03/23 - III. Transport Layer

UDP Socket = Socket Datagram
TCP Socket = Socket Stream

APP <- message -> App
TP. <- segment -> TP.
Network <- packet -> Network
Link <- frame -> Link
Physical <- ..... -> pyhsical

Multiplexing and demultiplexing(demux)
Transport Layer Services라면 제공해야 하는 것 중 하나. 컴퓨터 내부에 있는 프로세스에서 소켓을 쓰면 transport layer로 메시지가 내려오고, 하나의 segment를 만들어 하위 계층으로 내려준다. 이걸 내려오는 메시지마다 segment를 만든다고 해서 multiplexing이라고 하고, 리시버에 도착한 segment들을 APP의 메시지를 받아야 할 프로세스들에 정확히 배분하는 작업을 demultiplexing이라고 한다. 근데 이걸 어떻게 판단? segment의 header에 적힌 정보를 가지고 판단한다.

1. Socket
OS에서 제공하는 API의 일종
애플리케이션 프로세스(서버와 클라이언트)의 통신
Application은 transport에서 제공하는 통신 방식을 사용할 수밖에 없는데, 이 중에 TCP, UDP가 구현되어 있다. 즉 소켓을 사용하되 두 가지 방식으로 사용할 수 있다.

Sockets API and process (TCP)

socket()
Web Server가 TCP 소켓을 연다.
socket id를 return한다.

bind()
소켓을 특정 포트에 바인드한다.

listen()
서버는 요청을 받아야 하므로 passive state로 지정한다.
backlog 파라미터로 동시에 몇 개까지 처리할지 정할 수 있다.

accept()
서버가 요청을 받을 준비를 함
요청이 들어오면 리턴되는데, 파라미터로 클라이언트의 ip와 포트넘버가 전달되면서 이를 사용할 수 있게 된다.

socket()
클라이언트에서 요청을 보내려고 소켓을 엶

connect()
서버와 연결, 이렇게 되면 TCP three-way handshacking

write() / read()
write is blocking: returns only after the message sent

close()
소켓을 닫는다. 이때 소켓 포트 번호를 release한다.
Ctrl+C 등 rough exit에 대비해,
socket을 release하는 처리를 해 주면 좋다(해당 포트가 비게 된다) 이거 안 하면 프로세스는 죽더라도 포트는 들고 있는 현상이 벌어짐. 몇 분 후에는 포트가 사라지지만...

서버에서 소켓을 만들고 포트에 bind-listen-accept하면,
리턴되지 않는 상태로 block하고 있다가 클라이언트에서 요청이 들어오면 동작을 수행하게 된다.
클라이언트 또한 소켓을 만들고 서버와 connect를 한 후, 서버와 서로 write, read를 하고 close하면서 연결을 종료한다.

UDP segment format
Headers
Source Port# (16bits)
Dest. port# (16bits)
length (16bits)
checkSum (16bits)
Payload

Multiplexing and demultiplexing

Transport Layer Services라면 제공해야 하는 것 중 하나. 컴퓨터 내부에 있는 프로세스에서 소켓을 쓰면 transport layer로 메시지가 내려오고, 하나의 segment를 만들어 하위 계층으로 내려준다. 이걸 내려오는 메시지마다 segment를 만든다고 해서 multiplexing이라고 하고, 리시버에 도착한 segment들을 APP의 메시지를 받아야 할 프로세스들에 정확히 배분하는 작업을 demultiplexing이라고 한다. 근데 이걸 어떻게 판단? segment의 header에 적힌 정보를 가지고 판단하는데, UDP의 경우 dest. IP와 dest. portNo를 사용해서 어떤 소켓으로 올릴지 결정하고, TCP의 경우 Source IP, Source portNo까지 고려해서 어떤 소켓으로 올릴지 결정한다.
TCP의 경우 UDP와 다르게, 네 가지 정보 중 하나라도 다르면 다른 소켓으로 demux된다. (connection-oriented) 즉 (웹 서버의 입장에서) TCP는 클라이언트별로 다른 소켓이 만들어지고, UDP는 하나의 소켓에서 다 받는다.
UDP가 아무것도 안 하는 것 같지만... multiplexing, demultiplexing, error checking이라는 기본적인 것은 한다.

Principles of Reliable Data Transfer

UDP와 같이 unreliable한 환경에서는 패킷에서 에러가 나거나 패킷이 유실된다. TCP에서는 reliable하게 데이터를 제공하기 위해 어떤 일을 하는가?
reliable한 프로토콜을 디자인해 본다고 생각하자.
1. 한 번에 패킷 하나만 보낸다
2. 수신이 확인되면 다음 패킷을 보낸다
다음 케이스에 대해 패킷을 완벽히 보내려면 어떻게 해야 하는가?
case 1) channel이 완벽한 경우 – 문제 없음
case 2) 패킷 에러 발생 가능성이 있는 경우
(1) checkSum 헤더에서, 에러가 있는지를 받는 쪽에서 판단
(2) 보내는 쪽으로 피드백을 줘야 함
Acknowledgement(ACKs) – 잘 받음
Negative ACKS(NAKs) – 뭐라고요?
(3) sender는 NAK를 받으면 retransmission
– ACK에 에러가 있다면? 이게 NACK인지 아닌지 판단을 할 수 없게 됨. 일단 그냥 다시 보내는 방법이 있음. 그러면 receiver는 두 번째로 받은 패킷이 첫 번째의 duplicated인지 판단할 길이 없음. 이 문제는 패킷에 번호를 붙여 해결한다. (sequence number)
– 그러면 seq#는 어디에 들어가는가? 헤더에 넣어야 할 텐데 이게 계속되면 헤더가 너무 크지 않니?(overHead)

– field의 수를 최소화하고 싶은 상황에서, 이런 프로토콜이라면 seq#는 두 개면 충분하다. 한 번에 패킷이 한 개만 오기 때문. 즉 하나와 구분되는 다른 숫자 하나면 된다. (1bit)
– seq0을 보냈는데 NAK가 오면 다시 seq0을 보내면 된다.
– 그러면 receiver는 seq0이 duplicated인지 구분할 수 있고, 같은 패킷이 왔으니 버린다.
– Sender에서 packet(0)을 보내면, receiver가 ACK에 받은 패킷 번호를 적어서 보내는 방법이 있다. 에러 패킷을 받아도 receiver는 그전에 가장 마지막으로 받은 것 중 정상적인 패킷번호를 써서 돌려보낸다.(NAK free protocol)

case 3) case 2 + 메시지 자체가 유실될 수 있는 경우
– 메시지가 유실되면 sender는 아무것도 모름 –> Timer!!!
– 타이머가 다 가도록 피드백이 없으면, sender는 재전송
– 그럼 타이머 몇 초? 답이 없음..(reasonable) 제한시간이 짧으면 사실 피드백 자체가 오래걸리는건데 중복패킷을 계속 보내서 네트워크에 오버헤드가 갈 수 있고... 너무 길면 반응 자체가 너무 늦어질 수 있음

이를 바탕으로 한 최종 디자인
(1) 패킷에는 seq#를 붙여서 보냄
(2) receiver는 ACK만 보내고, 받은 seq#를 붙여서 보냄
만약 NAK을 보내야 하면 마지막으로 받은 seq#를 보냄
(3) sender는 타이머가 있어 피드백이 오지 않으면 재전송
(4) receiver는 받은 중복 패킷을 버리고 ACK + seq#
(5) ACK가 유실되었다면, 이것도 똑같이 sender가 재전송
(6) receiver는 받은 중복 패킷을 버리고 ACK + seq#
(7) 피드백이 타임아웃보다 늦었을 때에도 재전송
(8) receiver는 받은 중복 패킷을 버리고 ACK + seq#

문제점
– 피드백 올 때까지 아무것도 안 한다는 문제가 있음. 그래서 실제로는 일단 쏟아붓고, 피드백도 한번에 받는다.