패킷 교환: 패킷은 한 호스트(출발지)에서 시작하고 일련의 라우터들을 통과하며 다른 호스트(목적지)에 도달(store&forward: 패킷의 비트를 먼저 저장하고 라우터가 패킷의 모든 비트를 수신한다면 출력 라우터로 forwarding) // 트래픽이 많거나, 라우터에서 패킷이 들어오는 속도가 더 빠르면 지연 또는 손실 발생

// **router**: 1 패킷검사 2 Outputlink(패킷이 줄을 서야함 -> 큐잉 지연 발생) 3 Outputlink를 타고 나감(트랜스미션 지연 발생) 4 링크를 타고 나감 (프로파게이션 지연 발생)

// **처리 지연**: 패킷에 에러가 있는지 검사 // **큐잉지연**: 패킷이 큐에서 링크로 전송되기를 기다리는 시간, output router를 나갈 때 패킷이 줄 서는 시간(큐잉 지연은 트래픽이 큐에 도착하는 비율과 링크의 전송률과 도착하는 트래픽의 특성, 트래픽 강도가 0에 가까울 경우: 패킷 도착이 그물고 다음에 도착하는 큐에서 패킷을 발견하는 경우가 없음, 트래픽 강도가 1에 가까울 경우: 패킷 도착이 전송용량을 초과하여 큐가 생성됨, 도착률이 전송률보다 작아질 때 큐의 길이가 줄어듦 // **전송지연**: 패킷의 모든 비트를 링크로 밀어내는 데 걸리는 시간(패킷길이/전송률) // **전파지연**: 비트가 두 라우터 사이를 이동하는 시간(라우터 사이 거리/링크의 전파 속도) // 전송지연은 라우터가 패킷을 내보내는데 필요한 시간, 전파 지연 두 라우터 사이 거리를 이동하는데 걸리는 시간 // 전체 노드 지연 = 전파지연 + 전송지연 + 처리 지연

// ISO 7 layer – presentation: application이 데이터를 처리하는 계층(암호 등), session: data exchange시 checkpointing과 synchronization이 일어나는 계층

Application layer) client – server 구조: server(항상 동작, 고정 IP, data center for scaling), 클라이언트(서버에 요청을 보냄-간헐적 연결, 동적 IP 주소를 가짐, 무조건 서버를 거쳐 통신해야 함) // 5계층에서 end는 process이다. 실제 통신은 프로그램이 아닌 프로세스가 한다. 수신 프로세스를 식별하기 위해 IP와 port가 필요함

// 전송 계층에서 신뢰적 데이터 전송, 처리율, 시간 보장, 보안 서비스를 제공할 수 있다.(TCP-시간과 처리율, 보안을 제공하지 않는다 / UDP는 정확성, flow control, congestion control, timing, 처리율, 보안, 연결 등을 지원하지 않는다)

// web page는 객체들로 구성되어 있다(html 파일, 이미지, 오디오 파일 등), Http는 애플리케이션 계층의 프로토콜이다(hypertext transfer protocol) //HTTP는 stateless -> 객체를 요청한 기록을 남기지 않는다.(비용문제, 구현의 복잡성) // HTTP는 TCP를 사용한다.(클라이언트는 TCP연결을 한다.(기본 포트 80) 서버가 클라이언트로부터 요청을 받아드리면 클라이언트는 서버와 HTTP 요청을 주고 받는다. 연결을 종료한다. – 실제로 클라이언트가 응답 메세지를 올바로 받을 때까지 연결은 유지된다.)

// **non-persistent http:** 객체를 요청할 때, 객체 하나 당 연결하고 종료한다. 2RTT가 필요하다, 총 걸리는 시간은 transmission delay + 2RTT 이다. 매 요청마다 새로운 연결이 설정되고 끊어야 하므로 서버입장에서 부담이다. 매번 2RTT가 필요하다. // **persistent http**: HTTP/1.1 지속 연결에서 서버는 응답을 보낸 후에 TCP 연결을 그대로 유지한다. (비지속 연결도 지원한다.) 같은 클라이언트와 서버 간의 이후 요청과 응답은 같은 연결을 통해 보내진다. 즉, 같은 서버에 있는 여러 웹 페이지들을 하나의 지속 TCP 연결을 통해 보낼 수 있다. 이들 객체에 대한 요구는 진행 중인 요구에 대한 응답을 기다리지 않고 연속해서 만들 수 있다. (파이프라이닝, pipelining) 일반적으로 HTTP 서버는 일정 기간 사용되지 않으면 연결을 닫는다. // HTTP 메세지는 아스키 텍스트로 작성된다. 메세지가 다섯 줄로 되어있고, 각 줄은 CR과 LF로 구별된다. 마지막 줄에 이어서 CR과 LF가 따른다. HTTP 요청 메세지의 첫 줄은 요청 라인(method, url, http version), 그 이후 줄은 헤더 라인(호스트, 연결(non persistent인지), user agent(브라우저 타입), 객체 언어)

// **쿠키**: HTTP서버가 상태를 유지하지 않는 점을 보완해서, 사용자의 데이터를 캐싱하는 방식 // 쿠키 동작 과정: HTTP 메세지 요청, 서버는 클라이언트에게 식별 번호 부여해서 테이블을 생성, 클라이언트에게 쿠키 번호를 포함한 응답 메세지 전송, 브라우저는 헤더를 보고 관리하는 쿠키 파일에 그 라인을 덧 붙임, 클라이언트가 동일한 웹 서버에 요청을 보낼 때 서버는 식별번호를 참조하여 요청을 주고받음

// **웹캐싱**: proxy server, 1 브라우저는 웹 캐시에 TCP연결하고 웹 캐시에 있는 객체에 대해서 HTTP요청을 보낸다. 2 웹 캐시는 객체의 사본이 있다면 클라이언트에 전송한다. 3 없다면 웹 캐시는 오리진 서버에 TCP로 연결하고 객체를 요청한다. 웹 캐시는 오리진 서버로부터 객체를 수신 후 이미 연결한 TCP 소켓을 통해 메세지를 전송한다.

// 일반적으로 웹 캐시는 ISP가 구입하고 설치한다. // **웹 캐싱의 사용 이유**: 1 클라이언트 요구에 대한 응답 시간 감소 2 웹캐시는 한 기관에서 인터넷으로 접속하는 링크 트래픽을 대폭 줄일 수 있다. 3 인터넷 전체의 웹 트래픽을 실질적으로 줄여 모든 애플리케이션의 성능이 좋아진다. // 웹 캐시를 로컬에 설치할 경우: 전체 요청 중 n%는 로컬에 위치한 웹캐시가 처리하므로 링크 처리율이 감소, 링크 처리율이 감소하면 자연스럽게 오리진 서버로 가는 트래픽이 감소 (라우터 밖으로 나가는 부담이 적어지니 전파지연에 좋은 영향을 끼침), 단 오버헤드가 발생한다. (데이터 업데이트, 복잡한 context, money) // **Conditional Get**: 클라이언트에에 캐싱된 후, 서버에 있는 객체가 저장될 수 있는 문제를 해결하기 위해 브라우저로 전달되는 모든 객체가 최신인지 확인하며 캐싱 (1 브라우저의 요청을 대신해서 웹 캐시가 요청메세지를 서버로 보냄. 2 웹 서버는 캐시에게 객체를 가진 응답 메세지를 보낸다. 캐시는 객체를 저장하고 보낼 때 update date도 함께 저장한다. 3 시간이 지난 후 클라이언트가 같은 웹서버에 요청을 보냈을 때 클라이언트는 조건부 Get을 수행(if-modified) 4 변경되지 않았다면 같은 객체를 보낸다.

// **DNS(domain name system)**: 호스트 이름을 IP주소로 변환해주는 서비스 – 계층 구조로 분산된 데이터베이스, 호스트가 분산 데이터베이스로 질의하도록 하는 애플리케이션 계층 프로토콜이다. **UDP 사용**, 포트번호 53 (Why UDP? 연결 설정에 드는 비용이 없고 DNS는 신뢰성보다 속도가 더 중요한 서비스이므로 UDP가 더 적합, 전달하는 패킷의 크기가 작기 때문에 신뢰성이 보장되지 않아도 됨 – 못 받으면 다시 전달하며 됨 & 비 연결형 프로토콜이므로, 어떤 정보도 기록하지 않고 유지할 필요가 없다 즉, DNS 서버는 TCP보다 많은 클라이언트를 수용할 수 있으므로 연결 상태를 유지하지 않고 정보 기록을 최소화할 수 있는 UDP를 채택하였다.) 사용자의 호스트에서 수행하는 브라우저가 Url을 검색했을 때 발생하는 일(1 같은 사용자 컴퓨터는 DNS 애플리케이션의 클라이언트를 수행한다. 2 브라우저는 URL로부터 호스트 이름을 추출하고 그 호스트 이름을 DNS 애플리케이션의 클라이언트에 보낸다. 3 DNS 클라이언트는 DNS 서버로 호스트 이름을 포함하는 질의를 보낸다. 4 DNS 클라이언트는 결국 호스트 이름에 대한 IP 주소를 할당받는다. 5 브라우저가 DNS로부터 IP 주소를 받으면, 브라우저는 해당 IP 주소와 그 주소의 80번 포트에 위치하는 HTTP 서버 프로세스로 TCP 연결을 초기화한다.) DNS는 위 예시에서 알 수 있듯이 추가 지연을 주지만, local DNS서버에 캐싱되어 있어서 DNS 네트워크 트래픽을 줄일 수 있다. // DNS가 중앙 집중 데이터베이스라면? 만약 DNS가 간단한 설계로 모든 매핑을 포함하는 인터넷 네임 서버를 갖고 있다면 서버의 고장에 대처할 수 없고, 트래픽양이 과부화되고 모든 유저를 관리하기 어려워진다. // 계층 DNS(root, TLD, Authoritative DNS(DNS 레코드가 존재하는 계층), local DNS(ISP에 존재, 대체로 호스트에 가까이 있음)) // 동작 과정(로컬 DNS 쿼리를 보냄-호스트 이름, 로컬 DNS 서버는 질의 메세지를 루트 DNS 서버에게 전달, 루트 DNS는 tld(.com)를 인식하고 TLD서버로 보냄, TLD서버는 host.com을 인식하고 책임 DNS로 다시 질의를 보냄, 최종 IP주소를 얻고, 호스트에 최종 IP주소를 응답) // 재귀적 질의는 높은 계층에 있는 DNS서버가 책임져야 하는 것들이 많다. // 중요한 infra를 지키는 것이 훨씬 낫기 때문에, 중요한 root name server보다는 default name server가 일을 더 하는 것이 좋다. // DNS 자원 레코드: DNS서버가 IP주소를 매핑하기 위한 자원 레코드를 저장 // DNS 메세지(header(12byte), payload(질문 영역, 답변 영역, 책임 영역, 추가 영역)) // DDos: 루트 서버로 다량의 패킷을 보내 다른 DNS 질의들이 응답을 받지 못하게 함 // DNS 중독: DNS 서버로 가짜 응답을 보내, 그 서버가 자신의 캐시에 가짜 레코드를 받아들이도록 속임수를 쓰는 것

Transport Layer) 송신측은 프로세스의 애플리케이션 계층에서 받은 메세지를 다중화한다.(세그먼트로 변환) 송신 종단 시스템에 있는 네트워크 계층으로 세그먼트를 전달한다. 수신측에서 네트워크 계층은 데이터그램으로 부터 트랜스포트 계층 세그먼트를 추출하고 트랜스포트 계층으로 세그먼트를 보낸다. 트랜스포트 계층은 수신 애플리케이션에서 세그먼트 내부 데이터를 이용할 수 있도록 수신된 세그먼트를 처리한다. // 네트워크가 호스트들 사이 논리적 통신을 제공한다면 트랜스포트 계층은 다른 호스트에서 작동하는 프로세스들 사이의 논리적 통신을 제공한다.

// 트랜스포트 레이어는 유일한 식별자를 가짐: 각 세그먼트는 전달될 적절한 소켓을 가리키는 특별한 필드를 가짐(출발지 포트, 도착지 포트) // 호스트의 각 소켓은 포트 번호를 할당 받음, 세그먼트 호스트가 도착하면 트랜스포트 계층은 세그먼트안의 목적지 포트 번호를 검사하고 그에 상응하는 소켓으로 세그먼트를 내보냄, 세그먼트 데이터를 소켓을 통해 해당하는 프로세스로 전달됨

// **UDP**: 프로세스로부터 메세지를 가져와서 출발지/도착지 포트 첨부, 출발지/도착지 호스트의 IP 주소 필드를 추가한 후 최종 트랜스포트 계층 세그먼트를 네트워크 계층으로 넘겨줌 // 네트워크 계층은 세그먼트에 헤더를 붙여 데이터그램으로 만든 후 수신 호스트에게 전달, **UDP는 세그먼트를 해당 프로세스에게 전달하기 위해 포트 번호 사용**

// **UDP 장점**: 연결 설정이 없다 – 빠르다, 연결 상태가 없다, 패킷오버헤드가 작다(8바이트, TCP의 경우 20바이트이다.) // 단점: 혼잡 제어가 없다. (네트워크 상태가 혼잡할 경우 라우터에 많은 패킷 오버플로우가 발생한다.) // UDP 체크섬: 만약 패킷에 어떤 오류가 없다면 수신자에서 합은 `1111111111111111`, `0`이 하나라도 있을 경우 패킷에 오류가 발생 // UDP는 왜 체크섬을 제공하는가? 출발지와 목적지 사이의 모든 링크가 오류 검사를 제공한다는 보장이 없기 때문 – 단, 수정은 하지 않는다.

// **신뢰적 전송의 원리(RDT 3.0):** 비트 오류와 손실 있는 채널상에서의 신뢰적 데이터 전송 – **패킷 손실 검출, 패킷 손실 대응** –> 송신자에게 손실된 패킷의 검출과 회복 책임을 부여 –> 타이머를 사용해서 패킷과 ACK손실에 대응 BUT 송신자는 ACK을 받을 때까지 아무런 액션을 취하지 않음 따라서, 파이프라이닝 방식을 사용해 송신자에게 확인응답을 기다리기 전에 송신을 허용하도록 함 // 체크섬, 타이머, 타임아웃, 순서 번호, ACK, NACK, WINDOW(전송률), PIPELINING

// **GBN:** 확인이 안된 가장 오래된 패킷의 순서 번호를 base로 한 다음 `nextseqnum`까지 N개의 패킷을 순서대로 보냄(이때 N은 윈도우 사이즈라고 부름)이때 클라이언트가 Ack을 수신하지 않으면 해당 패킷을 base로 한 다음 다시 `window sizw`만큼 패킷 전송 – **이 과정에서 순서가 잘못된 패킷은 모두 버림 –> 많은 재전송이 필요할 수 있음** // **채널 오류의 확률이 증가할수록 파이프라인은 불필요한 재전송 데이터로 채워짐**

// **Selective Repeat:** 수신자에게 오류가 발생한 패킷을 수신했다고 의심되는 패킷만 재전송 – 순서가 바뀐 패킷들은 빠진 패킷이 수신될 때 까지 버퍼에 저장, 빠진 패킷이 수신된 시점에서 일련의 패킷을 순서대로 상위 계층에 전달할 수 있음 // 타임아웃: 타이머는 손실된 패킷을 보호하기 위해 재사용됨(각 패킷은 자신의 논리 타이머가 필요함) – 해당 패킷 타이머의 시간이 다 지났을 경우 해당 패킷만 다시 요청한다. (그 앞 패킷은 버퍼에 저장하다가 빠진 패킷이 다시 왔을 때 다 함께 상위 계층에 전달) // **딜레마:** 1 모든 Ack을 받지 않은 모든 패킷에 대해서 타이머가 있어야 한다. 2 reciever입장에서 sr은 올바르게 동작하지만 sender입장에서는 문제가 생길 수 있다. Seq number가 window size보다 충분히 커야 제대로 동작한다. 그렇지 않은 상황에서 패킷손실이 발생하면 문제가 발생할 수있다.(최소한의 윈도우 크기: SR프로토콜에 대한 순서 번호 공간 크기의 절반보다 작거나 같아야 함)

// **TCP(transmission control protocol)**: 신뢰적이고 연결지향형 서비스를 제공, flow control / congestion control 제공 // **TCP 소켓: 출발지 IP 주소, 출발지 포트 번호, 목적지 IP주소, 목적지 포트 번호** // **연결**: 3-way handshake 1 클라이언트 애플리케이션 프로세스는 서버의 프로세스와 연결을 설정하기 원한다고 TCP 클라이언트에게 먼저 알린다. 2 클라이언트의 트랜스포트 계층은 서버의 TCP연결을 시작한다. 3 서버는 요청에대한 ACK을 보낸다. 4 마지막으로 클라이언트가 ACK으로 응답후 연결을 설정한다.(마지막 세그먼트는 페이로드를 포함할 수 있다.) // 연결을 끊을 경우: 클라이언트가 서버에게 Fin bit = 1인 세그먼트를 보냄(데이터를 보낼 수 없지만 받을 수는 있는 상태), 서버는 여전히 데이터를 보낼 수 있으며 마저 보내야 할 데이터를 다 보낸다. 서버는 다 보낸 후 연결을 끊겠다는 Fin bit을 보낸다. 각각 finish에 대한 ACK을 받은 후 연결을 종료한다.

// 최대 세그먼트 크기: MSS (통상 40 바이트) // TCP 헤더 필드(출발/도착 포트 번호, 체크섬 필드, 순서번호 32비트, 확인응답ACK 32비트, 수신 윈도(흐름제어에 사용 – 수신자가 받아들이려는 바이트의 크기), 옵션 필드, 플래그 필드 // **Timer**: 시간제한이 너무 작을 경우 불필요한 재전송이 발생하고, 클 경우 패킷 손실에대한 반응이 너무 느려진다. 효율적인 방식은 RTT보다 살짝 큰 경우이다. 샘플 RTT(평균 값)을 활용하여 -> Estimated RTT = (1-a)Estimated RTT + a*Sample RTT를 활용하여 -> Timeout Interval = Estimated RTT + 4 * Dev RTT(margin)

// **TCP fast retransmit**: 세그먼트가 손실되었을 때 ACK이 연속적으로 온다. 이 경우 송신자가 동일한 ACK을 3번 받았을 경우 수신자에게 segment를 재전송한다.(ACK된 세그먼트의 다음 3개의 세그먼트가 분실되었음을 의미 따라서 타이머가 만료되기전에 재전송 가능)

// **Flow Control**: 패킷을 주고 받는 과정에서 transport layer로 패킷들이 도착, 네트워크 레이어에서 데이터를 전송하는 속도가 빠를 경우 트랜스포트 레이어의 소켓 버퍼에(receiver) 많은 데이터가 쌓여 손실될 수 있다. 따라서 수신자는 자신이 받을 수 있는 바이트 수를 조절하여 송신측에게 전달해 buffer가 overflow되지 않도록 조절한다.

// **Congestion Control**: transmission rate(window size)가 점점 증가함에 따라 데이터 전송률을 감당하지 못해 버퍼에 오버플로우가 발생해 손실이 발생할 수 있음(너무 많은 패킷이 보내지는 상황), AIMD를 활용해 sender가 window size를 조절할 수 있다. → flow control은 sender와 host사이 overflow를 다루지만 congestion flow는 windowsize를 조절한다.(전송률) // **AIMD(additive increase multiplicative decrease)**: 패킷손실이 일어나기 전까지 cwnd(congestion window size)를 linear하게 증가하시킨다. 패킷 손실이 발생한 시점부터는 cwnd를 절반씩 감소시킨다.

Network layer) 모든 호스트와 라우터에는 네트워크 계층이 존재한다. // **forwarding(data plane)**: 인풋 포트 라우터로 들어온 datagram을 어떤 아웃풋 포트 라우터로 보내야 할지 결정(forwarding table활용), **routing(control plane)**: forwarding을 바탕으로 라우팅 알고리즘을 활용해 어떤 경로로 어떻게 갈지 결정(tradition: 모든 라우터 마다 라우팅 알고리즘 존재, 각 라우터끼리 통신, SDN: remote server에 각각 라우터들이 구현됨)

// **Router**: input port[decentralized switching(forwarding table을 이용해 어떤 아웃풋 포트로 나갈지 확인 – destination-based forwarding (longest prefix matching)), switching fabric [input port에 있는 패킷들을 output packet으로 이동시키는 역할], outport[transmission rate보다 빠르게 datagram이 버퍼에 도착하면 버퍼링 발생 → 손실 또는 지연의 원인 → scheduling decipline을 활용해 대기중인 패킷 중 정해진 우선순위에 따라 어떤 패킷(데이터그램)을 보낼지 결정(priority scheduling에 따라 결정)

// **IP**(라우팅 프로토콜: 패킷의 경로, 인터넷 프로토콜: 패킷을 데이터그램으로 만듦, ICMP: 에러 기록) // **transport layer**에서 **network layer**로 내려온 segment는 IP헤더가 추가적으로 붙어 datagram의 형태로 패킷을 주고 받음 - datagram = segment + IP 헤더 **//** IP fragment: 데이터그램을 나눠 나중에 재 조립(MTU: 최대 단위 – 1500바이트, offset = length/8) // IP 헤더: 버전, 헤더 길이, 서비스 타입, 데이터그램 길이, 식별자, 플래그, TTL, 프로토콜, 헤더 체크섬, IP주소(src/dest), 옵션, 페이로드(세그먼트)

// **서브넷**: 라우터 없이 통신할 수 있는 고립된 네트워크 eg. a.b.c.d/x → x는 서브넷 마스크 // CIDR: 인터넷 주소 할당 방식으로 서브넷 주소 체계 표기를 일반화

예를들어 200.23.16.0/23에서의 서브넷 마스크는 23, 사용가능한 서브넷의 크기(한 서브넷에서 할당 가능한 호스트 수)는 2*(32-23)

// **주소 블록 획득**: 기관의 서브넷에서 사용하기 위한 IP블록을 얻기 위해 네트워크 관리자는 이미 할당받은 주소의 큰 블록에서 주소를 제공하는 ISP와 접촉해야 함 // 상위 블럭이 a.b.c.d/20이고 하위 서브넷 마스크가 23일 경우 23-20 = 3 → 2^3 = 8개까지 서브넷 파트를 할당가능 // 서브넷 마스크가 작을 수록 서브넷의 크기가 큼

// **DHCP**: 호스트가 자동으로 IP주소를 얻을 수 있게 하는 프로토콜 -> 서브넷 마스크, first hop router(gateway IP), 로컬 DNS 서버 이름, 주소 // 작동원리: 1 클라이언트는 DHCP 발견 메세지(67포트 – UDP 사용 IP 데이터그램)를 브로드캐스트 2 DHCP서버는 클라이언트가 받을 수 있도록 브로드캐스트(트랜잭션 ID-식별, 클라이언트에게 제공할 IP, 서브넷마스크, IP주소 임대 기간) 3 클라이언트는 DHCP 서버에게 IP주소를 사용하겠다고 브로드캐스트 4 서버는 클라이언트에게 ACK으로 응답 // **DHCP가 UDP를 사용해야 하는 이유**: 노드가 새로운 서브넷에 연결하고자 할 때마다 새로운 IP주소를 얻어야하기때문에, 이동 노드가 서브넷 사이를 이동할 때 원격 애플리케이션에 대한 TCP연결이 유지될 수 없다는 결점이 있음

// **NAT**: IPv4 주소는 한정되어 있음 따라서 서브넷을 기준으로 사설 IP와 공용 IP를 구분하고 라우터를 기준으로 변환시켜줌 이와 관련된 사설IP와 공용 IP의 매핑 정보를 NAT translation table에 저장함, 사설/공용 IP주소와 포트번호를 저장함 -> 한계: 포트번호를 참조해야하기 때문에 다른 계층을 침범함 // ISP에게는 단 하나의 ISP만 할당 받으면 됨. 서브넷 안에서는 다른 IP주소로 작동 // 외부에서 볼 때 사설 IP를 모르기 때문에 보안에 강함 // NAT table을 사용하기 때문에 외부 NAT 내부로 접근하지 못함 // 2^16개의 호스트를 가질 수 있음(16비트 포트 넘버가 있기 때문)

// **IPv6**: 헤더가 40바이트로 고정(fragmentation, checksum(이미 TCP가 무결성 보장), 옵션 제거) → **터널링을 통해 IPv4와 6를 함께 사용.** IPv4 라우터를 이동할때는 헤더를 붙여 IPv4로 만들어줌 // 송신과정: 1 v6노드는 v4를 지날때 페이로드가 됨 2 v4 데이터그램에 목적지 주소를 터널 수신측 v6노드로 적어서 터널의 첫번째 노드에 보냄 3 v4는 v4라우터(터널) 통과 3 터널을 나온 후, 다시 v6라우터 이동

// **라우팅 알고리즘**: 다익스트라, distance vector // intra – AS routing: 시스템 내에서 라우팅 컨트롤 가능 // inter AS routing: 단순 비용뿐만 아니라 사회, 정치적 내용도 연관이 깊어 라우팅 컨트롤이 불가능 // Intra AS routing에서 BGP, RIP, OSPF, IGRP사용

Link layer) frame(데이터 그램 다음 패킷 단위) // 무선링크와 같은 높은 오류율을 가진 링크에서 RDT사용 // **MAC address**사용: 링크 상 SRC/DEST 노드를 찾아가기 위한 주소

// flow control, error detection, error correction(많은 reduction이 필요) // 링크 레이어는 NIC(네트워크 어댑터)에 구현됨 – 시스템 버스를 통해 하드웨어 계층, 소프트웨어 계층 연결 // 수신측 노드는 에러가 없으면 데이터 그램을 추출해서 상위 계층으로 보냄 // EDC: 에러 수정을 위한 필드 // two dimensional bit parity: 에러 탐지 및 수정 가능 // 수신측에서 체크섬 계산 후 체크섬 필드와 비교

// **CSMA**: 캐리어 감지와 충돌 검출을 통해 충돌을 방지한다. // 흐름: NIC는 데이터그램을 받아서 frame으로 만든다. 채널이 혼잡하면 기다렸다가 보낸다. NIC는 다른 전송을 탐지하고 만약 있다면 중단하고 기다렸다가 다시 보낸다.

// 스위치는 링크 계층에서 동작한다 따라서 IP주소가 아닌 MAC주소를 사용한다. LAN은 인터넷이 아닌 링크를 이동하기 위해 설계한다. 만일 NIC가 IP주소를 사용하면, IP주소를 RAM에 저장하고 어댑터를 이동할 때마다 재구성해야한다. → 48비트

// 송신 어댑터는 프레임에 목적지 어댑터의 MAC주소를 넣고 랜상으로 전송한다. 프레임을 수신한 어댑터는 목적지 주소와 자신의 주소가 일치하는지 확인한다. 일치하면 프레임에서 데이터그램을 추출하여 네트워크 레이어로 이동한다. 일치하지 않으면 폐기한다. // MAC은 위치가 변하더라도 주소가 바뀌지 않는다.(portability 특성)

// **ARP**: IP주소와 MAC주소의 매핑 정보가 담긴 테이블, TTL동안 매핑 정보를 기억함 // 과정: 호스트A가 B에게 데이터 그램을 보내고 싶지만 MAC주소를 모름 → ARP쿼리 브로드캐스트 모든 노드가 ARP쿼리를 받음 → B는 ARP패킷을 받고 A에게 unicast → A는 B의 ip와 mac을 캐싱(ARP저장) ~ 이 과정은 자동임

// router는 IP주소를 알아야하지만 switch는 IP주소를 몰라도 된다.

시나리오

// DHCP를 사용해 ip주소를 받아야 함 – 먼저 DHCP 요청을 UDP 사용해(67, 68) 보냄 // 송신측: dhcp req -> udp segment -> datagram -> frame // DHCP 서버는 IP를 할당해주기 위핸 Ack을 보냄(ip addr, gatewat ip(first hor router) dns server name & ip, subnet mask) // 서버측: ack – udp – datagram – frame // 클라이언트: frame – datagram – udp – ack ~ IP주소와 DNS서버 주소 저장, forwarding table에 gatewat 정보 저장)

// DNS – DNS쿼리를 UDP를 통해 보내야 함 – 링크를 타고 나갈 때(link layer)에서 MAC주소를 모르므로 ARP 쿼리를 보냄 // DNS쿼리가 담긴 데이터그램이 게이트웨이 라우터에서 포워딩됨 DNS서버에 도착해서 demultiflexing → 이제 IP주소를 알아서 HTTP 요청가능(TCP)

// TCP 연결 – 3way handshake TCP SYN segment를 보냄, response ACK을 보냄, 연결 완료 // HTTP 요청을 보냄