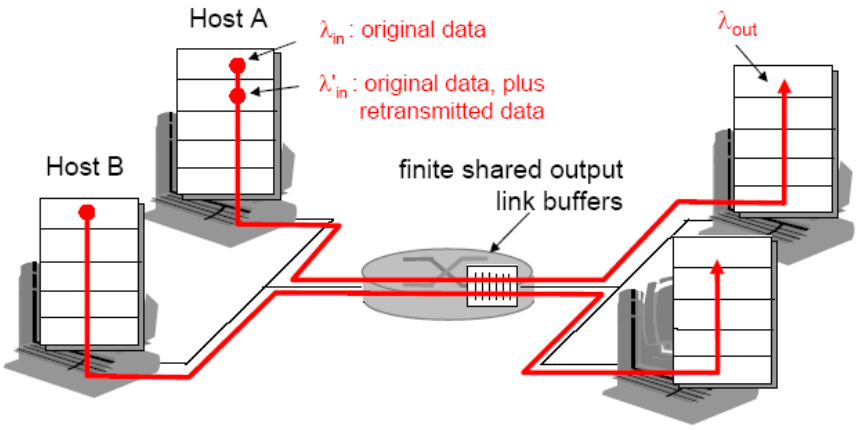




- 혼잡시에 매우큰 지연이 있다.
- 패킷의 처리량이 링크의 용량에 접근하게 되면 매우큰 queueing delay가 발생한다.

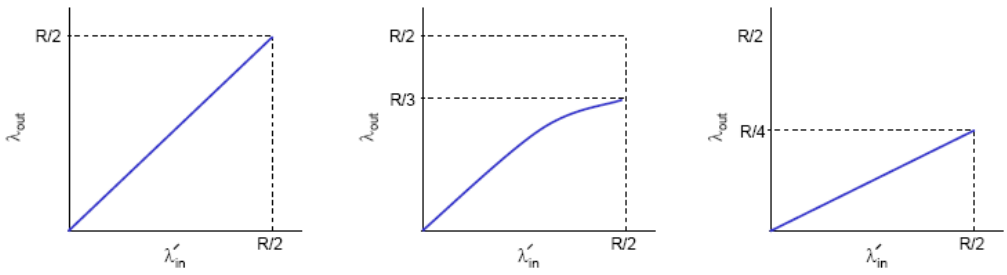
congestion scenario 2

- 유한 buffer를 가진 라우터
- 손실된 pkt에 대한 재전송이 이루어진다.



congestion scenario 2

- 손실이 발생하지 않는 이상적인 경우 : $\lambda_{in} = \lambda_{out}$ (그림 a)
- 손실된 pkt만 재전송하는 perfect한 경우 : $\lambda'_{in} > \lambda_{out}$ (그림 b)
- pkt의 delay로 인한 불필요한 재전송으로 λ'_{in}이 매우 커지고 λ_{out}이 더욱 줄어든다. (그림 c)



congestion의 cost

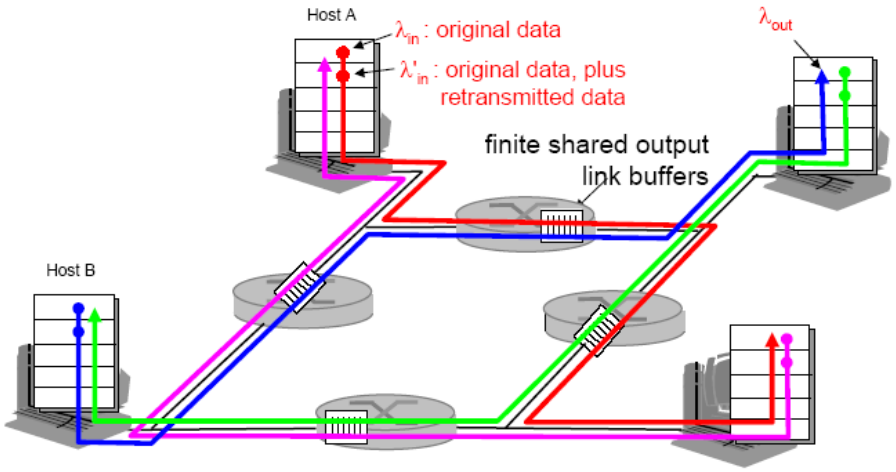
- Buffer overflow로 인해 버려진 pkt에 대한 재전송
- 큰 지연으로 인한 불필요한 재전송

congestion scenario 3

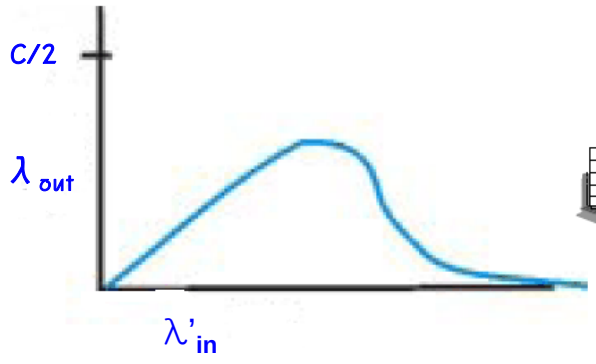
- 4개의 sender
- 여러 개의 라우터와 다중 경로
- timeout/재전송

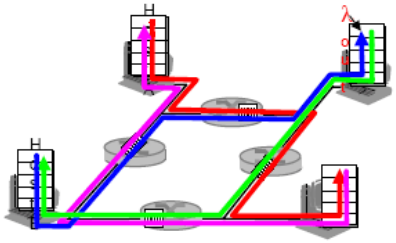
Q : λ_{in} 이나 λ'_{in} 가 커지면 어떤 현상이 일어나는가??

A : 다음 장의 도표를 참고



congestion scenario 3





congestion의 또하나의 cost

- 혼잡으로 인해 pkt가 경로상에서 버려질때 버려지는 지점까지 pkt를 전송하는데 사용된 router의 전송 용량은 쓸모없는것이 되었다.

congestion control에 대한 접근 방법

End-end congestion control (종단간의 혼잡제어)

- Network으로 부터 어떠한 feedback도 제공되지 않는다.
- 손실이나 지연등의 현상을 관찰해서 end system이 추측해야한다.
- TCP가 사용하는 방식

Network-assisted congestion control (네트워크지원 혼잡제어)

- router가 혼잡상태에 관한 정보를 sender에게 직접적인 feedback을 제공한다.
  - 하나의 bit로 혼잡을 나타낸다. (SNA, ATM, DECbit...)
  - 전송률을 sender에게 명확히 전달한다.
- 두가지 혼잡 data 전송방식
  - choke pkt – 라우터가 sender
  - 송,수신자 사이의 pkt의 field에 추가

ATM ABR 혼잡제어

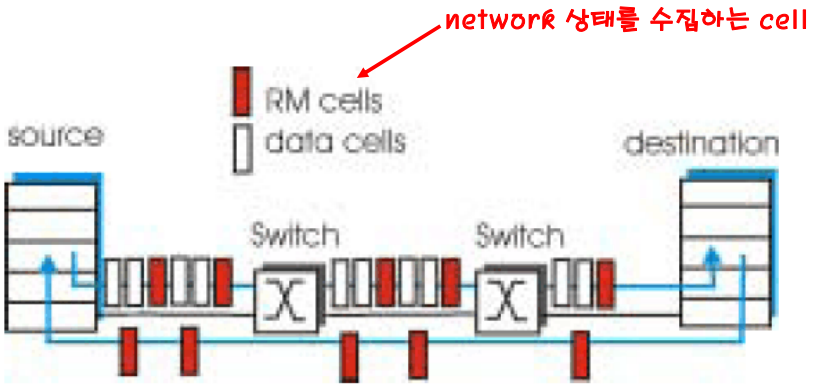
ABR: available bit rate:

- "elastic service"
- if sender's path "underloaded":
  - sender should use available bandwidth
- if sender's path congested:
  - sender throttled to minimum guaranteed rate

RM (resource management) cells:

- sent by sender, interspersed with data cells
- bits in RM cell set by switches ("*network-assisted*")
  - NI bit: no increase in rate (mild congestion)
  - CI bit: congestion indication
- RM cells returned to sender by receiver, with bits intact

ATM ABR 혼잡제어



- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - sender' send rate thus minimum supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, sender sets CI bit in returned RM cell

3장 Transport Layer

3.1 Transport Layer 서비스 개요

3.2 다중화와 역다중화

3.3 Connectionless transport : UDP

3.4 신뢰적 data 전송의 원리

3.5 Connection oriented transport : TCP
- segment structure
- reliable data transfer
- flow control
- connection control

3.6 혼잡제어의 원리

3.7 TCP의 혼잡제어

TCP congestion control

- end-end control (network으로부터 지원은 없다)
- sender의 전송량을 제한

  LastbyteSend – LastbyteAcked
  ≤ CongWin (TCP 수신buffer가 충분히 큰경우 : 혼잡제어만 필요)
  [≤ min{congWin, RcvWindow}]
  (혼잡제어) (흐름제어)
- 송신률

  rate = CongWin / RTT (byte/sec)
- CongWin는 network 혼잡도에 따라 동적으로 변한다.

Sender가 congestion에 대한 반응

- 손실이 발생
  - timeout 또는 3번의 중복 ACK
  - 손실에 의한 재전송이 발생
- TCP sender는 손실이 발생하면 CongWin을 줄인다.

혼잡제어 알고리즘

- AIMD
- slow start
- timeout event에 대한 반응

TCP AIMD

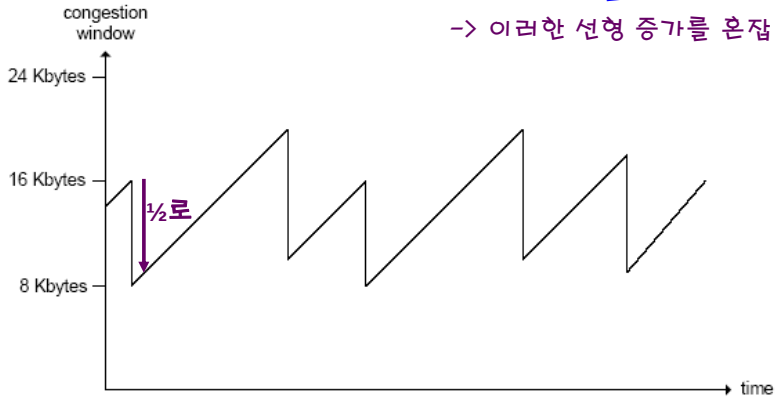
multiplicative decrease

loss event가 발생하면 CongWin의 size를 반으로 줄인다.
- 1MSS 이하로 떨어지지는 않는다.

additive increase

loss event가 없으면 매번 ACK를 확인 할때 각 RTT마다 congWin의 증가폭이 1MSS가 되도록 조금씩 confWin을 증가 시킨다.

-> 이러한 선형 증가를 혼잡 회피라고 한다.



Long-lived TCP connection

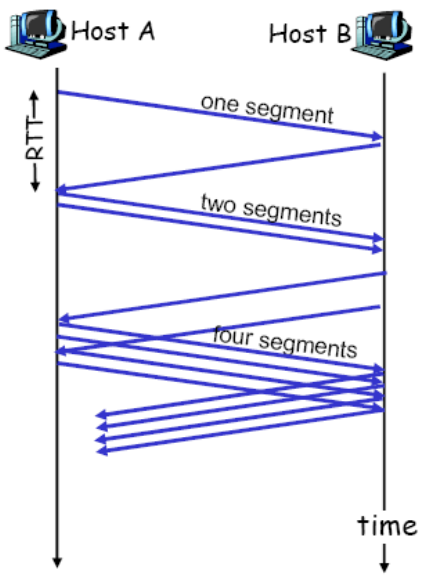
TCP Slow start

- TCP연결이 시작될때 CongWin의 크기는 1 MSS로 초기화 되고 초기 전송률은 MSS/RTT가 된다.
  - Ex
    MSS(500byte), RTT(200ms)
    init rate = 20kbps
    500byte/200ms
    = 4,000bit/0.2s = 20,000bps
    =20kbps
- 사용가능한 대역폭은 MSS/RTT보다 매우 크다
  - 선형비률로 증가시키면 전송률이 적당한 수준에 오를때 까지 긴 지연이 초래된다.
- 첫번째 loss가 발생할 때 까지 각 ack를 확인할 때 마다 congWin을 1MSS만큼 증가 시켜 각 RTT마다 congWin이 두배가 되도록 한다.

congWin의 증가치
AIMD : 각 RTT마다 1MSS
SS : 각 ACK마다 1MSS
각 RTT마다 두배

TCP Slow start

- 접속이 되면 loss event가 발생할 때 까지 지수적으로 속도를 증가 시킨다.
  - 각 RTT마다 CongWin이 두배
  - 모든 ACK가 수신될때마다 CongWin을 증가 시킨다.
- 초기의 속도는 매우 느리지만 지수적으로 빠르게 증가하는 모델이다.



Refinement

◈ 세번의 중복 ACK수신
- ❍ CongWin은 반으로 줄어든다.
- ❍ Congwin은 선형적으로 증가한다.
- ❍ AIMD

◈ 그러나 timeout이 발생하면
- ❍ CongWin을 1MSS로 줄인다.
- ❍ 지수적으로 CongWin을 증가시킨다.
- ❍ threshold에 도달하면 다시 선형적으로 증가한다.

  : 혼잡 회피에 들어간다.

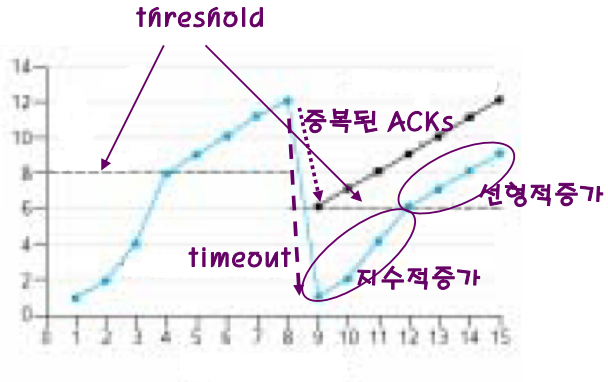
◈ threshold : CongWin의 절반

개념
- 세번 중복된 ACKs는 네트워크가 거의 대역폭 한계에 도달했다고 판단
- timeout는 삼중 ACK보다 심각하게 처리한다.

Refinement

Q :언제 지수적인 증가가 일어나는가?

A : timeout에 의해 CongWin이 1로 줄어든 이후에 threshold까지



구현

◈ threshold는 가변적이다.

◈ Loss event가 일어나면 threshold는 congWin의 1/2로 설정된다.

TCP 혼잡제어 요약

- CongWin은 threshold에 도달 할때 까지 slow-start방식으로 빠르게 증가한다.
- CongWin이 threshold에 도달하면 혼잡회피를 위해 선형적으로 천천히 증가한다.
- 3번 중복된 ACK가 감지되면 CongWin은 threshold 까지 줄어들고 선형으로 증가한다.
- Timeout이 발생하면 CongWin은 1 MSS로 줄었다가 다시 threshold에 도달 할 때 까지 빠르게 증가한다.