2장. 데이터 링크 네트워크
(Data Link Networks)

2장 데이터 링크 네트워크 (Data Link Networks)

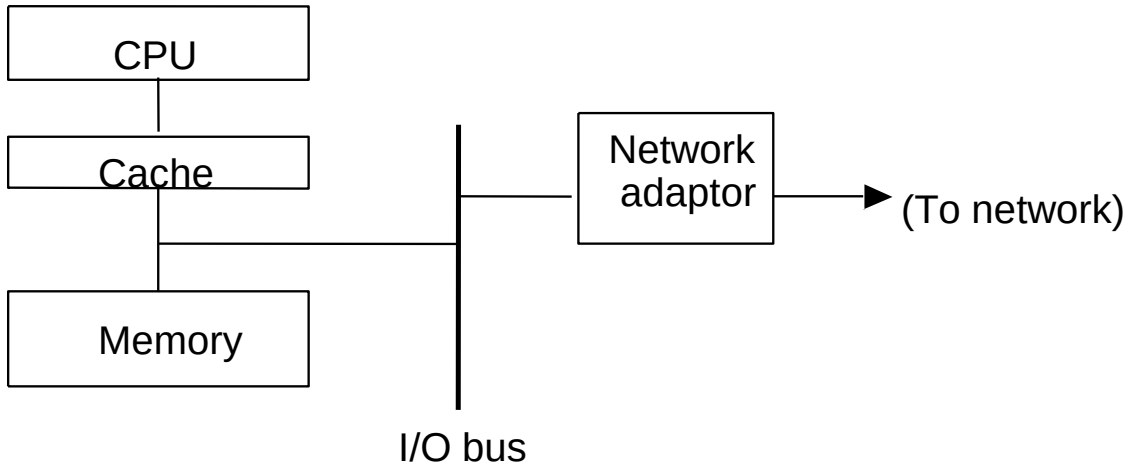
- ☑ 점대점(Point-To-Point) 링크
  - ☐ 하드웨어 구성요소
  - ☐ 인코딩
  - ☐ 프레이밍
  - ☐ 오류검출
- ☐ 신뢰성있는 전송
- ☐ 이더넷 / FDDI
- ☐ 네트워크 어댑터

데이터 링크 계층

- OSI 7계층의 1,2계층은 무엇을 담당?
  - 하나의 링크로 연결된 노드 사이의 비트묶음 (**프레임**) 교환
- 하나의 링크로 연결된 두 노드
  - **점대점 연결** 네트워크: 가장 간단한 네트워크
  - 일반적 네트워크 구성의 기본 block
- 노드란? **노드의 실체**는?
- 링크란? **링크의 실체**는?
- 점대점 연결에서의 통신의 실체
  - 비트 교환: **신호 인코딩 / 디지털 전송**
  - 비트 묶음 교환: **프레이밍 (frame)**
  - **오류 검출/복구**

하드웨어 구성요소 : 노드(Nodes)

- (단말/호스트 , 스위치/라우터)
- 범용 (프로그래밍할 수 있는) 컴퓨터로 구성된다고 가정.
  - 예) PC
- 때때로 특수한 목적의 하드웨어로 대체되기도 한다.

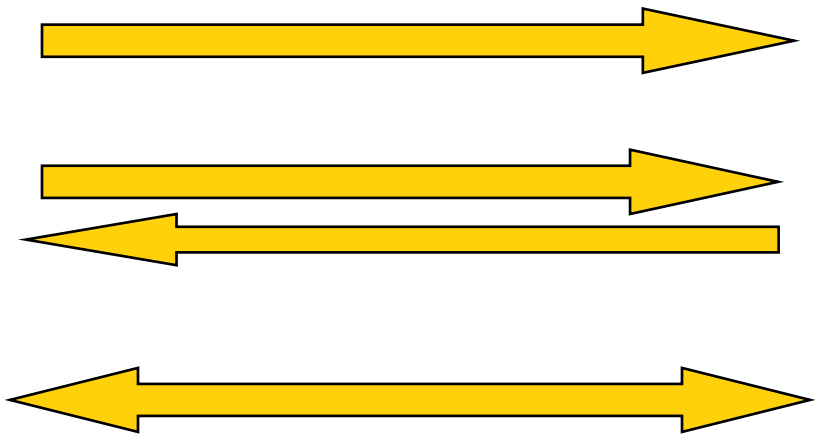


- 유한 메모리(제한된 버퍼공간을 의미)
- 네트워크 어댑터 (or NIC:Network Interface Card)를 통해서 네트워크에 연결
- 프로세서는 빠르고, 메모리는 느림.

링크 (Link)

- 데이터(신호) 전달을 위한 물리적 매체, 예) 케이블, 공기
- Wired vs Wireless

- 전송 모드
  - Simplex
  - 반이중(Half-duplex)
  - 전이중(Full-duplex)

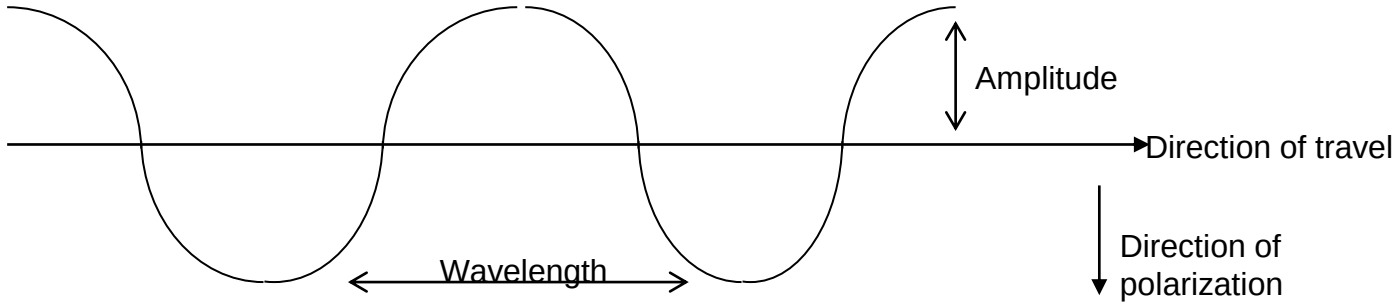


- 링크는 논리적 통로
  - 하나의 케이블에 여러 링크: 예) ADSL (전화선을 이용한 인터넷링크)
  - (당분간은 링크는 하나의 케이블)

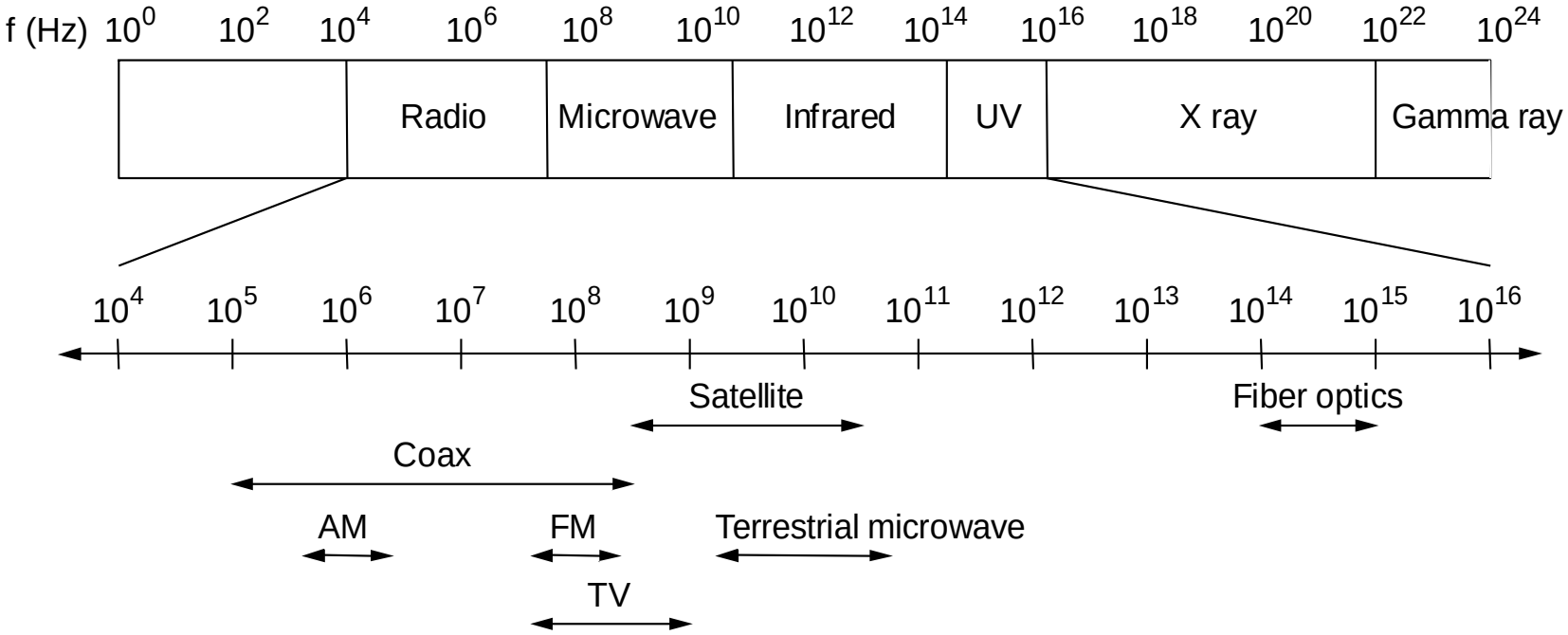
모듈레이션: 데이터의 신호화

- 데이터를 링크, 즉, 물리적 매체를 통해서 전달하기 위해서
- 인코딩/모듈레이션(Modulation) : 데이터 → 신호
- 수신쪽에서는 반대 작업 (Demodulation) : 신호 → 데이터
- 모뎀 (Modem) ?

- 신호의 종류: 전자기파 스펙트럼
  - 주파수(Hz), 파장(wave length)



전자기 스펙트럼과 매체 특성



- 저주파일수록 전송 특성이 좋다. (장애물 잘 통과)
- 저주파는 고속의 데이터 전송에 한계.
  - 통신속도 (대역폭)와 비트 폭(length)) 관계 (뒤에서 다시 설명)
- 통신의 발전: 저주파 → 고주파

사용 가능한 유선 링크의 종류

- 자신이 직접 선을 설치하는 경우

Category 5 twisted pair	10-100Mbps. 100m
50-ohm coax (ThinNet)	10-100Mbps. 200m
75-ohm coax (ThickNet)	10-100Mbps. 500m
Multimode fiber	100Mbps. 2km
Single-mode fiber	100-2400Mbps. 40km

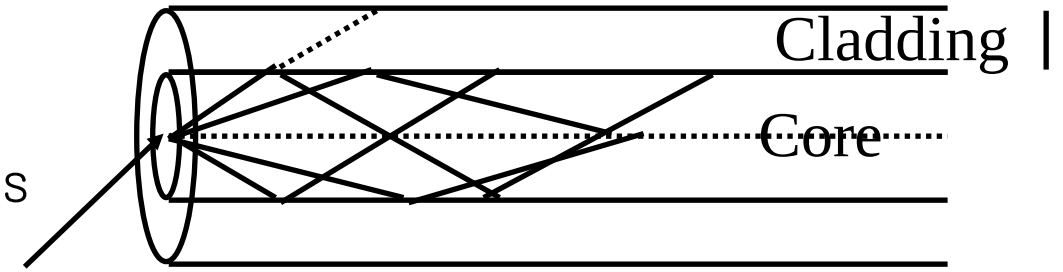
- 전화 회사로부터 선을 임대하는 경우

Service to ask for	Bandwidth you get
ISDN	64 Kbps
T1	1.544 Mbps
T3	44.736 Mbps
STS-1	51.840 Mbps
STS-3	155.250 Mbps
STS-12	622.080 Mbps
STS-24	1.244160 Gbps
STS-48	2.488320 Gbps

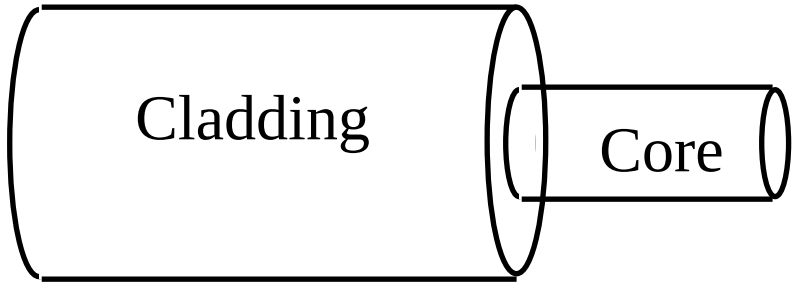
참조: 링크

광케이블: Optical Fiber

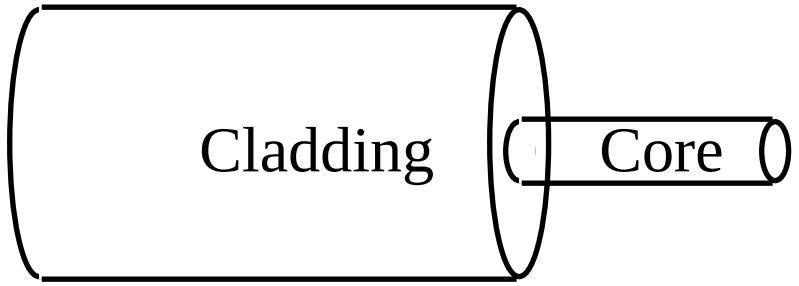
- Index of reflection
  =Speed in Vacuum/
  Speed in medium Modes



- Multimode

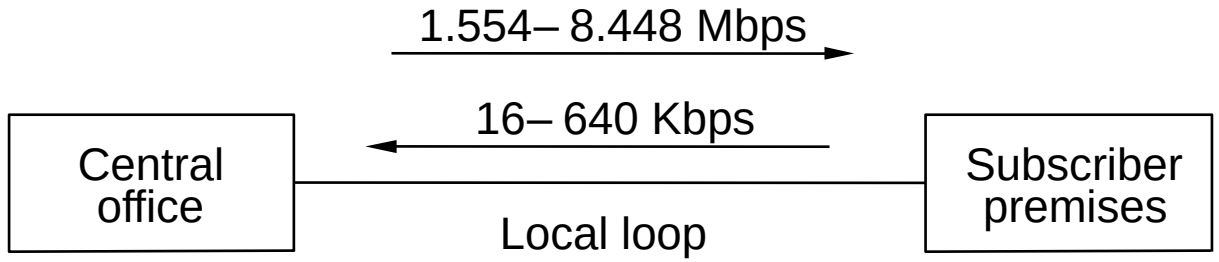


- Single Mode

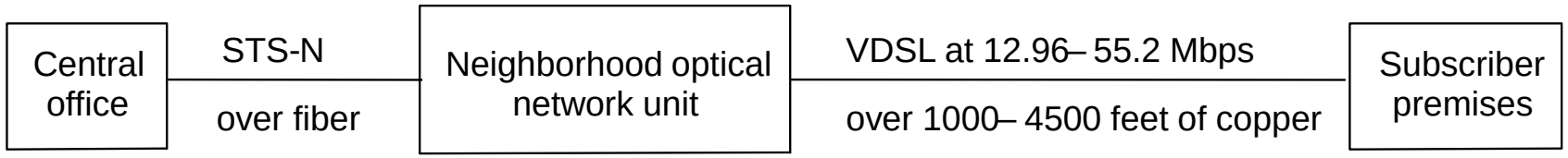


가입자 선로 (Last-Mile Links)

- 집(사용자)와 인터넷 공급자 사이를 마지막으로 연결하는 링크
- 사용자가 선택해서 사용하는 링크이므로 중요!
- 과거: 모뎀을 통한 음성 전화 링크
- xDSL(Digital Subscriber Loop) : 음성과 data를 FDM 방식으로 동시에
  - ADSL

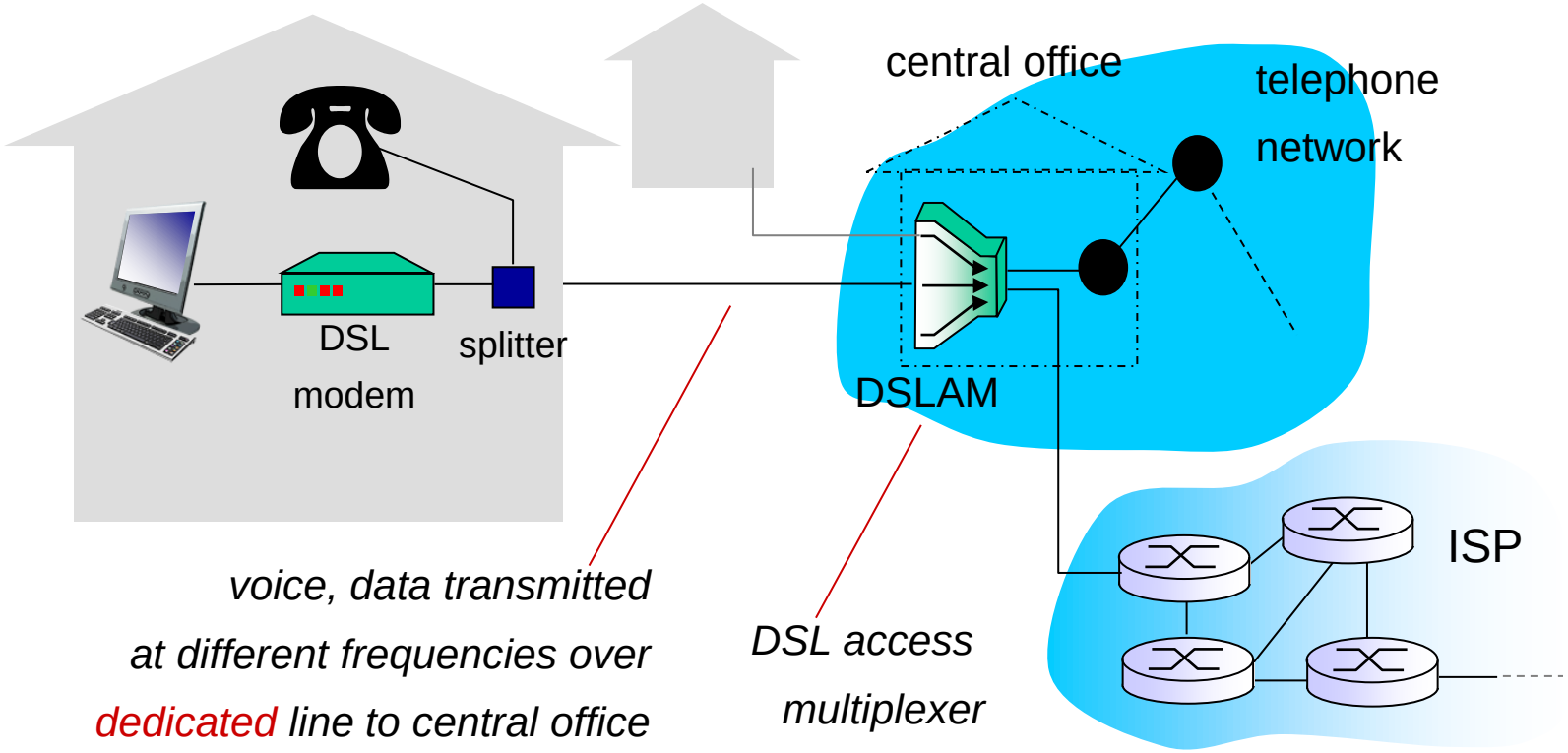


  - VDSL



- Cable Modem
  - 6M – 100 M; asymmetric
  - shared bandwidth

Digital subscriber line (DSL)



- use *existing* telephone line to central office DSLAM
  - data over DSL phone line goes to Internet
  - voice over DSL phone line goes to telephone net

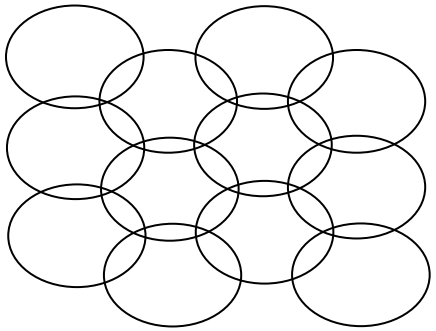
무선 링크 (Wireless Links): 일반

- +: 고정된 링크가 없음
  - 이동성 지원
  - 즉시 사용 가능

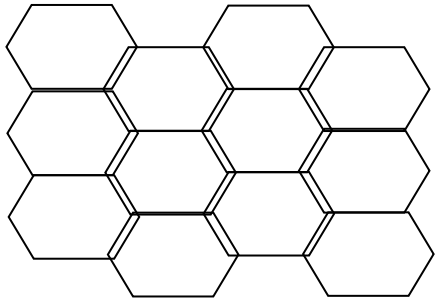
- –: 공중으로 퍼져나감
  - 고주파 vs. 저주파
  - 인접한 링크 사이에 간섭이 일어날 수 있음
    - 전파 사용에 규제가 필요 – 라이센스 제도
  - multipath problem

이동통신(Cellular Networks)

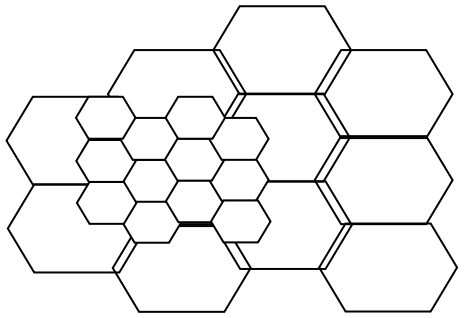
- 기지국 ↔ 단말기
  - 셀: 하나의 기지국이 관할하는 지역
  - 핸드오프 (hand-off) 문제
- 기술 발전
  - AMPS ⇒ PCS(GSM/CDMA) ⇒ W-CDMA ⇒ 4세대 이동통신 ⇒ 5G



Overlapping circular cells



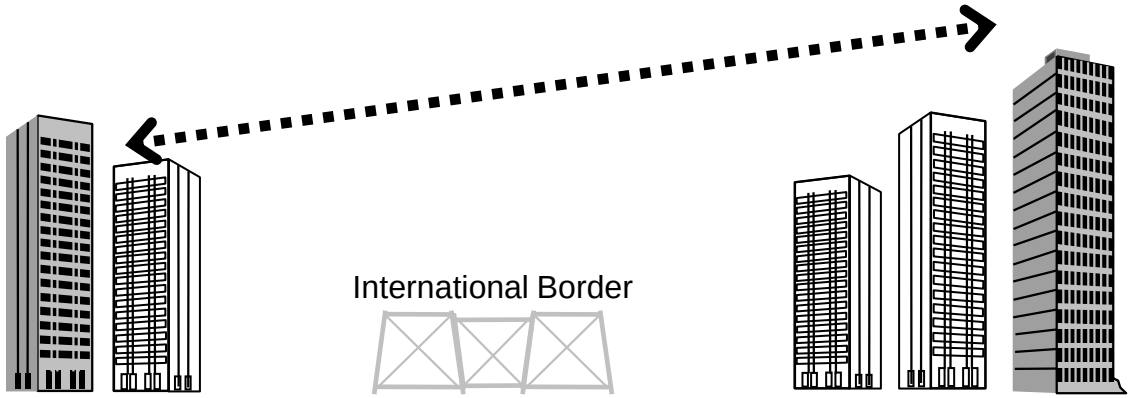
Idealised hexagonal network



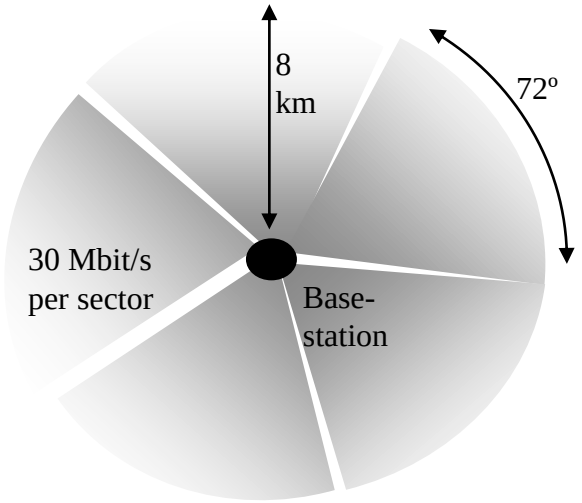
Microcells within a network

고정 무선통신(Wireless Fixed links)

- 무선 고속 전용 링크

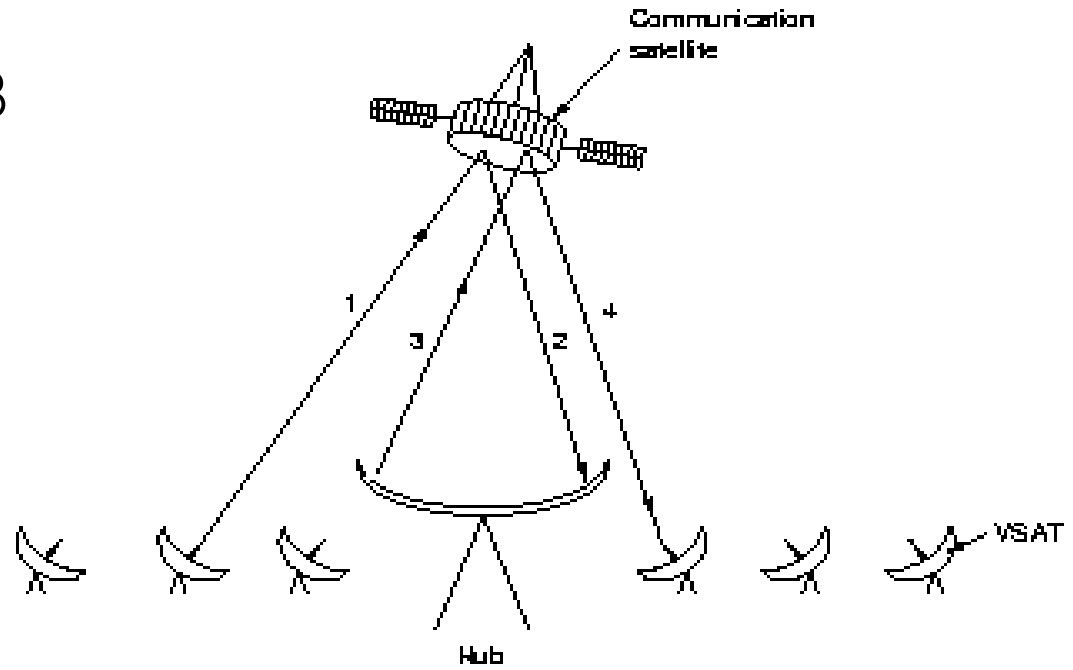


- 무선 가입자망



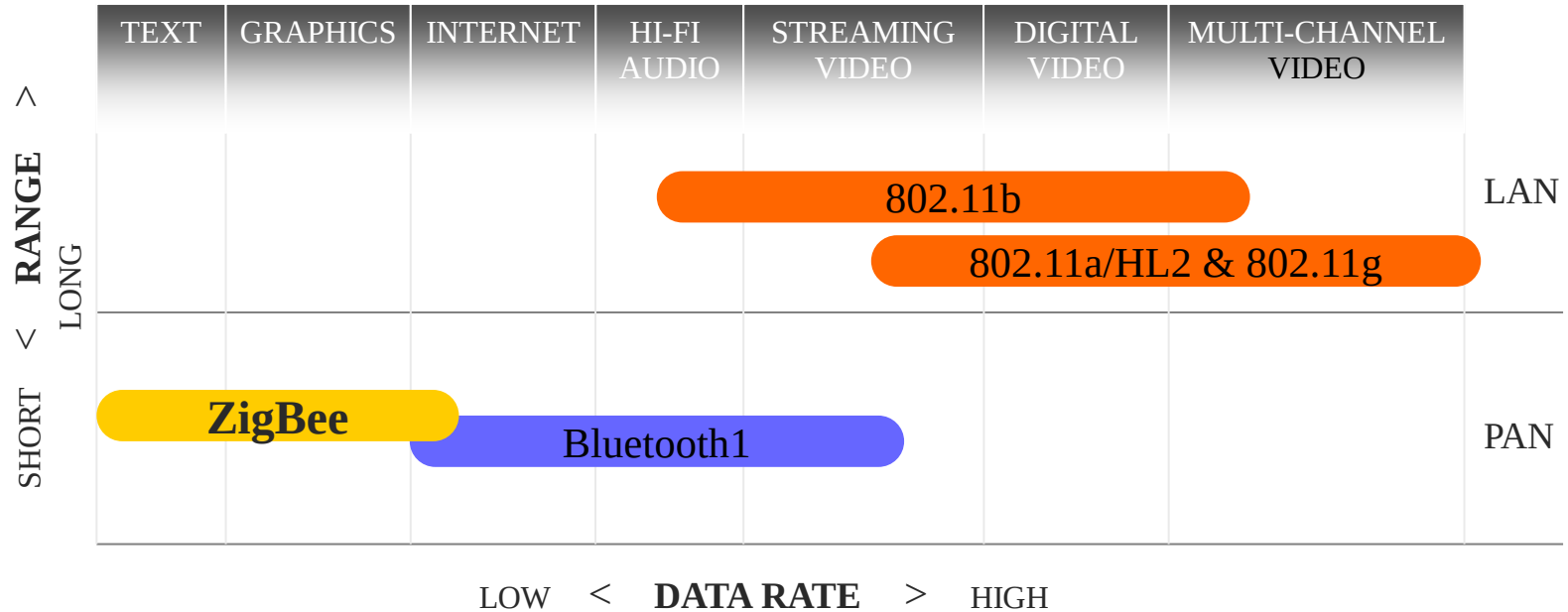
위성통신(Satellite system)

- 정지궤도 방송/전화
- 정지궤도 데이터 (VSAT)
- 저궤도 전화
  - Iridium
- 위성 방송 / DMB



단거리 무선통신(Short Range)

- Public (license-free) band 이용
- 적외선 통신
- 무선LAN - IEEE 802.11
- Bluetooth
- ZigBee / IEEE 802.15.4

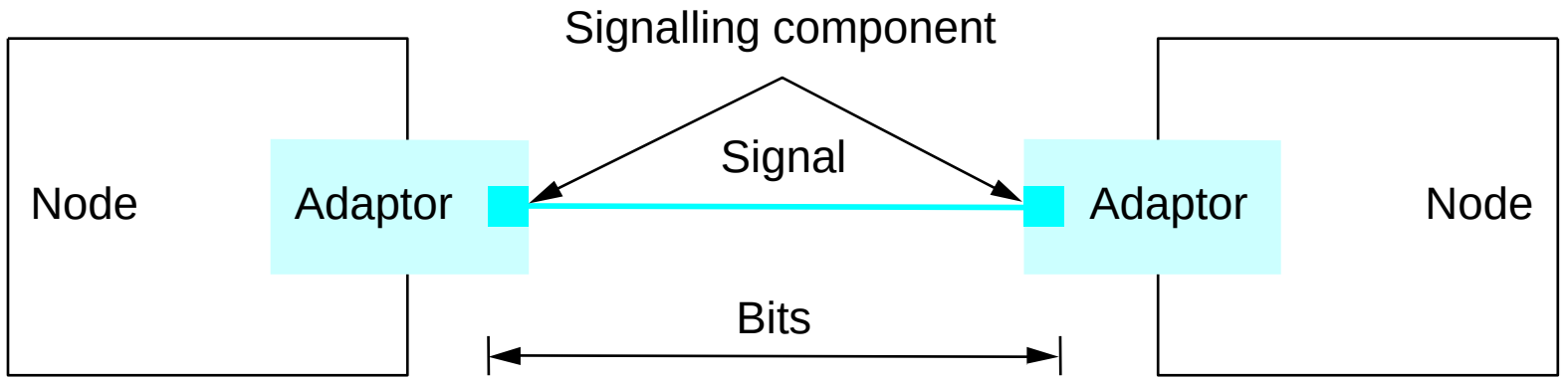


2장 데이터 링크 네트워크 (Data Link Networks)

- ☐ 점대점(Point-To-Point) 링크
  - ☐ 하드웨어 구성요소
  - ☑ 인코딩
  - ☐ 프레이밍
  - ☐ 오류검출
- ☐ 신뢰성있는 전송
- ☐ 이더넷 / FDDI
- ☐ 네트워크 어댑터

인코딩(Encoding) : 개요

- 신호(Signal)는 물리적 매체를 통해 전달된다.
  - 디지탈 신호
  - 아날로그 신호
- 데이터는 디지탈 데이터만을 취급.
  - 아날로그 데이터는 디지탈 데이터를 변환
- 문제: 발신지에서 목적지로 보내려는 이진 데이터를 전달될 수 있는 신호로 인코드해야 함.
- 보다 일반적인 용어는 변조 (Modulation)

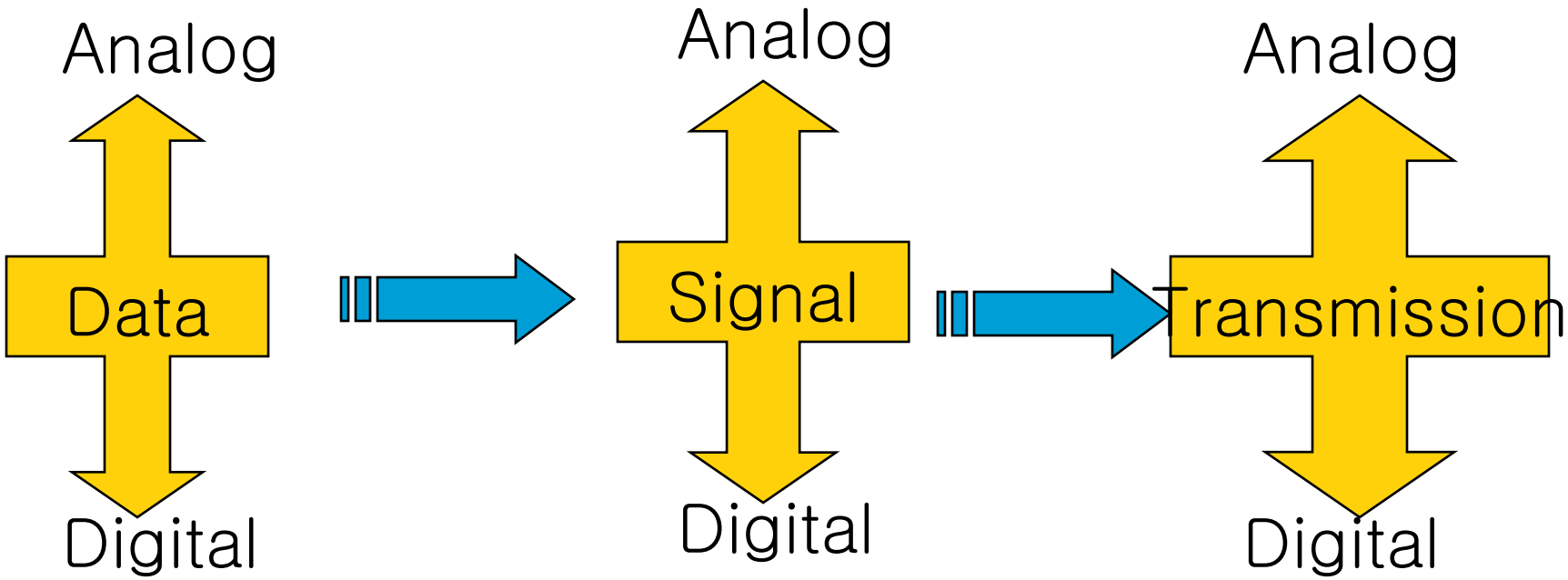


디지털 전송 (Transmission)

- 신호 중계 방법
- 아날로그 전송
  - 신호를 단순 증폭
  - 즉, 앰프(Amplifier) 사용
- 디지털 전송
  - 신호에서 데이터를 복원하여 다시 신호화
    - 디지털 데이터를 담은 신호만 사용 가능
  - 즉, 리피터(repeater) 사용
- 거의 모든 전송 방법이 디지털 전송 사용
  - noise 제거
  - 기기 비용
  - 부가 기능 추가 가능

참조: 인코딩

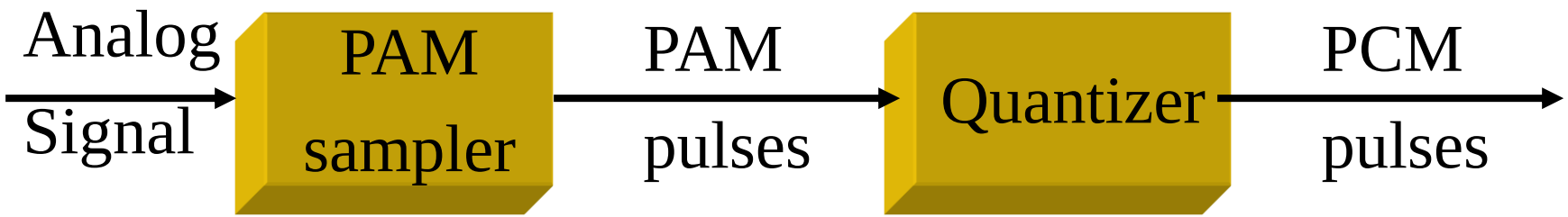
데이터 전송 (Transmission)



PCM (Pulse Code Modulation)

- Sampling Theorem:
  Sampling rate ≈ 2 × Highest signal frequency
- 4 kHz voice = 8 kHz sampling rate
- Represent samples as pulses (PAM)
- Quantize the samples (PCM)
- 8 k samples/sec × 8 bits/sample = 64 kbps

Pulse Code Modulation



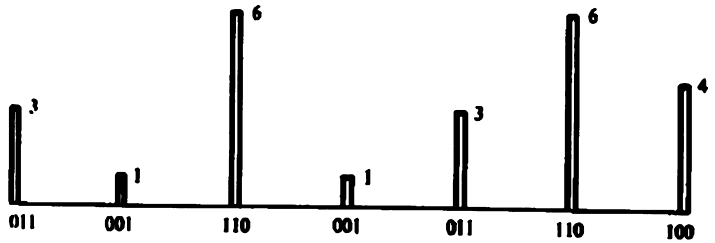
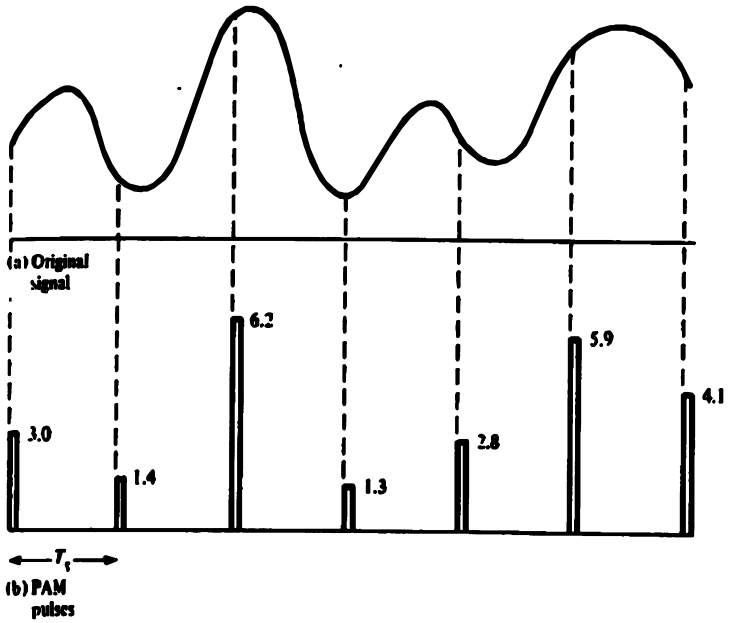
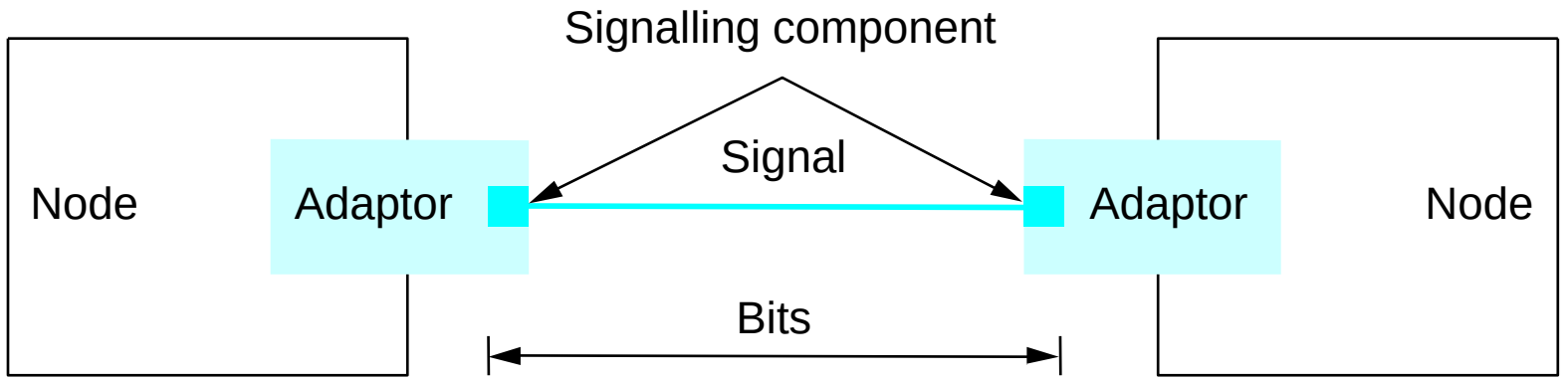


FIGURE 3.11. Pulse-code modulation.



인코딩(Encoding) : 개요

- 신호(Signal)는 물리적 매체를 통해 전달된다.
  - 디지탈 신호
  - 아날로그 신호
- 데이터는 디지탈 데이터만을 취급.
  - 아날로그 데이터는 디지탈 데이터를 변환
- 문제: 발신지에서 목적지로 보내려는 이진 데이터를 전달될 수 있는 신호로 인코드해야 함.
- 보다 일반적인 용어는 변조 (Modulation)



참조: 변조 (D-data → A-Signal)

변조: Amplitude Modulation

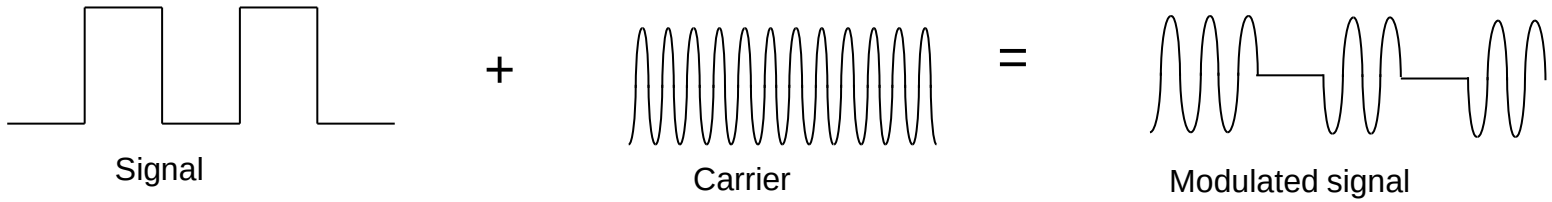


Figure 2.6
Amplitude modulation

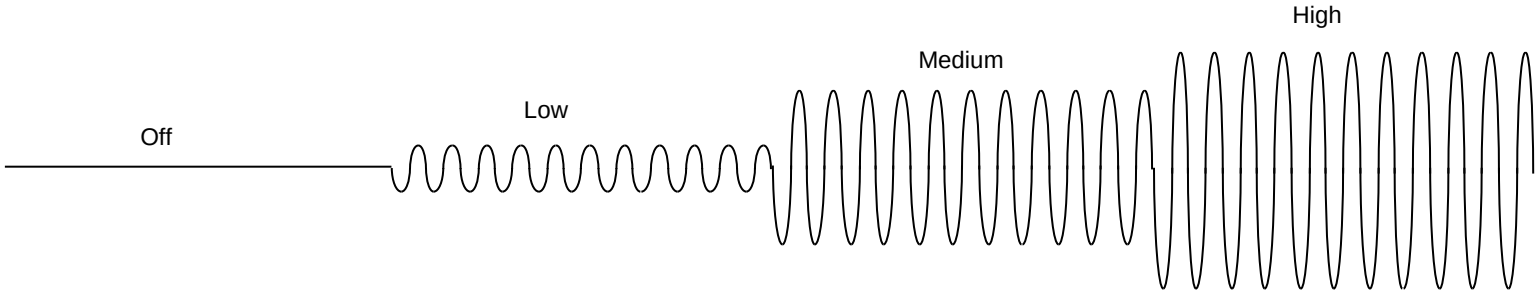


Figure 2.7
Quadrature amplitude modulation (QAM)

주파수 변조: Freq. Modulation

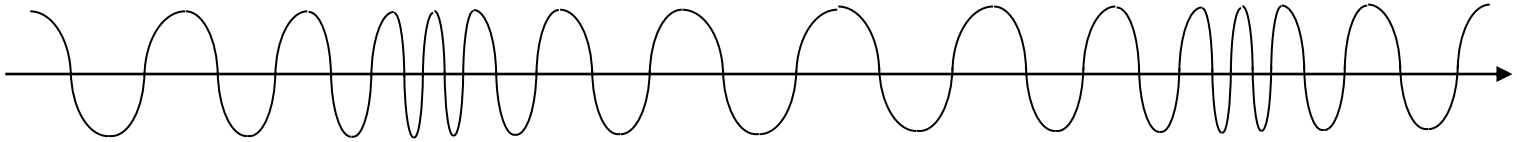


Figure 2.8
Frequency modulated carrier wave

위상 변조: Phase Modulation

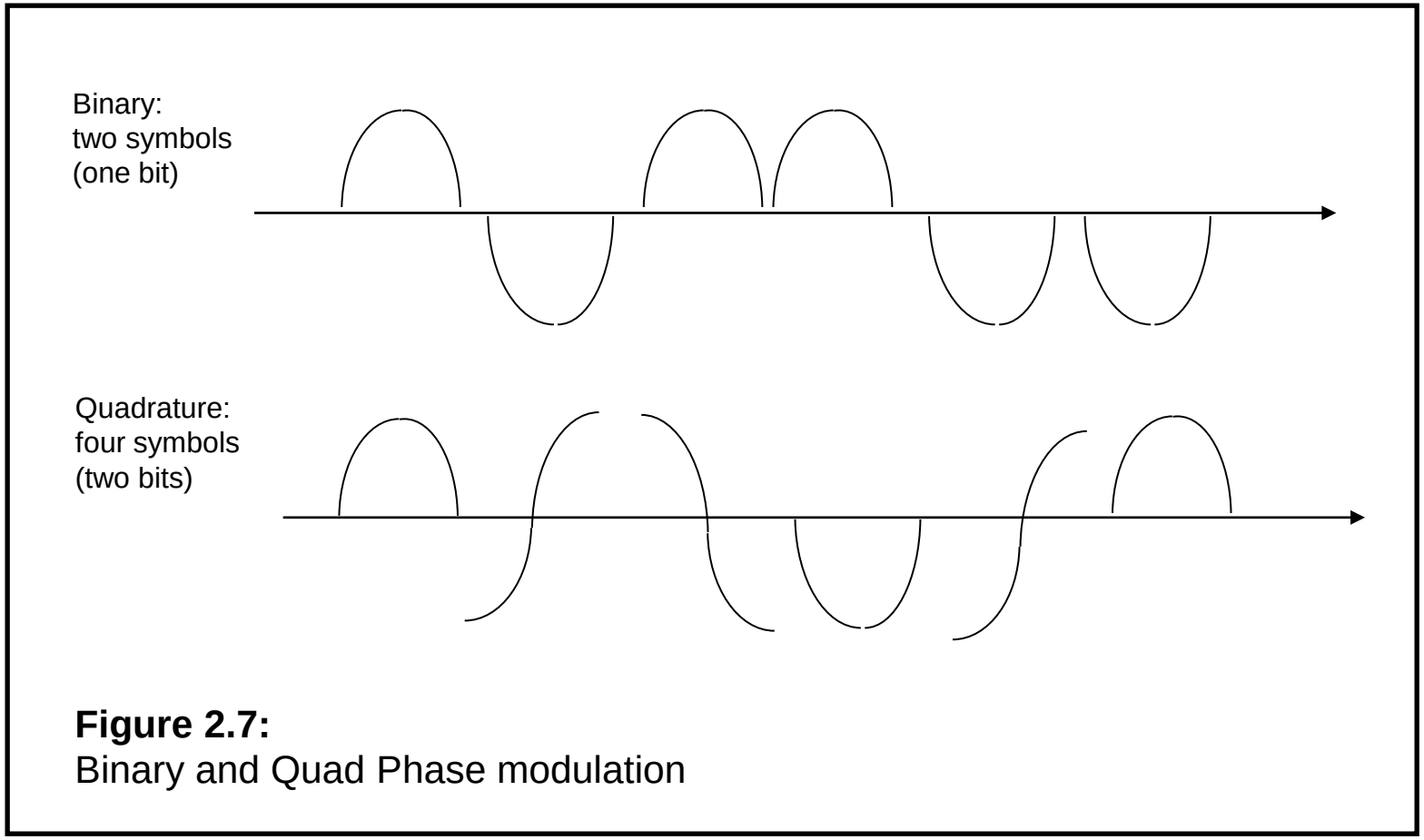


Figure 2.7:
Binary and Quad Phase modulation

이동통신의 속도가 2배 ↑

- 무선랜 속도 등 모든 통신 분야에서 마찬가지
- 어떤 방법으로?

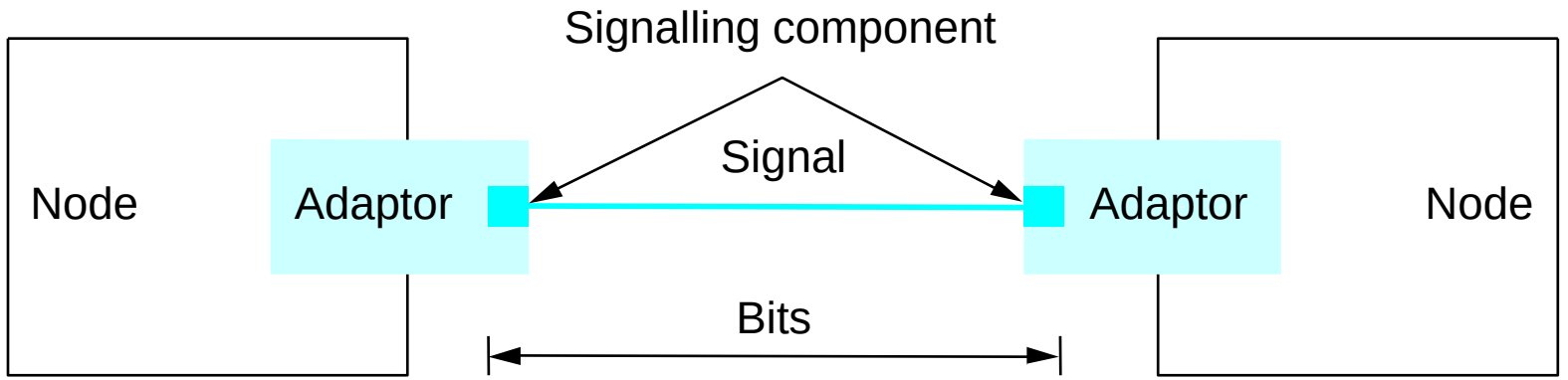
참조: 정리

디지털 전송 (Transmission)

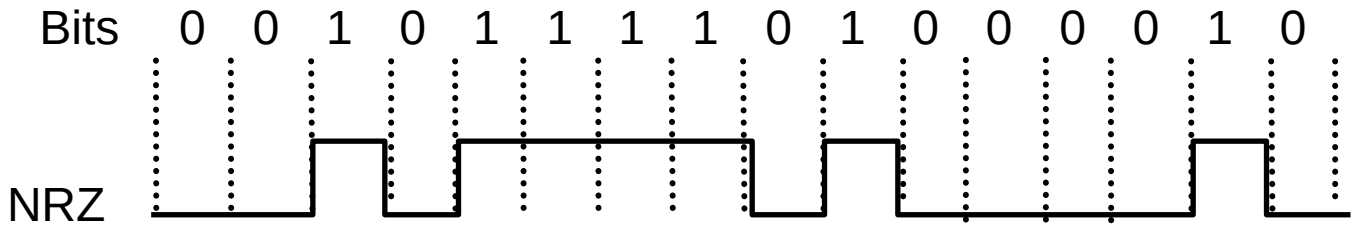
- A-Data, D-Data
- A-Transmission, D-Transmission
- Modulation : Data ⇒ A-Signal
- 실제 사용 예?
  - 휴대폰?
  - 집 인터넷
  - 집(유선) 전화?
  - 아날로그 방송, AM/FM 라디오?
  - 디지털 방송 ?
- 마지막 주제: Data ⇒ D-Signal, Encoding!

인코딩(Encoding) : 개요

- 신호(Signal)는 물리적 매체를 통해 전달된다.
  - 디지탈 신호
  - 아날로그 신호
- 데이터는 디지탈 데이터만을 취급.
  - 아날로그 데이터는 디지탈 데이터를 변환
- 문제: 발신지에서 목적지로 보내려는 이진 데이터를 전달될 수 있는 신호로 인코드해야 함.
- 보다 일반적인 용어는 변조 (Modulation)



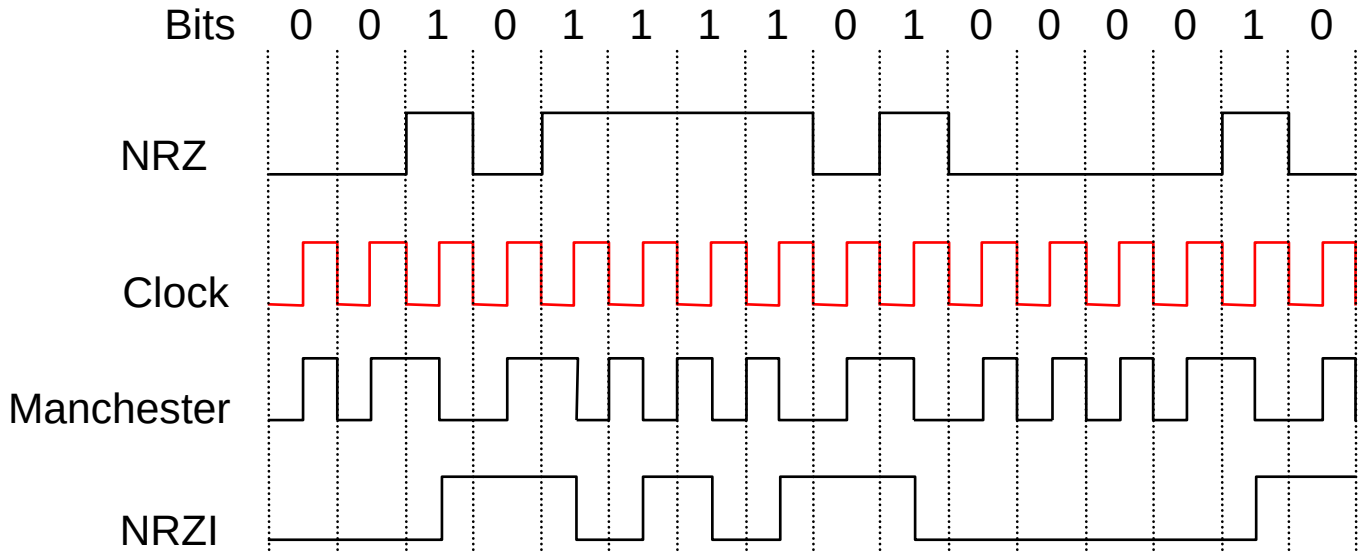
Non-Return to Zero(NRZ)



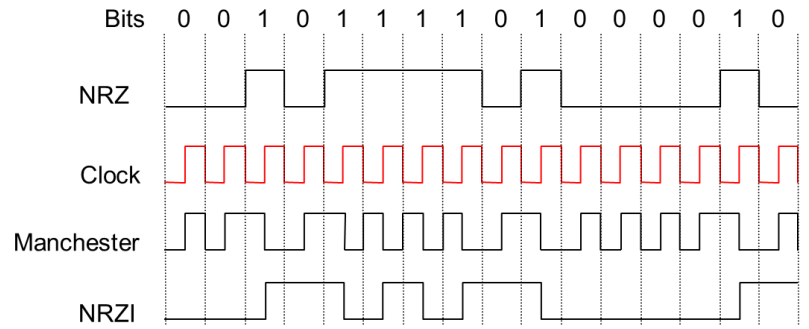
- 노드 내부의 데이터 표현과 일치 (즉, 별도의 인코딩 필요 없음)
- 문제점: 1 또는 0이 연속되는 경우
  - Low signal(0)의 경우 수신자는 신호가 없는 것으로 오해할 수 있음
  - High signal(1)의 경우 전류가 계속 흐르게 되고, 기저 전압의 혼돈을 야기
  - **클럭(clock) 복구**가 불가능
    - 송신자와 수신자의 클럭이 맞지 않으면 잘못된 비트 인식
    - 수신자가 송신자의 클럭에 자신의 클럭을 맞추는 작업

NRZI and Manchester

- Non-return to Zero Inverted (NRZI)
  - 1을 인코드할 경우 현재의 신호로부터 중앙 지점에서 전이(mid-transition)를 하고, 0을 인코드할 경우 현재의 신호 상태를 유지함; 연속되는 1의 문제를 해결.
- Manchester
  - 0 : up (↑) transition, 1: down(↓) transition
  - NRZ방식으로 인코드된 데이터와 클럭을 배타적 논리합을(XOR) 시켜서 바꾼다; 50 % 효율을 갖는 문제점.



Mid-transition의 의미 (Manchester 코드 예)



- 송신 쪽에서, $M_i = Data_i \oplus Clock_i^S$
  - 참고. X ⊕ 0 = X, X ⊕ 1 = X', X ⊕ X = 0
- 수신된 M'_i 에서 0/1을, 즉, $Data_i$를 인식한 후,
  - 전송 중 오류는 없다, 즉, $M'_i == M_i$ 가정
  - $(M'_i \oplus Data_i)$를 하면,
  - $(M'_i \oplus Data_i) = (Data_i \oplus Clock_i^S) \oplus Data_i = Clock_i^S$
  - 즉, 수신자가 송신자의 클락을 꺼낼 수 있으며, 따라서 동기화를 할 수 있다.
- 결론: Manchester 코드는 Data에 Clock을 얹어서 동시에 보내는 것.
- 접근 방식의 의미? (넓게 보면, in-band signaling)

4B/5B

- 아이디어
  - 데이터를 매 4bits마다 5-bit코드로 인코드한다, 이 5-bit의 코드는 앞에는 1개, 뒤에는 2개까지의 0이 오도록 제한해서 선택된 코드이다. (따라서, 0이 4개 이상 연속되는 나올 수 없다.)
  - 5-bit코드는 NRZI 인코딩을 이용해서 전송된다.
  - 효율 80%를 달성.

4-bit Data	5-bit Code
0000	11110
0001	01001
0010	10100
0011	10101
0100	01010
0101	01011
0110	01110
0111	01111

4-bit Data	5-bit Code
1000	10010
1001	10011
1010	10110
1011	10111
1100	11010
1101	11011
1110	11100
1111	11101

비트(신호)의 실체 정리

실제 응용	개략적 신호 형태 그림	Data type? (A? D?)	링크 매체? (유선? 무선?)
AM 라디오 방송 (방송국 → 라디오)			
휴대폰 메시지 응용 (휴대폰 ↔ 기지국)			
휴대폰 통화 (휴대폰 ↔ 기지국)			
블루투스 기기 통신 (휴대폰 ↔ 이어폰)			
PC실 PC간 파일 교환 (PC ↔ 유선 케이블)			
노트북에서 웹 사용 (PC ↔ 무선 AP)			

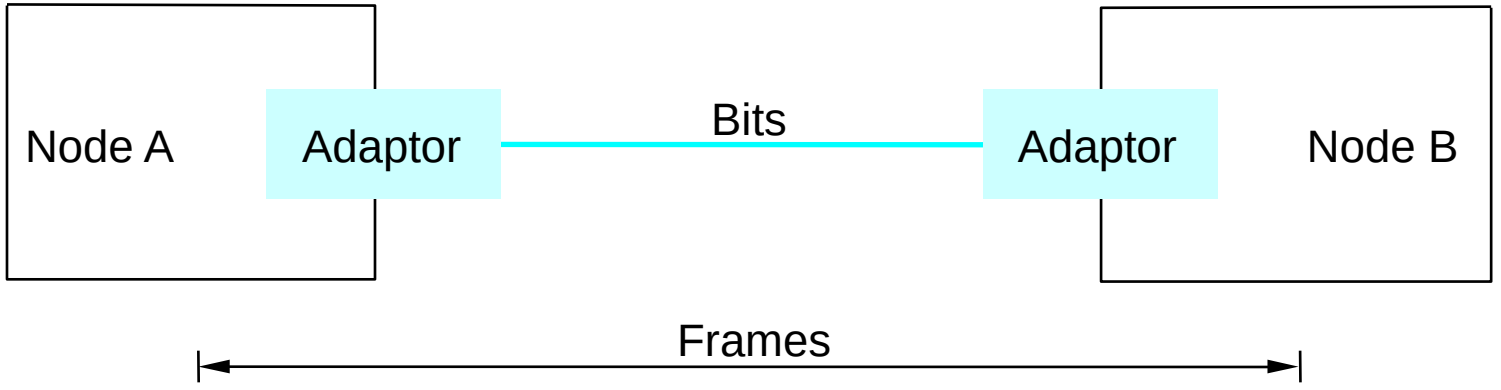
2장 데이터 링크 네트워크 (Data Link Networks)

- ☐ 점대점(Point-To-Point) 링크
  - ☐ 하드웨어 구성요소
  - ☐ 인코딩
  - ☑ 프레이밍
  - ☐ 오류검출
- ☐ 신뢰성있는 전송
- ☐ 이더넷 / FDDI
- ☐ 네트워크 어댑터

프레이밍(Framing) : 개요

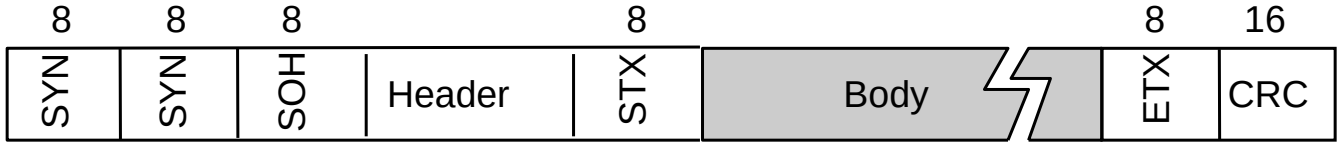
- 데이터를 끝없이 보낼 수는 없음. (특히, 패킷네트워크에서는)
- 문제
  - 비트(bit)들의 연속을 하나의 묶음(프레임:frame)으로 자르는 것
  - 수신 쪽이 프레임을 인식할 수 있도록 봉투를 씌워서 묶는 것.
  - 프레임의 처음과 끝 인식
    - 기타 추가 정보 기입 (추후에)

- 전형적으로 네트워크 어댑터에서 구현됨
- 어댑터는 호스트 메모리로부터 프레임(데이터+헤더 일부분)을 넣고 가져옴

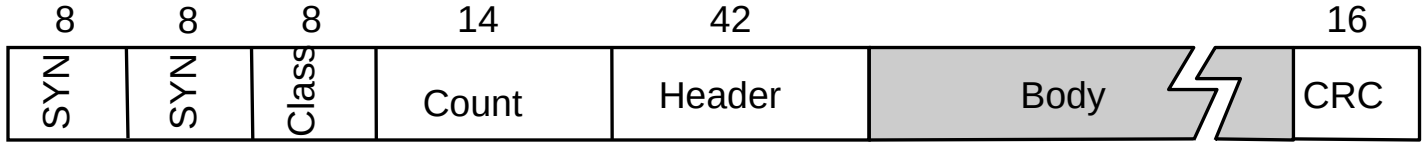


바이트 중심 프로토콜 (Byte-Oriented Protocols)

- 보초 방법(Sentinel Approach)
  - BISYNC



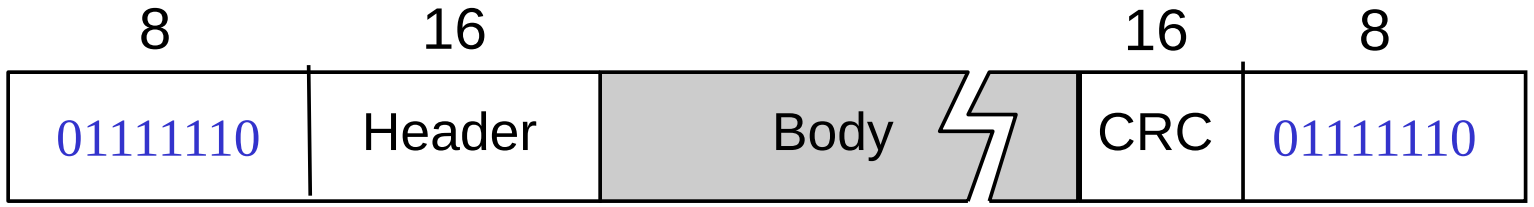
  - IMP-IMP, PPP
  - 문제점: 프레임의 데이터 부분에서 ETX 문자가 나올 경우
  - 해결: 확장 문자 (escaping character) 사용
    - BISYNC의 경우 DLE문자를 ETX문자를 앞에 부착
    - IMP-IMP의 경우에는 DLE문자 앞에 DLE문자를 부착
- 바이트 수 방법 (Byte Counting Approach) – DDCMP



  - 문제: 개수 (Count) 항이 잘못된 경우(framing error).
  - 해결: 순회중복검사(CRC)가 실패하므로 오류 검출

비트 중심 프로토콜 (Bit-Oriented Protocols)

- HDLC: High-Level Data Link Control (also SDLC and PPP)
  - 특별한 bit-sequence를 프레임의 앞과 뒤에 붙여 프레임을 구분: (01111110)



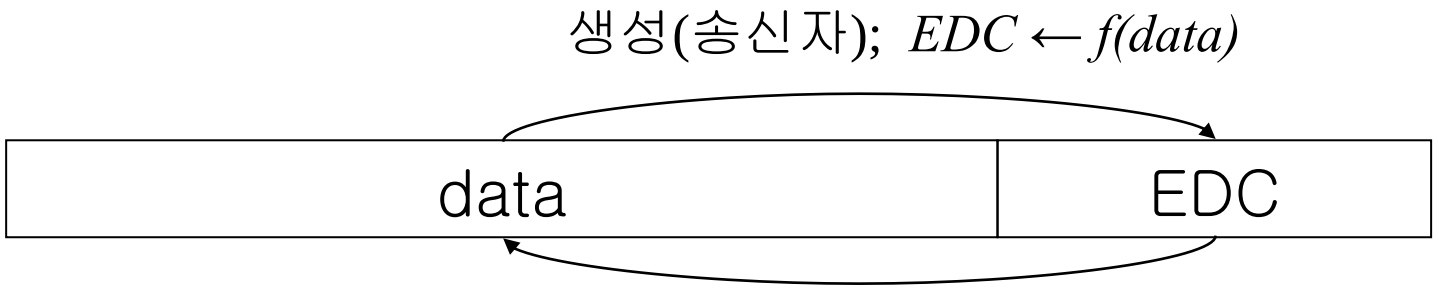
- 비트 삽입 (bit stuffing)
  - 송신자: 메세지의 중간에 연속되는 5개의 1이 나오면 0을 삽입함
  - 수신자: 1을 연속해서 5개 받았을 때, 다음 비트를 본다:
    - 다음 비트가 0이라면: 그 비트를 삭제함
    - 다음 비트가 10이라면: 프레임의 끝
    - 다음 비트가 11이라면: 오류

2장 데이터 링크 네트워크 (Data Link Networks)

- ☐ 점대점(Point-To-Point) 링크
  - ☐ 하드웨어 구성요소
  - ☐ 인코딩
  - ☐ 프레이밍
  - ☑ 오류검출
- ☐ 신뢰성있는 전송
- ☐ 이더넷 / FDDI
- ☐ 네트워크 어댑터

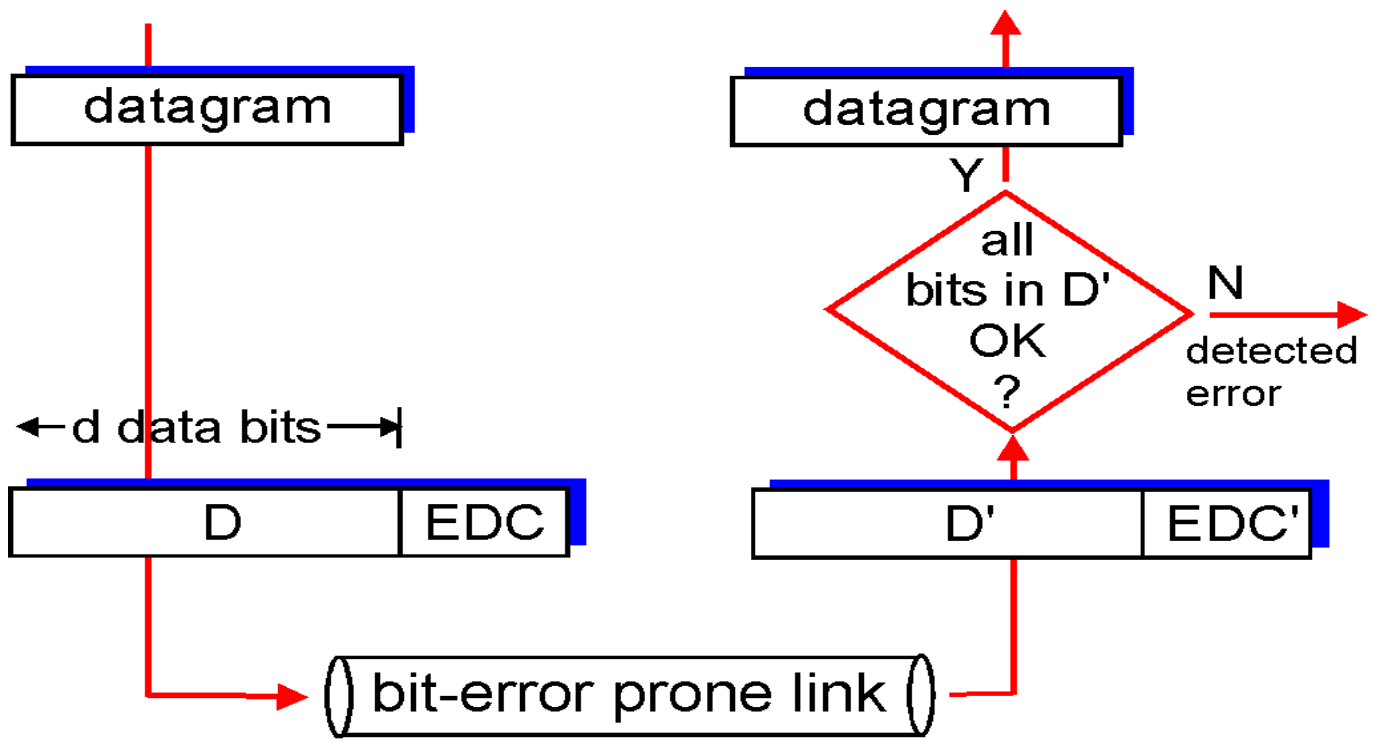
오류 검출 코드 (Error Detecting Code)

- 데이터 영역 안에 오류가 있는지를 없는지를 알아내는 부가 데이터
- 예: parity
- 크기 면에서, EDC << data ; 오류가 없는 경우 단순 부하이므로
- 효율 면에서, 오류 검출율이 높아야 함. (검출율 정의는 다음 페이지)
- 비용 면에서, *f()* 연산에 시간이 적게 소모되어야 함.

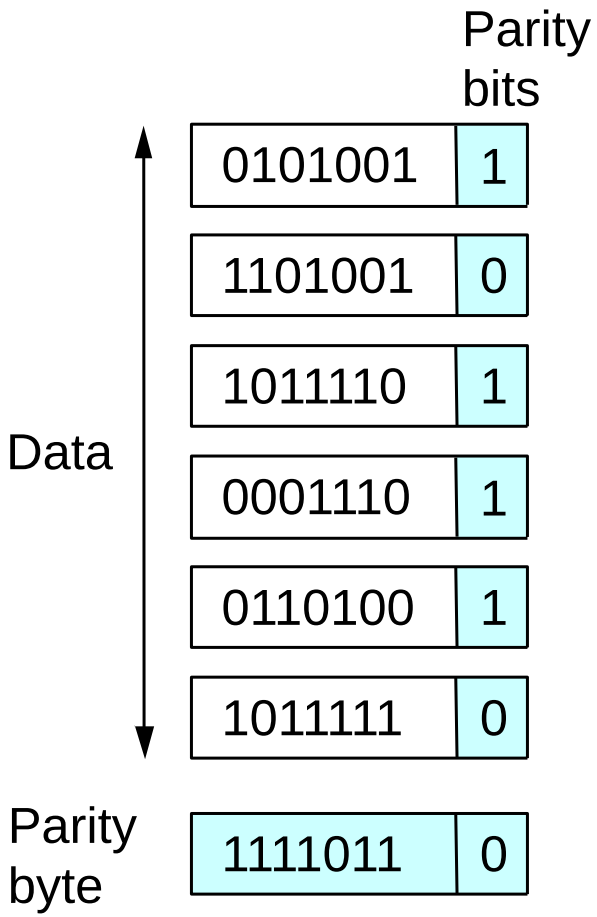


오류 검출율

- EDC가 완벽하게 오류를 검출할 수 있을까?
- 검출 실패 → 시스템 integrity 상실
- 매우 높은 오류 검출율 + 중복 오류 검사



2차원 패리티 (Two –Dimensional Parity)



인터넷 체크섬 알고리즘 (Internet Checksum Algorithm)

- 아이디어: 메시지를 16-bit의 정수의 연속으로 간주하고, 각 정수들을 16-bit 1의 보수 연산을 사용하여 모두 더한다. 그리고 그 결과의 1의 보수를 얻는다. 이 16-bit 숫자가 체크섬(checksum)이다.

```
u_short
cksum(u_short *buf, int count)
{
    register u_long sum = 0;
    while (count--) {
        sum += *buf++;
        if (sum & 0xFFFF0000) {
            /* carry occurred, so wrap around */
            sum &= 0xFFFF;
            sum++;
        }
    }
    return ~(sum & 0xFFFF);
}
```

순회 중복 검사 (Cyclic Redundancy Check: CRC)

- 더하기 보다 복잡한 나누기 사용 : 나머지를 오류 검출코드로 전송
- 젯수로 사용될 송수신 사이에 약속된 비트 패턴 : C
- 보낼 메시지 : M
- 오류 검출을 위해 추가되는 정보 : F (즉, EDC)

- 송신쪽
  - (M || F) % C = 0 이 되도록 F를 생성
    - 즉, (F에 나머지를 넣어서) 전체 프레임이 C로 나누어 떨어지도록 함.
  - (M || F) 를 전송
  - 예) C=1101, M= 10011010 이면
    - F = 101 를 생성 (10011010101 은 1101로 나누어 떨어짐)
    - 10011010101 전송

- 수신쪽
  - 수신된 메시지 전체를 C로 나누어서
    - 나누어 떨어지지 않으면 => 오류 발생
    - 나누어 떨어지면 => 오류 없는 것으로 **간주**

CRC의 개념적 이해

- **주의**: 실제는 10진수 연산 사용하지 않음. 다항식연산 (XOR 연산)

- 예) 송수신자가 약속하는 젯수 C = 11
- 송신자가 data 331을 보내려면,
  - 331과 함께 오류 검출용 숫자를 추가해서 전송.
  - 이 경우, 331x가 11로 나누어 떨어지도록 계산해서 1을 추가.
  - 즉, 3311을 전송
  - 일반적으로, 추가되는 자릿수는 (C의 자릿수 – 1)
- 수신자는 받은 비트 전체 (프레임)를 약속된 C로 나누어 봄
  - 이 경우, 전송 중 오류가 없다면, 수신된 3311을 11로 나누어 봄
    - 나머지가 0이면, 오류가 없다고 **간주**
    - 나머지가 0이 아니면 (예, 3310, 또는 2311), 오류 검출
  - 오류가 발생하더라도 검출에 실패할 수 있음. (예, 2211)

- 오류 검출율은 C의 값과 관계 있음. (예, C=10 으로 정한다면 …)

CRC의 성능

- 오류 검출율
  - 비트 패턴 C의 선택이 좌우
  - 수학적 분석에 의해 잘 잡으면, (분석 과정 슬라이드 참조)
  - 32비트 코드 사용하면, 1500 byte 이상의 데이터에 대해서도 99.99… 의 검출율
- 연산 속도
  - 전송 속도보다 늦으면 곤란
  - 가능하면 네트워크 카드 내에서 하드웨어로

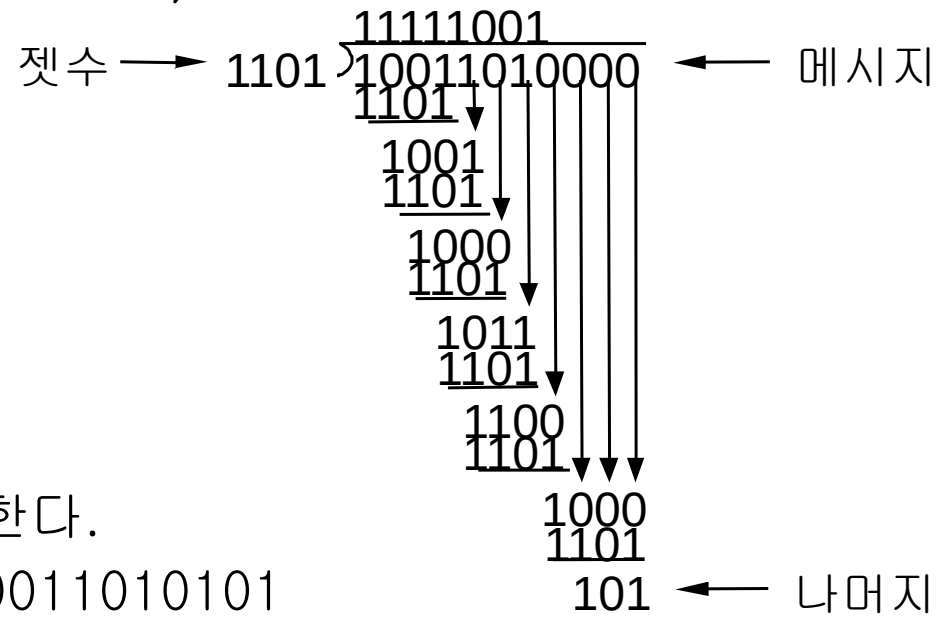
- 빠른 연산과 분석을 위해
  - 다항식연산 (즉, ⊕-연산)을 사용

- 표준화된 C 사용 → “CRC Code”

- 세부 사항은 옵션. 송신/수신 쪽에서의 CRC 동작 정리는 필수!

순회 중복 검사 : 송신자 세부 사항

- *n*-bit의 메시지는 n-1차 **다항식으로 표현**
  - 예) MSG=10011010은 $M(x) = x^7 + x^4 + x^3 + x^1$ 에 대응됨
- *(k+1)* 비트 젯수 패턴 C에 해당하는 *k*차 다항식 $C(x)$를 설정
  - 예) C=1101은 $C(x) = x^3 + x^2 + 1$
- $M(x)$에 x^k을 곱한다. (F가 들어갈 자리를 포함한다.)
  - $x^{10} + x^7 + x^6 + x^4$ (10011010000)
- $C(x)$ (1101)로 나눈다.



- 나머지를 추가한 $P(x)$를 전송한다.
  - 10011010000 – 101 = 10011010101
  - $P(x)$는 $C(x)$로 나누어 떨어진다.

세부사항 : 젯수 C 결정 및 검출률 분석

Error Pattern: E(x)

- 전송하는 프레임: P(x)
- 수신된 프레임: P'(x)
- 전송 중 발생한 오류를 프레임에 대응되는 형태로 나타내면: E(x)
- 관계

P(x)	0 1 1 0 … 0 … 1 … 0 1
P'(x)	0 1 1 0 … 1 … 0 … 0 1
E(x)	0 0 0 0 … 1 … 1 … 0 0

$$P(x) \oplus P'(x) = E(x), \quad P'(x) = P(x) \oplus E(x)$$

- 주의:
  - E(x)의 값을 구하는 것이 목적이 아님.
  - 오류가 없었다면, E(x) =0이고, P(x) = P'(x)라는 사실,
  - 오류가 있었다면, E(x) ≠ 0이고, P(x) ≠ P'(x)라는 사실이 중요
  - 결국, E(x)는 오류 패턴
    - E(x)가 0인지 아닌지를 판단할 수 있도록 C를 설계하는 것이 분석 목적

순회 중복 검사: 수신자 세부사항

- 수신자는 송신자가 보낸 $P(x)$에 전송 중 오류가 반영된 $P'(x) = P(x)+E(x)$를 받는다. $E(x)$=0은 오류가 없다는 의미
- 수신자는 ($P(x)+E(x)$)를 $C(x)$로 나눈다.
- 나머지가 0이 아니면, 즉, $(P(x) + E(x)) \% C(x) \neq 0$
  - $P(x) \% C(x) = 0$ 이므로 $E(x) \neq 0$ ⇒ 오류 발생
- 나머지가 0이면, 즉, $(P(x) + E(x)) \% C(x) = 0$
  - $E(x) = 0$ ⇒ 오류 없음
  - $E(x) \neq 0$ 면서 $E(x) \% C(x) = 0$
    ⇒ 오류는 발생하였지만 오류 검출 실패
  - 두 번째 경우가 거의 일어나지 않도록 오류 패턴을 분석해서 $C(x)$를 선택한다.

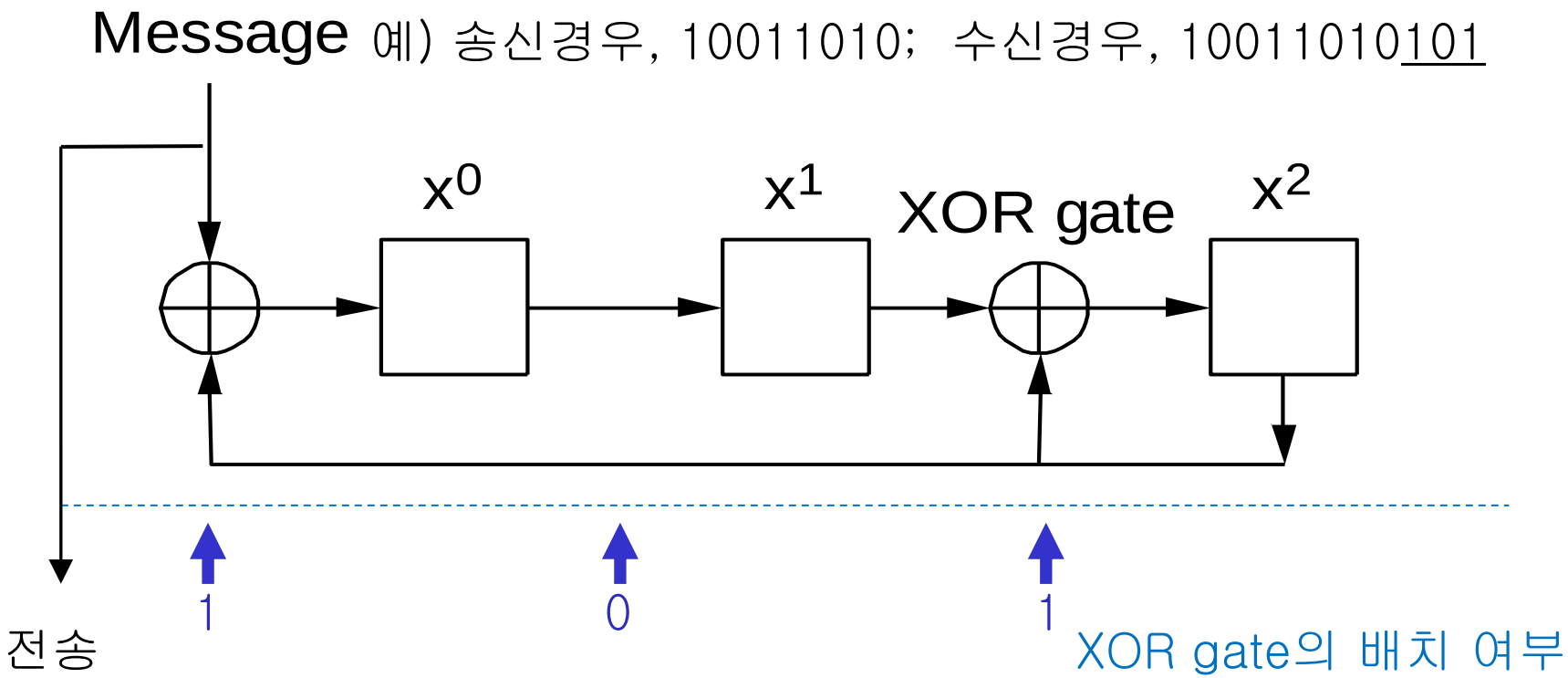
CRC 코드

보통 사용되는 $C(x)$ 다항식:

CRC	$C(x)$
CRC-8	$x^8 + x^2 + x^1 + 1$
CRC-10	$x^{10} + x^9 + x^5 + x^4 + x^1 + 1$
CRC-12	$x^{12} + x^{11} + x^3 + x^2 + 1$
CRC-16	$x^{16} + x^{15} + x^2 + 1$
CRC-CCITT	$x^{16} + x^{12} + x^5 + 1$
CRC-32	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$

CRC의 하드웨어 구현 (참조)

- 1-bit Latch와 XOR Gate를 비트패턴 *C*에 따라 조합
  - 즉, (|*C*| - 1) 길이 shift-register의 비트 사이에 XOR 배치
  - 모든 비트 통과 후, Latch에 남아 있는 결과가 CRC 비트
- 참고: frame format에서 CRC의 위치 ?
- 예) 1101



CRC에 대해 간단히 설명하시오.

- 전송되는 데이터의 오류 여부 확인을 위해 부가되어 보내지는 오류 검출 코드
  - 송신자가 계산에서 추가, 수신자가 확인
- 오류 검출율 매우 높고, 하드웨어 구현이 가능해서 거의 모든 링크 전송에서 사용
  - 예) 이더넷/무선랜 어댑터에서 CRC-32 사용; 99.999…% 검출율
- 자세한 설명?
  - XOR-나누기 연산 기초
  - 프레임 전체가 약속된 젯수로 나누어 떨어지도록, CRC 값을 만들어서 이를 데이터 뒤에 붙여 전송
  - 수신자는 프레임을 약속된 젯수로 나누어서 나머지가 0이 아니면 오류로 판단
  - 나머지가 0이면 오류가 아니라고 간주

2장 데이터 링크 네트워크 (Data Link Networks)

- ☐ 점대점(Point-To-Point) 링크
  - ☐ 하드웨어 구성요소
  - ☐ 인코딩
  - ☐ 프레이밍
  - ☐ 오류검출
- ☑ 신뢰성 있는 전송
- ☐ 이더넷 / FDDI
- ☐ 네트워크 어댑터

개요

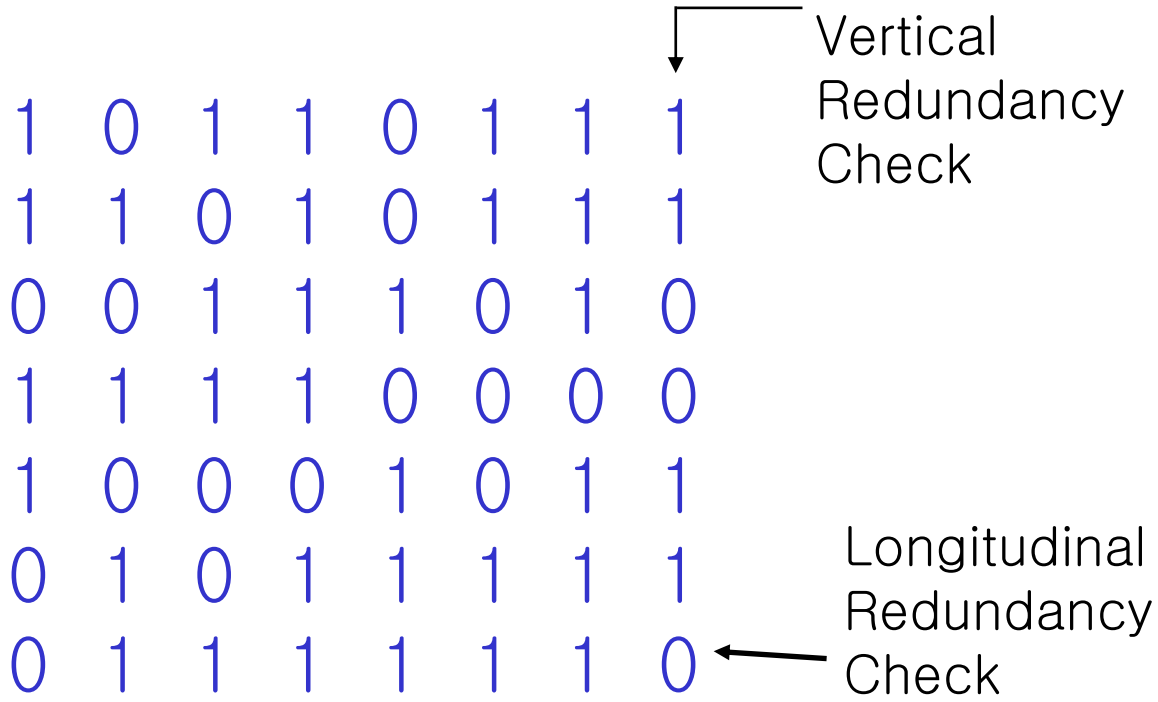
- 오류에 의해 변질된 프레임의 복구
- 오류 수정 코드 (Error Correction Codes: ECC)
  - 순방향 수정 (Forward Error Correction: FEC)
- 자동 반복 요청 (Automatic Repeat reQuest: ARQ): 재전송
  - ACK와 타임아웃 (Acknowledgements and Timeouts)
  - 역방향 수정 (Backward Error Correction)

- 참고
  - 오류 중에는 프레임 자체가 성립이 안 되는 framing error 도 있음.
    - 이 경우는 수신 쪽이 frame 수신 여부를 인식하지 못함.
    - 즉, frame loss
    - 의미는?

참조: 오류복구

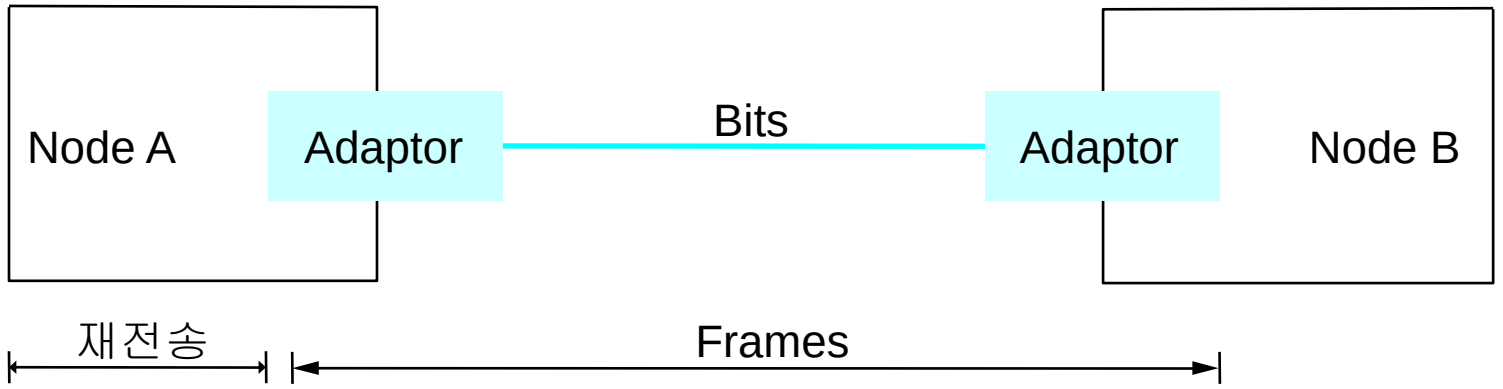
오류 수정 코드 (Error Correcting Codes)

- 예: 2차원 패리티에서는 모든 1비트 오류를 수정 가능
- Forward error correction (FEC)
- 재전송이 용이하지 않은 경우 유용
  - 예) 전화 같은 실시간 통신
    - 재전송을 통해 늦게 수신된 데이터는 가치가 없음

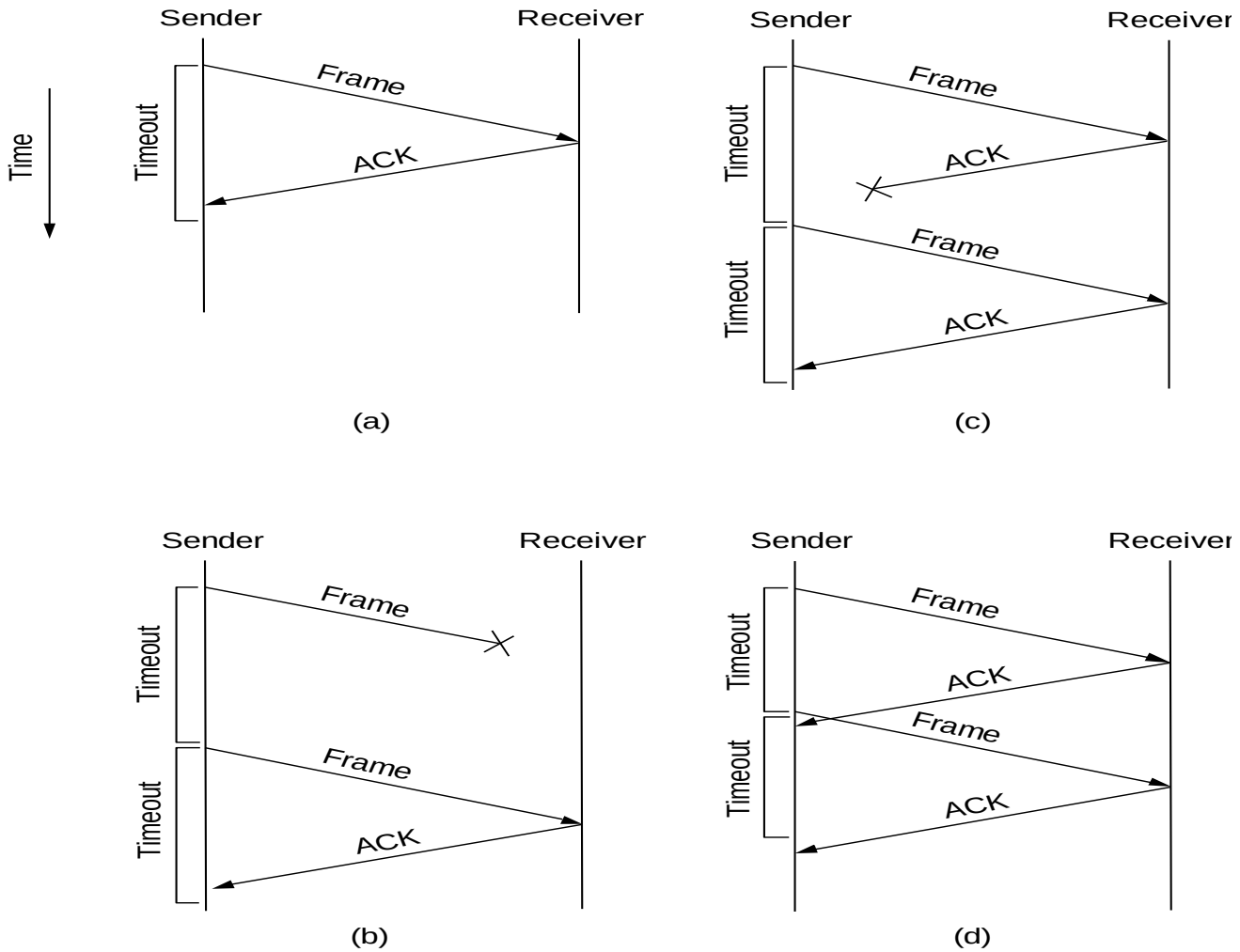


재전송을 통한 오류 복구

- 타임아웃, 재전송용 버퍼 처리 등 필요
- 어댑터에서 단독으로 처리하는데 한계
- 2계층 기능이지만, 노드 내 소프트웨어로 처리



ARQ : 응답 (ACK) 및 타임아웃

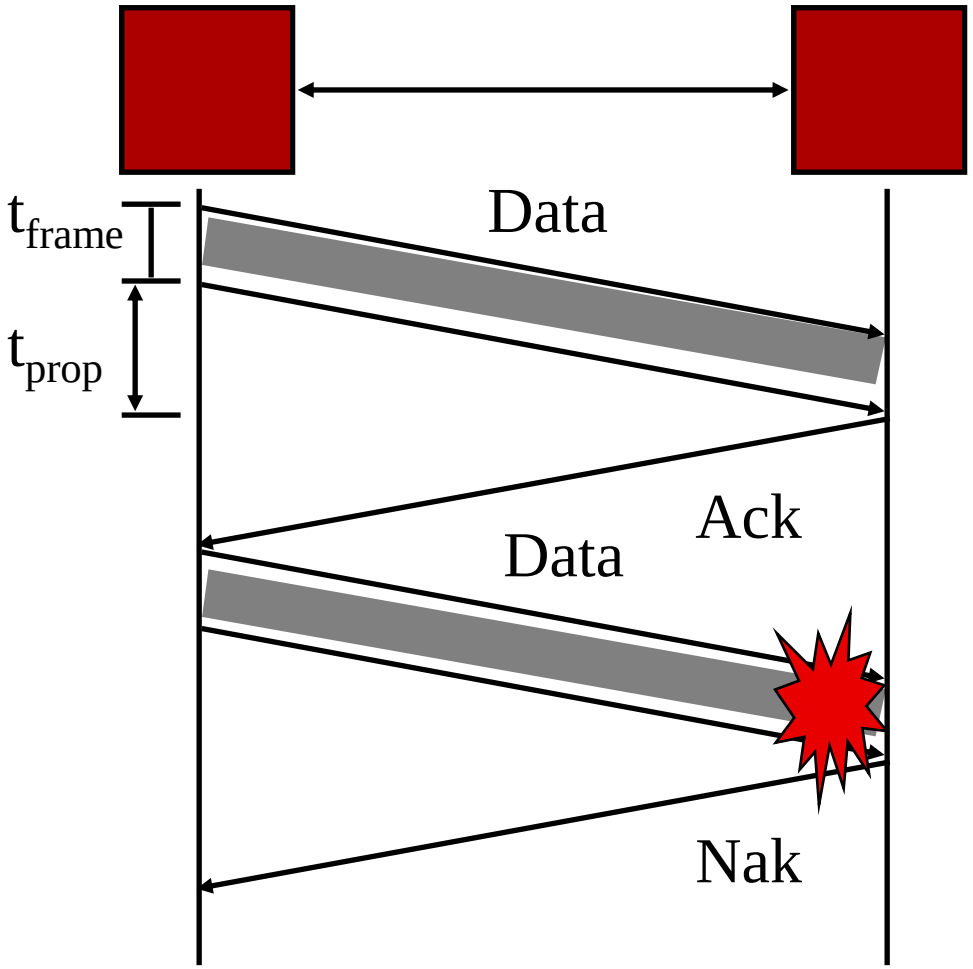


ARQ : 순서번호 (Seq. #)

- ACK 의 분실 경우 : 중복 data 문제 발생
  - 순서번호 필요
  - 수신된 중복 data는 discard
  - 그러나 ACK는 반드시 전송

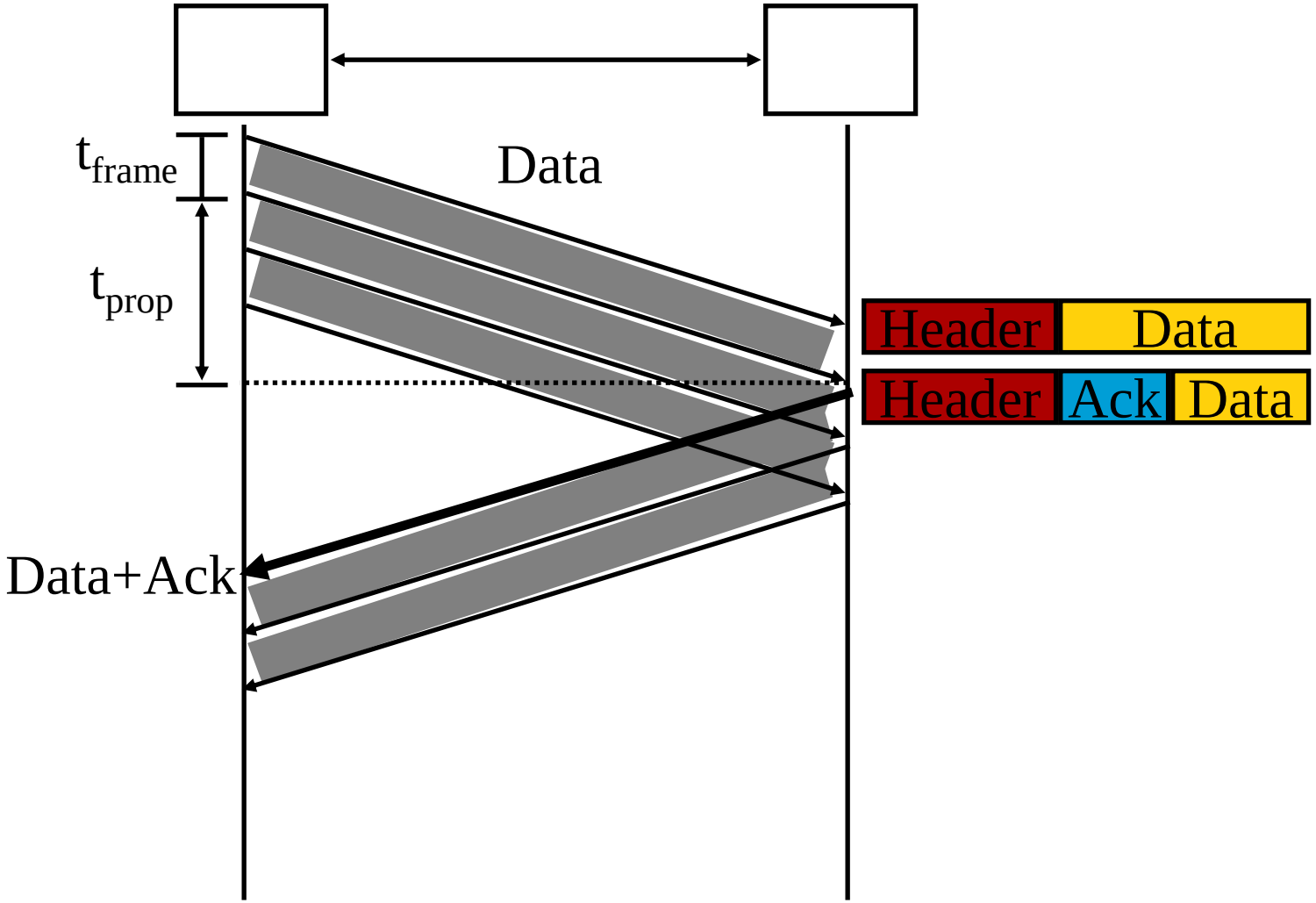
Automatic Repeat Request

- Automatic Repeat Request (ARQ)
  - Error detection
  - Acknowledgment
  - Retransmission after timeout
  - Negative Acknowledgment (optional)



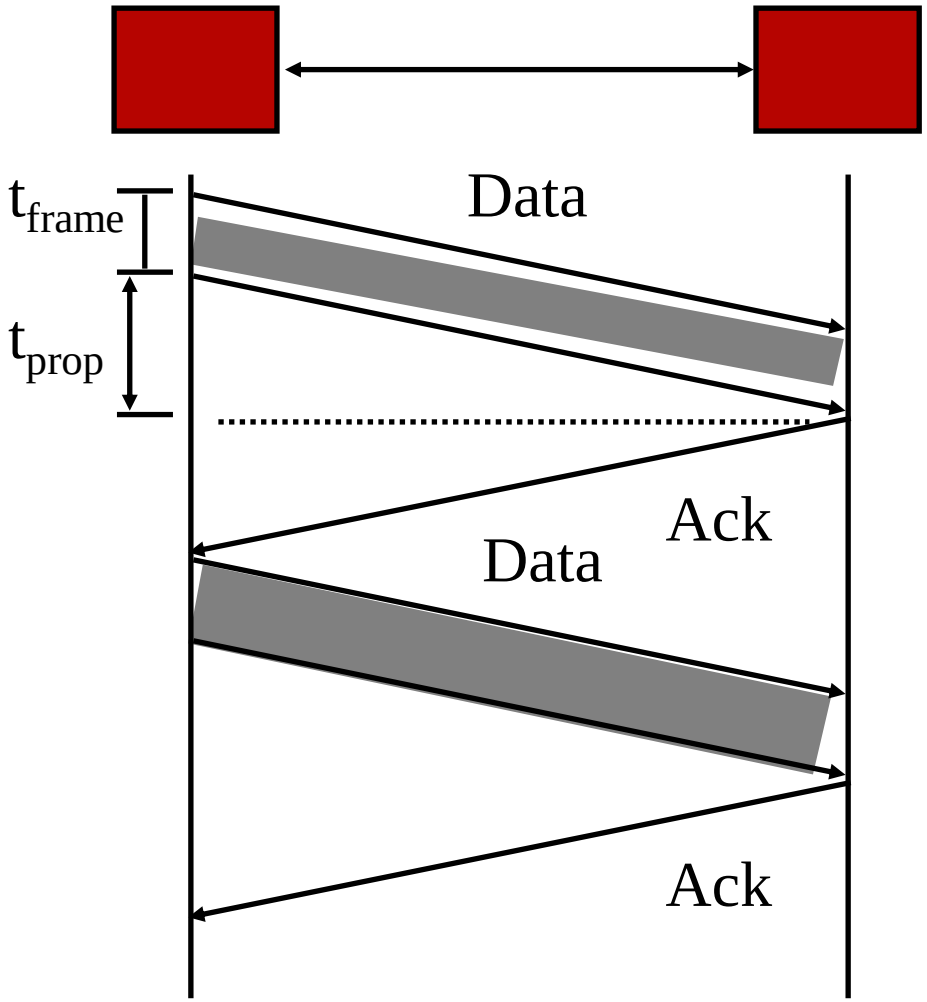
참조: 오류복구

Piggybacking



참조: 오류복구

Stop and Wait : Timing 분석



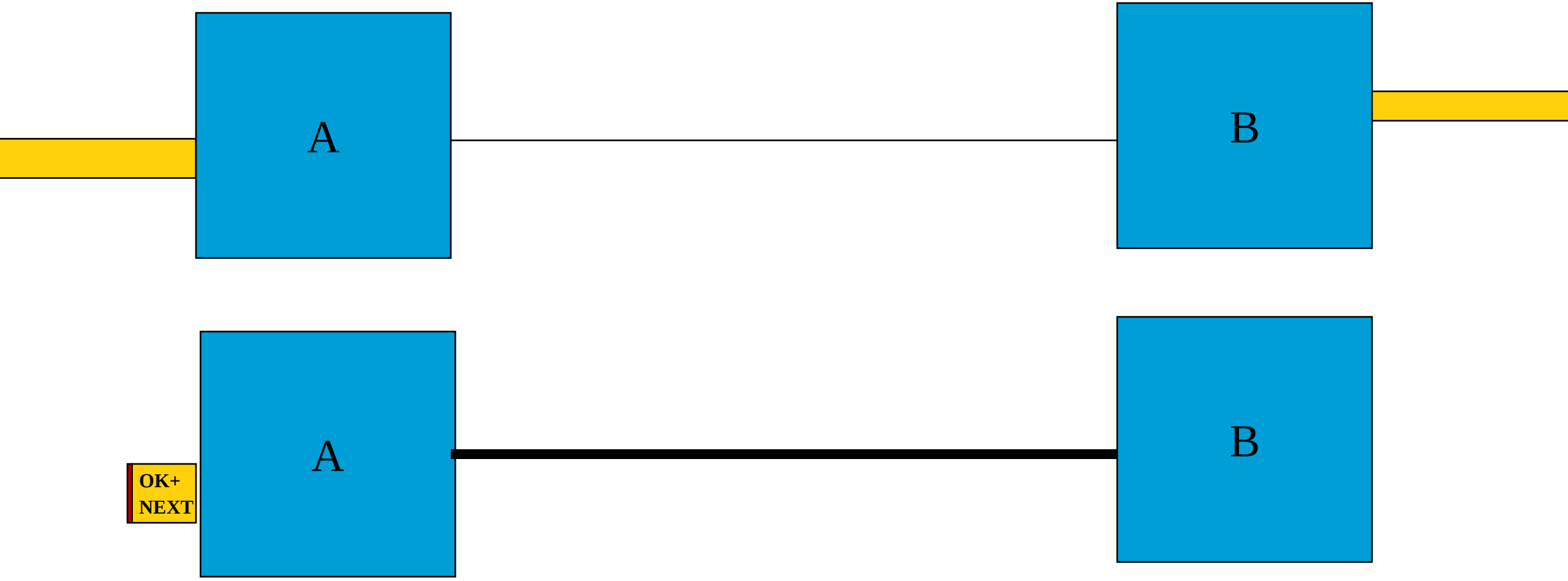
U (링크 효율/사용율)

$$\propto \quad t_{frame} \quad , \quad 1/t_{prop}$$

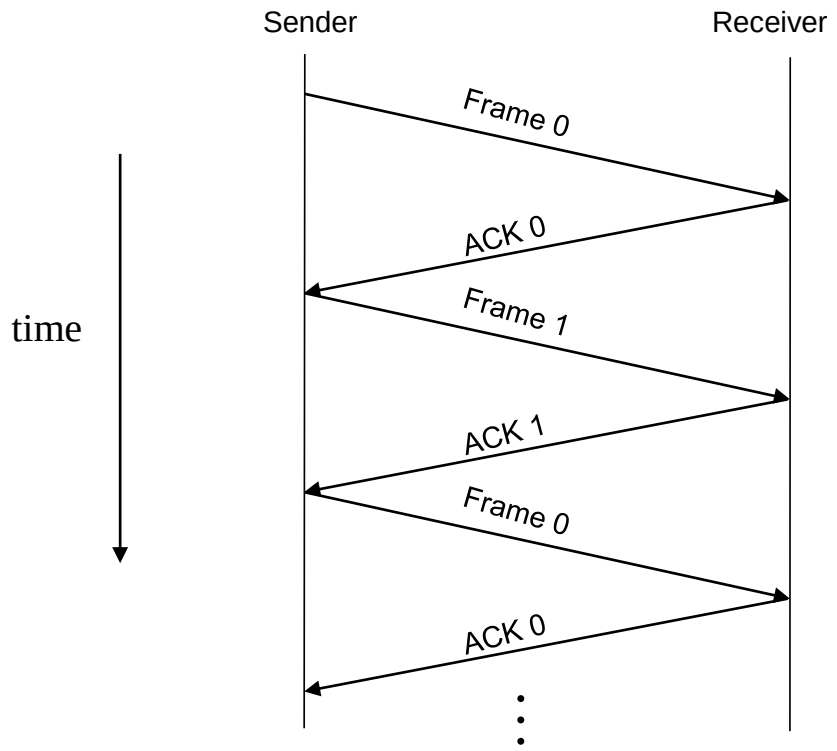
$$\propto \frac{\text{Frame size / Bit rate}}{\text{Distance / Speed of Signal}}$$

$$\propto \frac{\text{Frame size}}{\text{Distance} \times \text{Bit rate}}$$

Frames



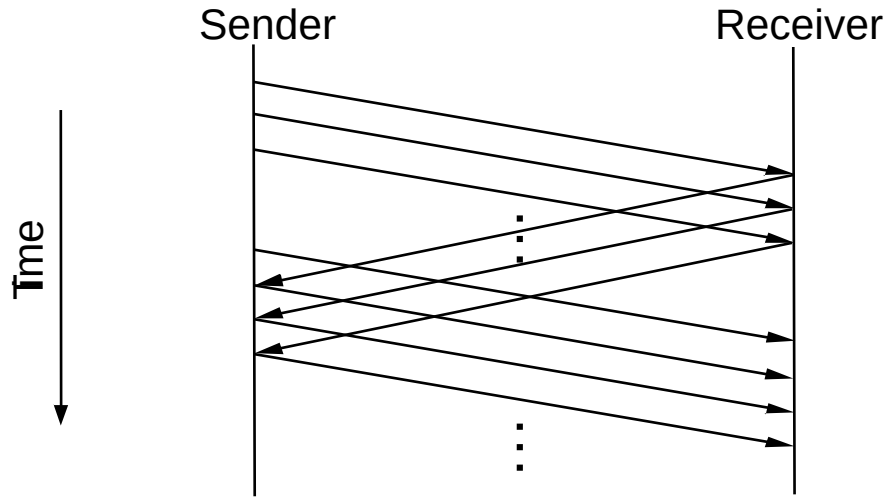
정지대기(Stop-and-Wait)



- 문제점: 파이프를 꽉 채운 상태로 유지하지 못함.
- 예: 1.5Mbps link x 45㎳ RTT = 67.5Kb (8KB). 프레임의 사이즈가 1KB인 경우, 정지 대기(stop-and-wait)는 링크 용량의 1/8만을 사용한다. 송신자는 ACK를 기다리기 전에 8개의 프레임을 보낼 수 있는 것이 바람직하다.

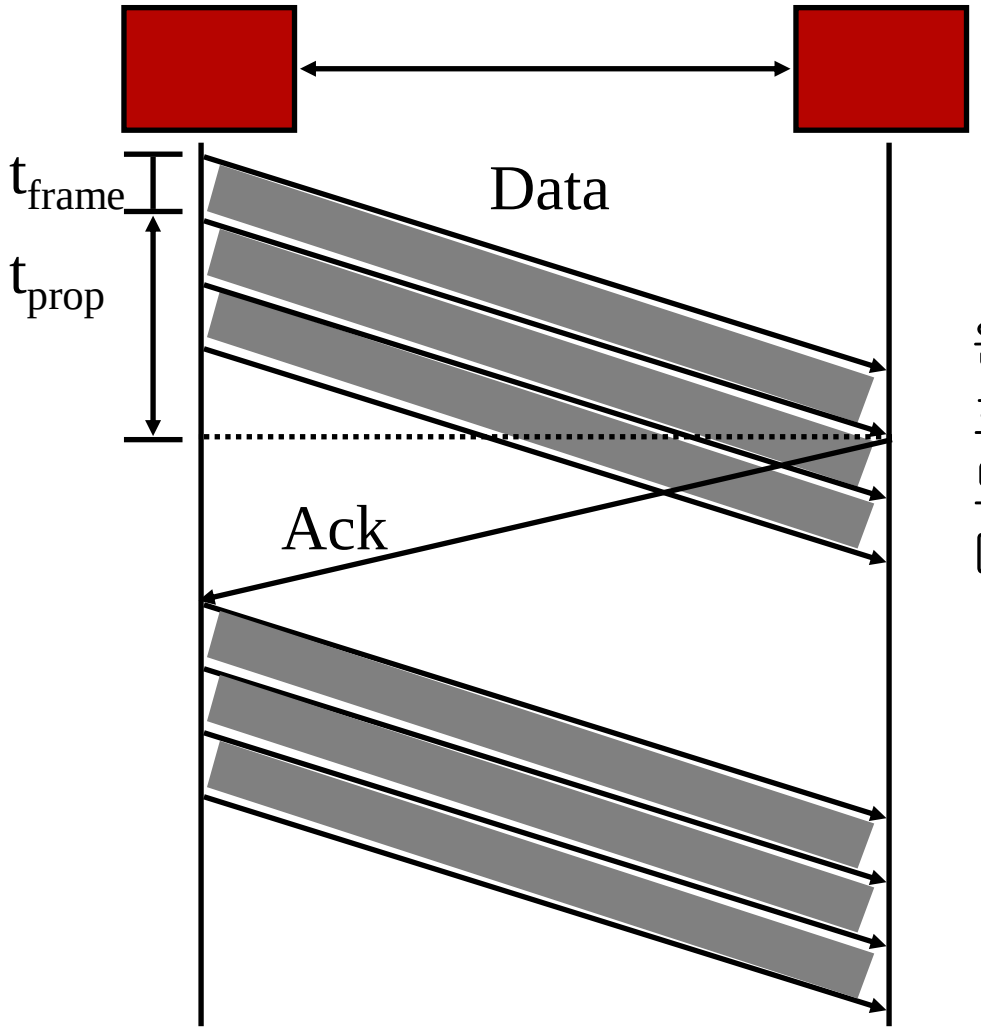
슬라이딩 윈도우(Sliding Window)

- 아이디어: 송신자가 ACK를 받기 전에 여러 개의 프레임을 전송할 수 있도록 한다. 따라서, 파이프가 꽉 차게 된다.
- ACK를 받지 않은 상태에서 보내지는 프레임(outstanding frame)이 복수로 늘어난다. 그 수는 제한된다 (window size).
  - 모두 오류 제어 대상. 순서 번호 필요.
  - stop&wait는 sliding window의 window size가 1인 경우
- 각각의 프레임에 대해서는 ARQ, 즉, Ack / timeout&재전송 을 수행
  - Ack #n 의 의미를 일관성 있게 정의해서 사용.
- 즉, 효율 높은 오류제어
  - 버퍼링보다 오류가 핵심



참조: 오류복구

Sliding Window Protocol의 성능



U (효율)

$\propto$ N, $1/t_{prop}$, t_{frame}

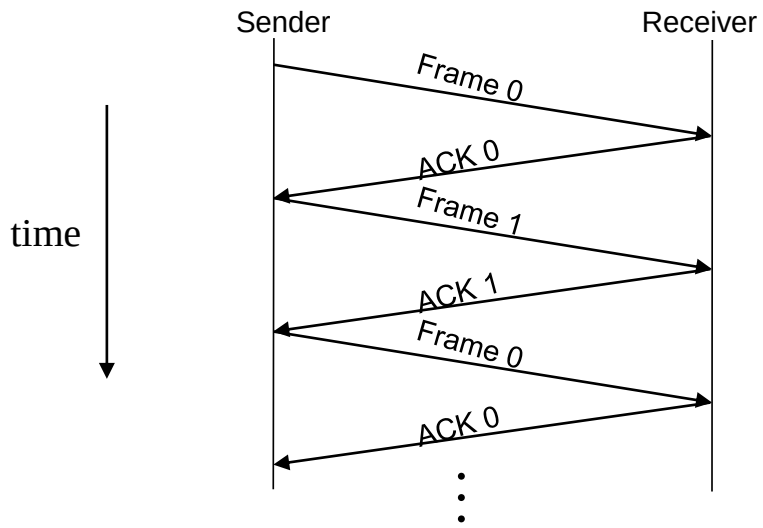
윈도우 크기(N)이 충분히 크면, 효율=1. 즉, Ack가 올 때까지 윈도우가 모두 소진되지 않으면 기다리는 시간 없이 100% 통신.

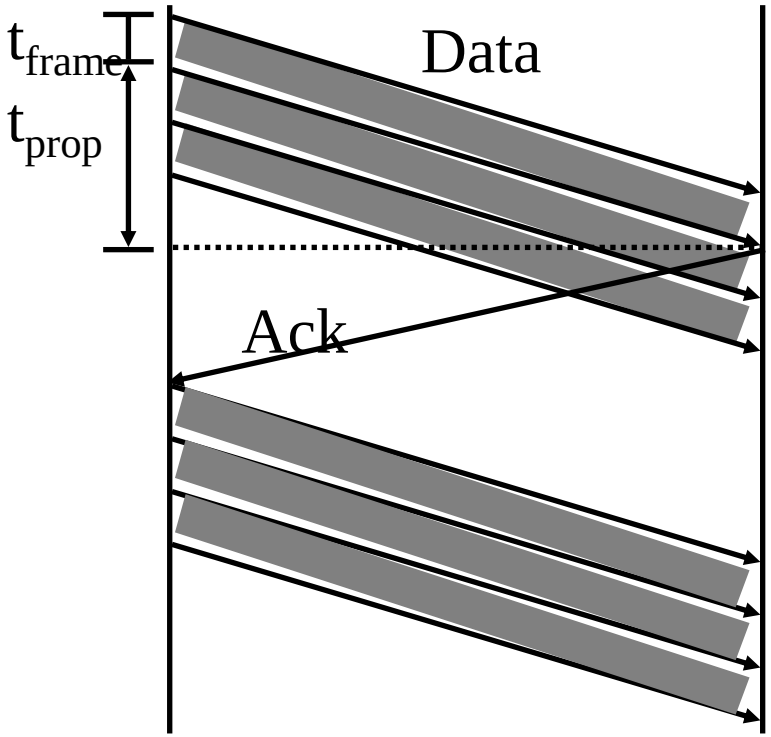
$$N \times t_{frame} \geq t_{frame} + 2\, t_{prop}$$

$$i.e., \quad N \geq 1 + 2 \times \frac{t_{prop}}{t_{frame}}$$

시간 진행 표시법 및 해석

- 두 가지 표시법 중 어떤 것 사용?



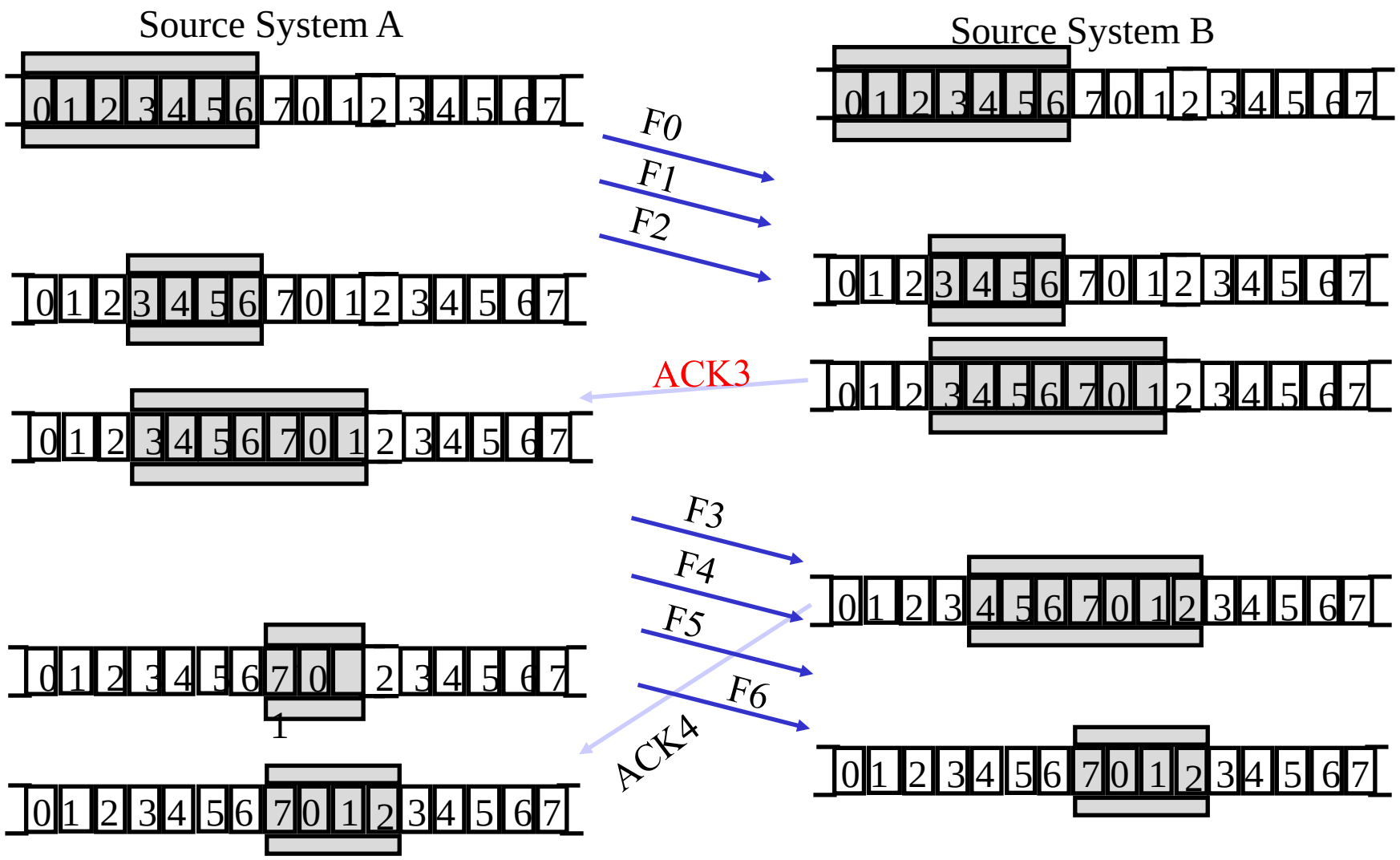


- 성능 분석이 목적?
- 시간 상황이 겹치는 세부 상황 분석이 필요?

참조: 오류복구

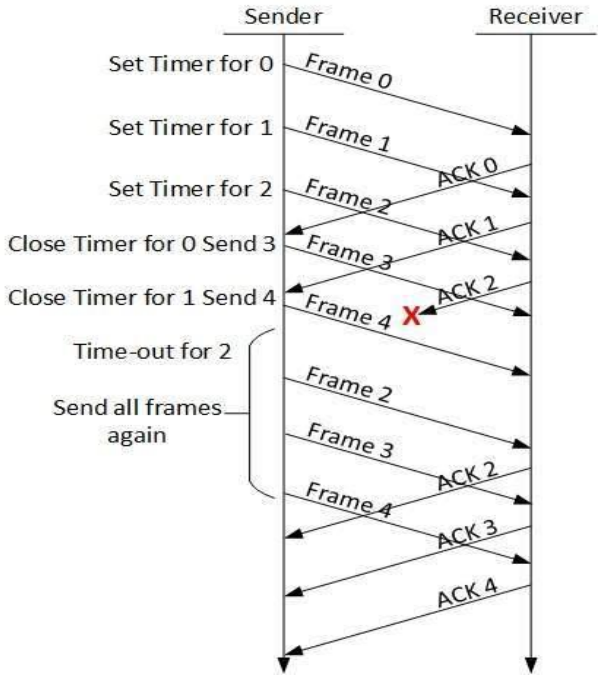
Sliding Window 개념: 오류 없는 경우

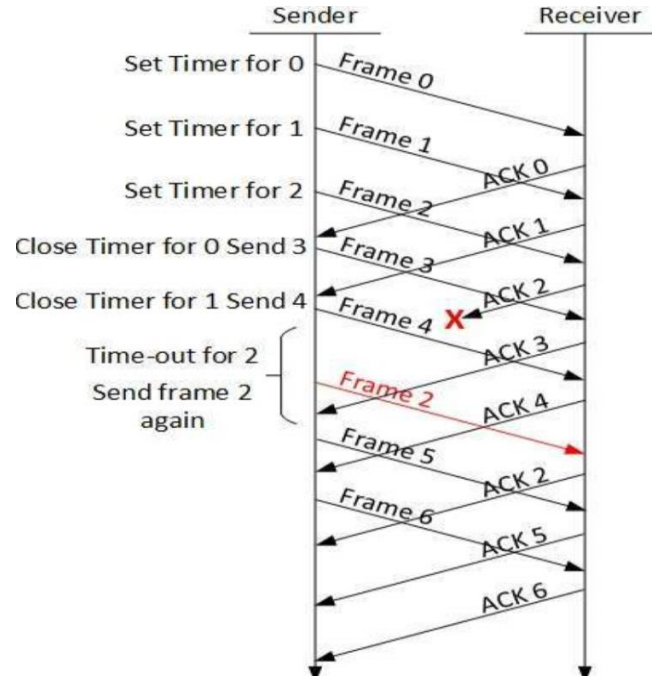
(회색부분은 더 보낼 수 있는 프레임 수인 윈도우 표시. 버퍼 표시 아님)



슬라이딩 윈도우의 오류 복구

- 복수의 outstanding frame 중 어떤 프레임에서도 오류 발생 가능
  - 따라서, 모든 프레임은 개별적으로 재전송 준비 (버퍼+타임아웃) 되어야 함.
- 오류 처리 정책
  1. Go-Back-N :오류 발생한 지점부터 새 출발
  2. Selective-Repeat : 오류 발생한 프레임만 재전송

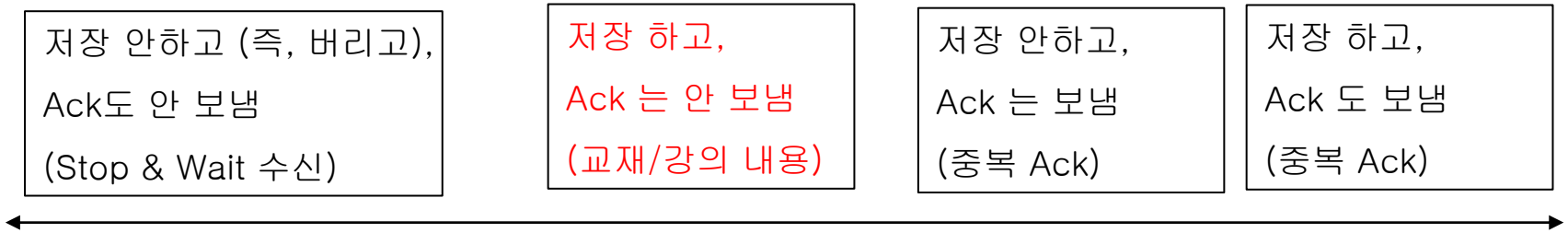




Go-Back-N (구현) 옵션

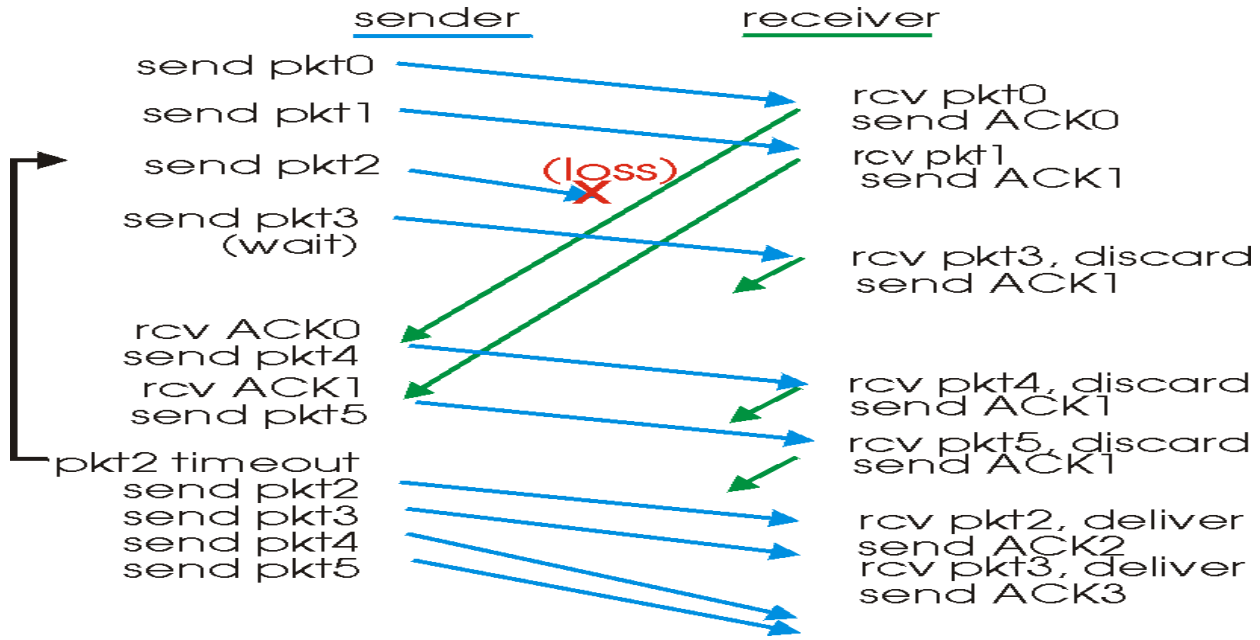
- Go-Back-N 에는 여러 옵션이 가능

- 수신 쪽에서 out-of-order 프레임을 어떻게 처리하는가에 따라서
  - Ack를 보낼 것인가? ("~까지 " 라는 누적 Ack 개념은 유지)
  - 저장을 할 것인가?



- 송신 쪽은 동일하게 동작
  - 각 outstanding 프레임에 대해
    - 재전송 버퍼에 저장; 타임아웃 설정
  - 오류 발생 인지 후,
    - 모든 outstanding frame을 동시 전송? 각 frame의 timeout에 전송?
  - 어떤 옵션으로 구현되어도 연동 가능. 즉, 오류 복구 가능

오류 처리 정책 : Go-Back-N 재검토



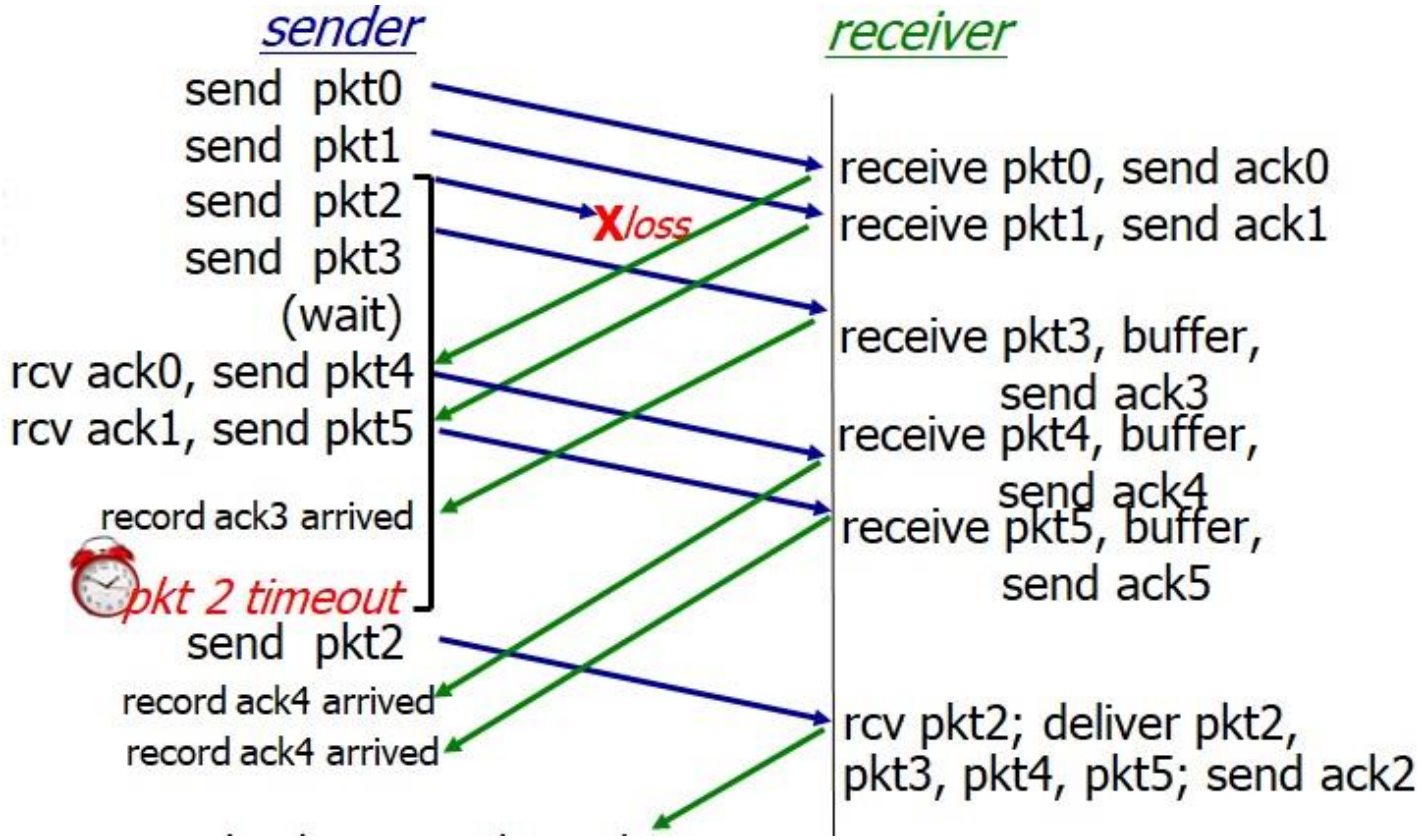
저장 안하고,

Ack 는 보냄

(중복 Ack)

- 오류 발생한 N부터 다시 출발 (N이하를 무조건 다시 보낸다는 의미 아님)
- 핵심: ACK는 "~까지 잘 받았다"는 의미의 누적(cumulative) ACK 사용
  - 교재/강의는 "~까지 잘 받았고, 다음은 몇 번"이라는 방식 사용
  - Ack를 보내는 시점은 수신자의 선택
  - 진행을 돕기 위해서 Ack를 보낸다면, 중복 ACK만 가능
- 수신 쪽에의 버퍼링은 수신자가 독립적으로 결정
  - 버퍼링 안 해도 됨.
  - 교재/강의는 버퍼 유지하면서 순서 바뀐 데이터를 수신. 단, 버퍼링이 selective repeat을 의미 하지는 않음.

오류 처리 : Selective-Repeat



- 필요 조건
  - 수신 쪽은 out-of-order 프레임 수신. 버퍼링 필수.
  - 수신 쪽의 정확한 수신 상황 정보를 송신 쪽에서 알려주어야 함.
    - "~은 잘 받았다"는 개별(individual)/선택(selective) ACK 필수

Sliding Window 세부 사항

- Sliding Window는 오류제어 프로토콜
  - 각각의 outstanding frame에 대해 기본적으로 ARQ 수행
  - 복수 개의 outstanding frame 처리를 위해 buffering 추가

- (Go-Back-N, Selective-Repeat) 선택은 송신자의 결정
  - 송신자의 재전송 방법; 수신자의 buffering은 별도 문제
  - Sending Buffer : 필수 (재전송)
  - Receiving Buffer
    - Selective-Repeat : 필수 (out-of-order 프레임 반드시 저장)
    - Go-Back-N : 성능 향상을 위한 option
  - 수신자가 out-of-order 를 저장한다고 Selective-Repeat은 아님

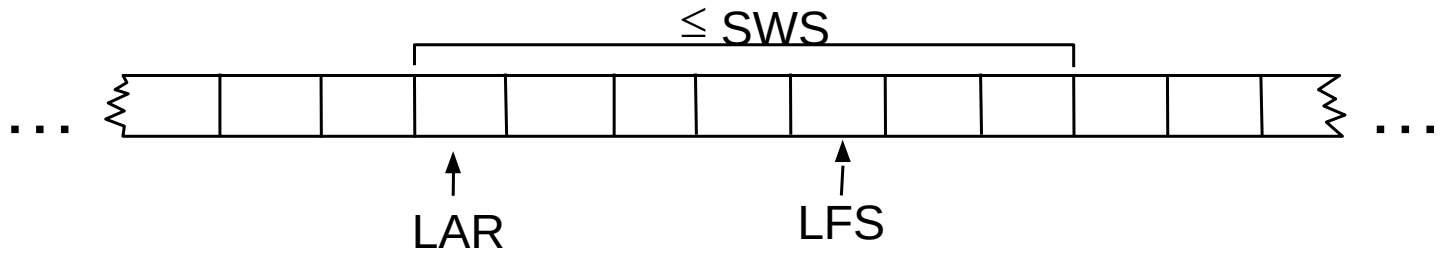
- 송신자가 Selective-Repeat 를 하기 위해서는 수신 상황 정보가 필요
  - 수신 쪽에서 송신 쪽으로 selective/individual ACK를 보내 주어야, 송신 쪽에서 selective repeat 가능

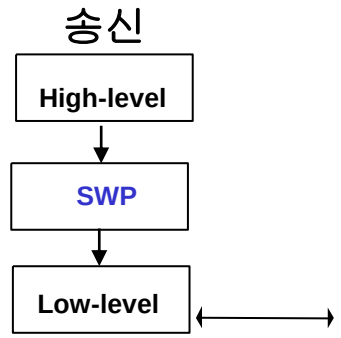
프로토콜의 구현

- 오류제어 이전까지는 Adaptor (NIC)에서 하드웨어로 구현
- 오류제어는 소프트웨어로 구현되는 첫 프로토콜
- 어떻게 프로그램 할 것인가? 얼마나 어려울까?
- 송수신 양쪽을 보는 것은 이론적 설명에서만.
  - 송수신 각각이 독립적으로 동작.
  - 프로토콜의 동작에 대해서 확실한 이해 필요.
  - 특히, 비정상 상황에 대해서.
- 난이도는?
  - concurrent and distributed program
  - 설계 과정이 절대적으로 필요!

슬라이딩윈도우(GoBackN) 세부알고리즘(1)

- 송신자
  - 각 프레임에 순서번호를 할당 (SeqNum): 각 outstanding frame의 ID
  - 세 개의 상태 변수를 유지
    - 송신창 – send window size (SWS)
    - 마지막으로 받은 ACK의 프레임 번호 – last acknowledgment received (LAR)
    - 마지막으로 보낸 프레임 – last frame sent (LFS)
  - 다음 항등식(invariant)을 유지: LFS – LAR + 1 ≤ SWS

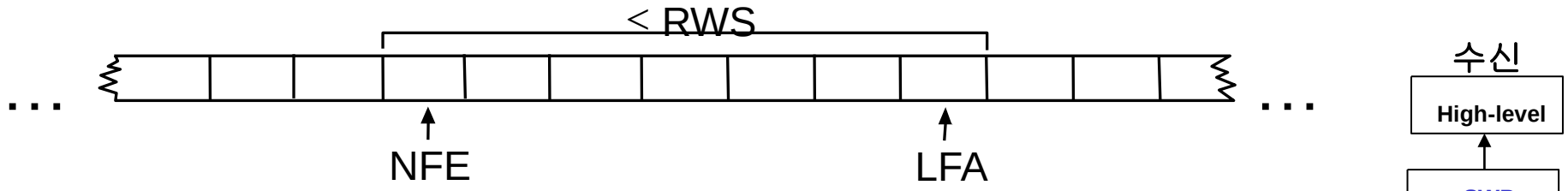




  - 상위 계층에서 전송 요청을 받으면, 1) LFS를 증가시키고 (LFS 증가가 불가능하면 wait), 2) 타임아웃을 설정한 뒤, 3) 프레임을 전송. (누구에게?)
  - ACK를 받으면, 1) 타임아웃을 해지하고, 2) LAR을 증가시키며, 3) 이에 따라 새로 창이 열리며 전송이 가능.
  - 타임아웃이 걸리면, 1) 타임아웃을 설정한 뒤, 2) 프레임을 재전송
  - SWS 만큼의 프레임은 버퍼에 유지 -- 재전송에 필요

슬라이딩윈도우(GoBackN) 세부알고리즘(2)

- 수신자
  - Out-of-order 프레임을 **저장**하는 GoBackN 알고리즘
  - 세 개의 상태 변수를 유지
    - 수신창 – receive window size (RWS)
    - 받아들일 수 있는 마지막 프레임 – last frame acceptable (LFA)
    - 수신 예상 프레임 – next frame expected (NFE)
  - 항등식(invariant)을 유지: LFA – NFE + 1 ≤ RWS



  - SeqNum의 프레임이 도착하면
    - If NFE ≤ SeqNum ≤ LFA → 받아들임(accept);
      - 중간에 빈 곳 없는 연속되는 데이터는 상위 계층으로 deliver
    - If (SeqNum < NFE) or (SeqNum > LFA) → 버림(discarded)
      - SeqNum < NFE 는 ACK 전송 필요
    - 누적 ACK(cumulative ACK)를 보낸다. (NFE 값으로)

순서 번호 공간 (Sequence Number Space)

- 순서번호는 오류제어에서 필수
- 프레임 헤더의 필드는 한정된 공간 (많이 사용하면 overhead 증가)

  ⇒ 결국, 순서 번호는 순환되며 사용
- 순서번호공간: 가능한 순서번호 구간
  - 예) 4-bit 필드 ⇒ [0..15]
- 문제: 순서번호 필드를 얼마로 잡아야 안전하겠는가?
  - 반대로, 주어진 순서번호 공간에서 최대 outstanding 프레임, 즉, 송신자 기준 WIndowSize는 얼마까지 늘릴 수 있나?

- 순서 번호 공간은 현재 전송 중인 프레임의 수보다 커야 한다.
  - 어떤 프레임이 오류 및 재전송의 대상이 될지 모르므로, outstanding Frame 각각은 서로 다른 SeqNum를 갖고 있어야 한다.
- outstanding 프레임의 최대 수 = SendingWindowSize (SWS)
- 따라서, 순서번호공간 크기 > SendingWindowSize (SWS) ← 충분?

순서 번호 공간 (2)

- “SWS ≤ 순서번호공간 크기”는 불충분.
  - 불충분한 예) 다음 장 그림 참조
  - 3-bit SeqNum 필드를 가정(0..7)
  - SWS=RWS=7
  - 송신자는 프레임 0..6 을 보냄
  - 성공적으로 도착했지만, ACKs를 잃어버림
  - 송신자는 0..6 을 재전송(retransmit)
  - 수신자는 7,0..5 를 기대하지만, 재전송된 0..5를 받게 됨

- 결론 : SWS < 순서번호공간 크기의 반
  - 즉, WindowSize, 즉, 최대로 보낼 수 있는 outstanding 프레임의 수는 순서번호공간의 반보다 작아야 안전하다.
  - 구체적으로, SWS < (MaxSeqNum+1)/2 이 정확한 규칙.
- 직관적으로 설명하면, SeqNum는 순서 번호 공간의 1/2사이를 오고 감.

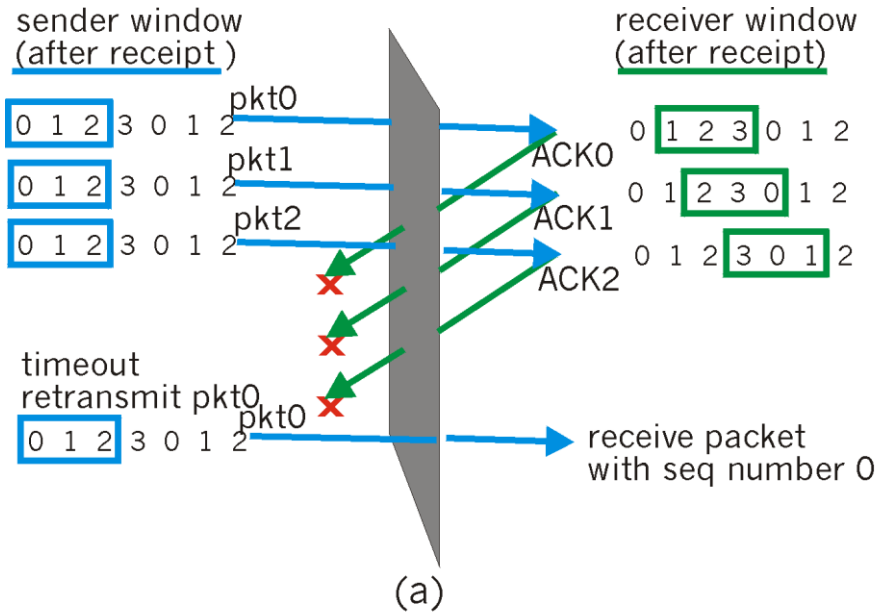
Out-of-order Receiving: dilemma

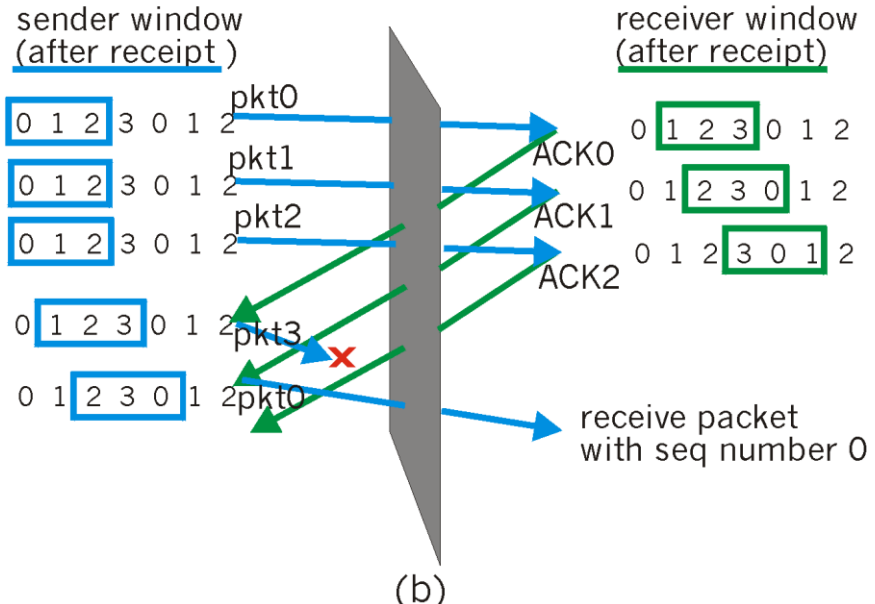
Example:

- seq #'s: 0, 1, 2, 3
- window size=3

- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?





Sliding Window Protocol 평가

- 복잡!
- 왜? “all-in-one” protocol
  - 오류 복구
  - 복수의 프레임 전송 + 버퍼 관리
  - 프레임 순서 유지

- Separate concerns!
  - 각 문제를 따로 푼다면?

동시 논리 채널 (Concurrent Logical Channels)

- 하나의 점대점(point-to-point)링크를 통해 여러 개의 논리적 채널을 동시/다중 송신함.
- 각각의 논리적 채널은 정지 대기(stop-and-wait)방식으로 운영됨

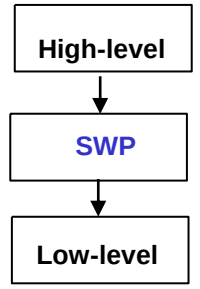
- 신뢰성 문제 : stop-and-wait
- 효율: parallel 논리 채널 운영 – 복수 프레임으로 파이프 채움
- 프레임 순서: 수신쪽에서 버퍼링

- Go-back-N or selective-repeat ?

- 실제는 거의 sliding window 사용. Why?

Sliding Window 구현

- 목적
  - 통신 알고리즘이 어떻게 구현되는가에 대한 궁금증 해결
    - 계층 구조는 어떻게 구현되는가?
  - 슬라이딩 윈도우 알고리즘 구현을 통한 확실한 이해



- 어디서, 즉, 시스템의 어느 단계에서, 동작하는 프로그램인가?
  - 프레임 하나 하나의 송수신은 NIC(Network Interface Card)이 담당
  - 슬라이딩 윈도우는 바로 그 위에서 동작 → 디바이스 드라이버
  - 계층 구조의 실체는? OSI 모델을 기준으로 한다면?

- 프로토콜 내부 동작의 실체는?
  - 계층구조에 따라, 헤더 정보를 써넣고, 읽고 처리하는 것이 기본
  - 담당 기능에 따라 세부 동작이 정해짐 (슬라이딩윈도우 – 오류제어)

- 지금까지 배운 것을 종합적으로 보여주는 예

Sliding Window 구현

```
typedef u_char   SwpSeqno;
typedef struct {
        SwpSeqno SeqNum;   /* sequence number of this packet */
        SwpSeqno AckNum;   /* allows window sizes of up to 128 */
        u_char   flags;    /* up to 16 bits worth of flags */
} SWPHdr;

typedef struct {
    /* sender side state: */
    SWPSeqno        LAR;                /* seqno of last ACK received */
    SWPSeqno        LFS;                /* last frame sent */
    Semaphore       sendWindowNotFull;
    SWPHdr hdr;                         /* pre-inintialized header */
    struct txq_slot {
        Event       timeout;    /* event associated with send-timeout */
        Msg         msg;
    } sendQ[SWS];

    /* receiver side state: */
    SWPSeqno        NFE;                /* seqno of next frame expected */
    struct rxq_slot {
        int         received;   /* is msg valid? */
        Msg         msg;
    } recvQ[RWS];
} SwpState;
```

Sliding Window 구현(2)

```
static XkHandle
sendSWP(SwpState *state, Msg *frame)
{
    struct sendQ_slot *slot;
    hbuf[HLEN];
    /* wait for send window to open */
    semWait(&state->sendWindowNotFull);
    state->hdr.SeqNum = ++state->LFS;
    slot = &state->sendQ[state->hdr.SeqNum % SWS];
    store_swp_hdr(state->hdr, hbuf);
    msgAddHdr(frame, hbuf, HLEN);
    msgSaveCopy(&slot->msg, frame);
    slot->timeout = evSchedule(swpTimeout, slot, SWP_SEND_TIMEOUT);
    return send(LINK, frame);
}
```

Sliding Window 구현(3)

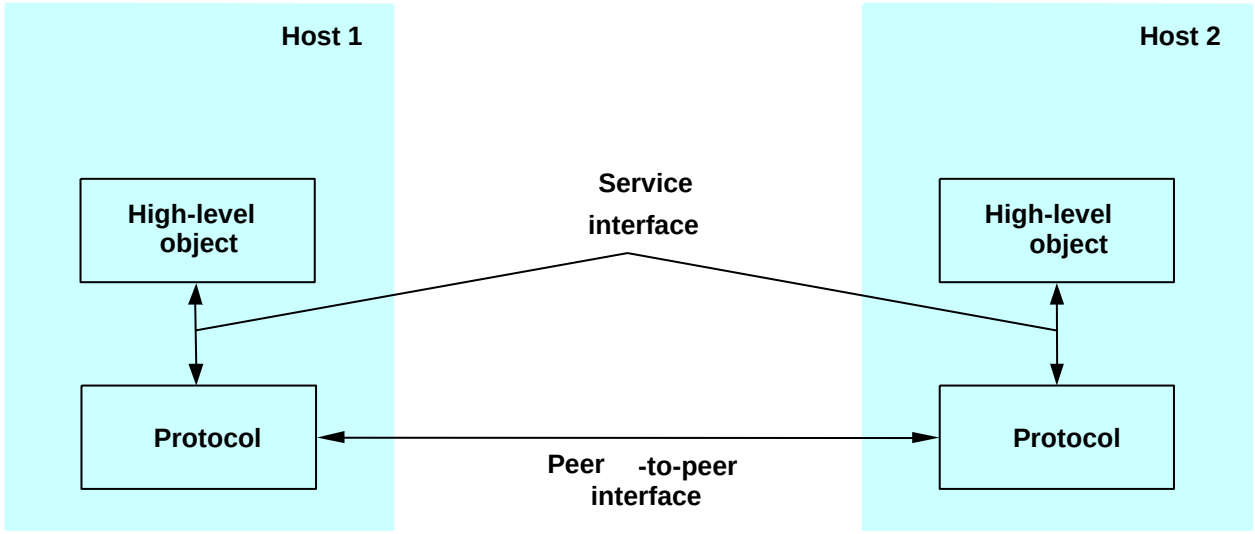
```
static int
deliverSWP(SwpState *state, Msg *frame)
    hbuf = msgStripHdr(frame, HLEN);
    load_swp_hdr(&hdr, hbuf);
    if (hdr.Flags & FLAG_ACK_VALID)
        /* received an acknowledgment --- do SENDER-side */
        if (swpInWindow (hdr.AckNum, state->LAR+1, state->LFS))
            do {
                struct sendQ_slot *slot;
                slot=&state->sendQ[++state->LAR%SWS];
                evCancel(slot->timeout);
                msgDestroy(&slot->msg);
                semSignal(&state->sendWindowNotFull);
             }while (state->stateLAR != hdr.AckNum) ;
        }
// 송신 쪽, timeout 처리는 생략
    if(hdr->Flags & FLAG_HAS_DATA)
```

Sliding Window 구현(4)

```
…
if(hdr->Flags & FLAG_HAS_DATA)
    struct  recvQ_slot *slot;
    /* received data packet --- do RECEIVER-side */
    slot = &state->recvQ[hdr.SeqNum  & RWS];
    if (!swpInWindow(hdr.SeqNum, state->NFE, state->NFE+RWS-1))
                /* drop the message */
                return SUCCESS;
    }
    msgSaveCopy(&slot->msg, frame);
    slot->received = TRUE;
    if (hdr.SeqNum  == state->NFE)
                Msg  m;
                while (slot->received)
                    deliver(HLP, &slot->msg);
                    msgDestroy(&slot->msg);
                    slot->received = FALSE;
                    slot = &state-> recvQ[++state->NFE  % RWS];
                }
                /*send ACK: */
                prepare_ack(&m, state->NFE);
                send(LINK, &m);
                msgDestroy(&m);
    }
    return SUCCESS;
}
```

프로토콜 계층/개체/인터페이스

- 1장에서 배운 일반적 정의



프로토콜 계층/개체/인터페이스의 실체

Sliding Window Protocol을 기준으로