Chapter 3
Transport Layer

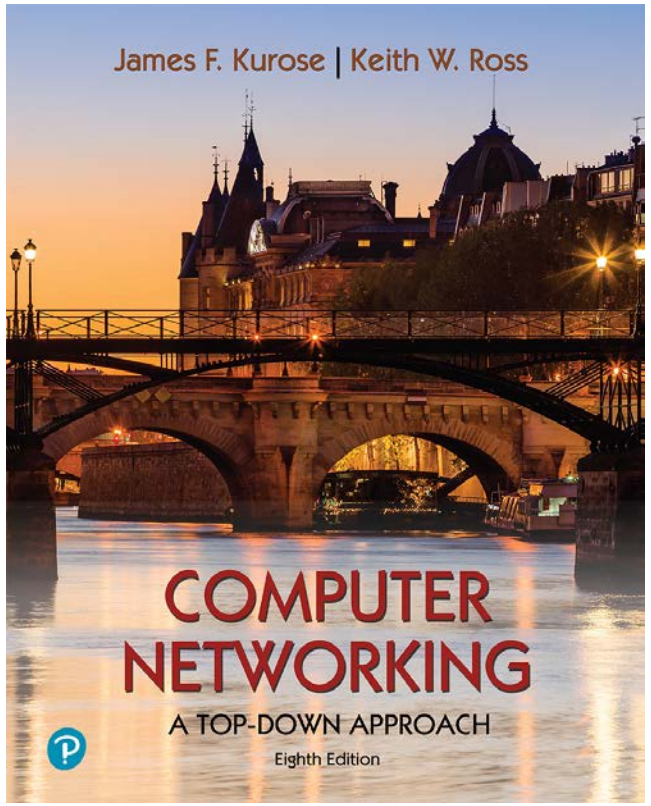
A note on the use of these PowerPoint slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

For a revision history, see the slide note for this page.

Thanks and enjoy! JFK/KWR



Computer Networking: A Top-Down Approach
8th edition
Jim Kurose, Keith Ross
Pearson, 2020

UDP: User Datagram Protocol

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

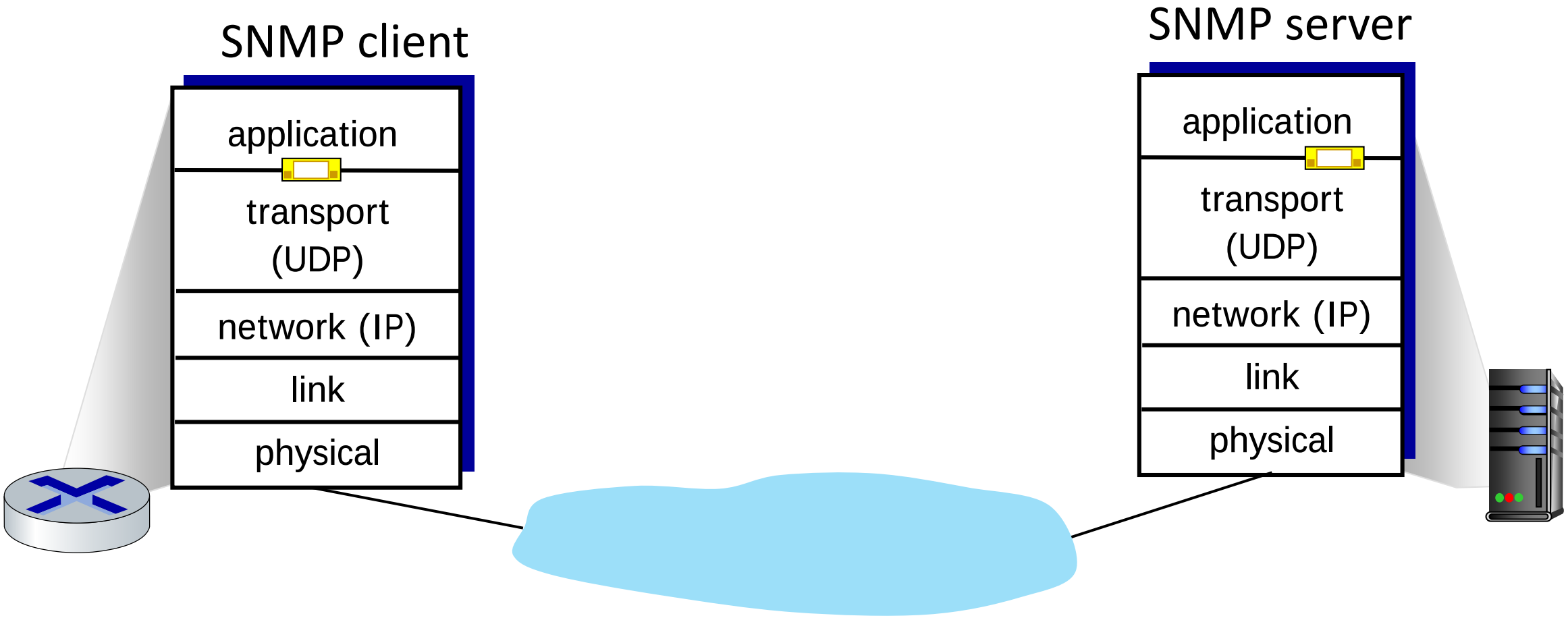
- no connection establishment (which can add RTT delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control
  - UDP can blast away as fast as desired!
  - can function in the face of congestion

-
UDP: پروتکل UDP یک سرویس حداقلی به اپلیکیشن میده و سرویس best effort لایه شبکه رو در اختیار اپلیکیشن قرار میده
best effort: ینی اگر بسته ای در شبکه گم بشه یا اگر ترتیب بسته ها بهم بخوره UDP کاری در این مورد انجام نمی ده
به همین جهت UDP احتیاجی به connection نداره ینی connectionless چون هیچ handshaking برای کنترل و مدیریت این ها انجام نمیده
پس هر سگمنت UDP مستقلا به شبکه داده میشه و در مقصد از شبکه گرفته میشه
یکسری از این ها اپلیکیشن هایی هستن که ترتیب براشون مهم نیست مثل DNS , SNMP و یکسری از اپلیکیشن ها هم اپلیکیشن هایی هستن که مکانیزیم هایی که TCP به کار می بره برای ترتیب نمی تونه تحمل بکنه بخاطر اینکه اون ها مستلزم تاخیر اضافه تر هستن و این اپلیکیشن ها، اپلیکیشن هایی هستن که تاخیر رو نمی تونن تحمل کنن
توی این موارد استفاده از UDP به این معناست که ترتیب و تکلیف بسته های گم شده رو اپلیکیشن خودش بعدا باید روشن بکنه

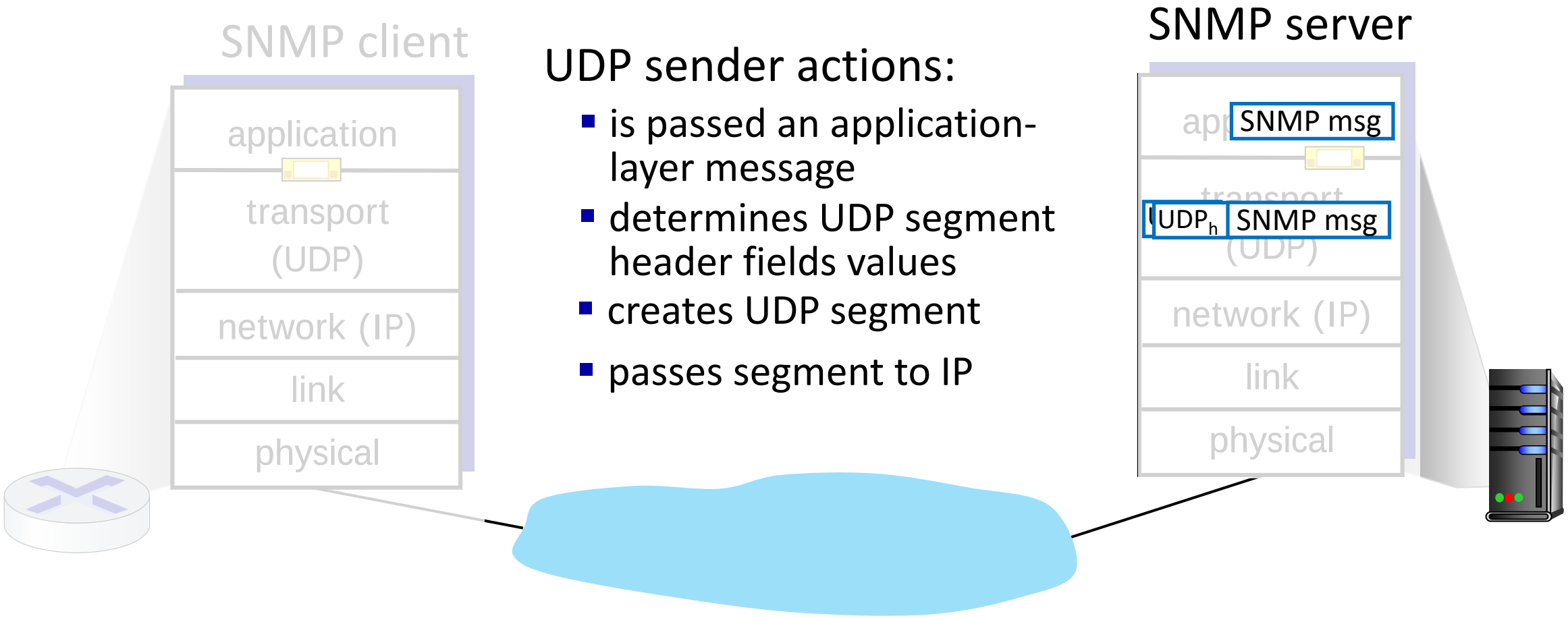
UDP: User Datagram Protocol

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
  - HTTP/3
- if reliable transfer needed over UDP (e.g., HTTP/3):
  - add needed reliability at application layer
  - add congestion control at application layer

UDP: Transport Layer Actions



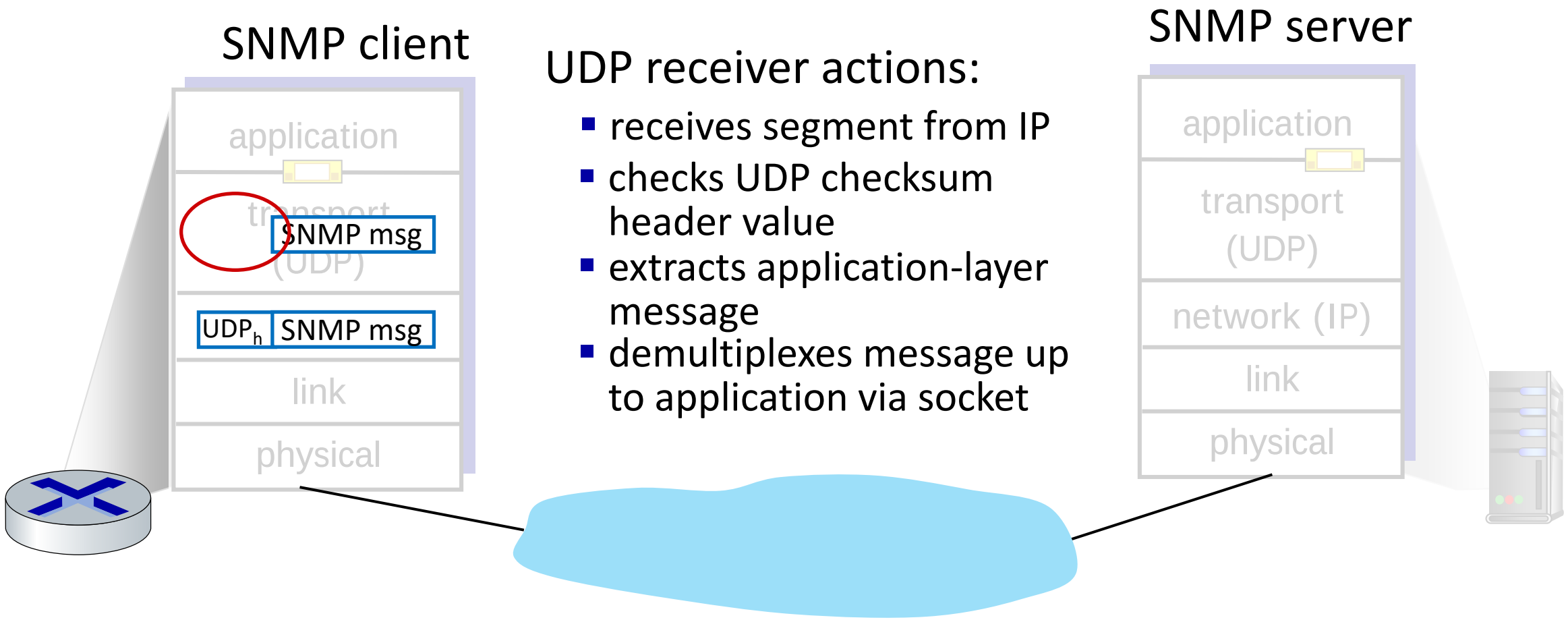
UDP: Transport Layer Actions



UDP sender actions:

- is passed an application-layer message
- determines UDP segment header fields values
- creates UDP segment
- passes segment to IP

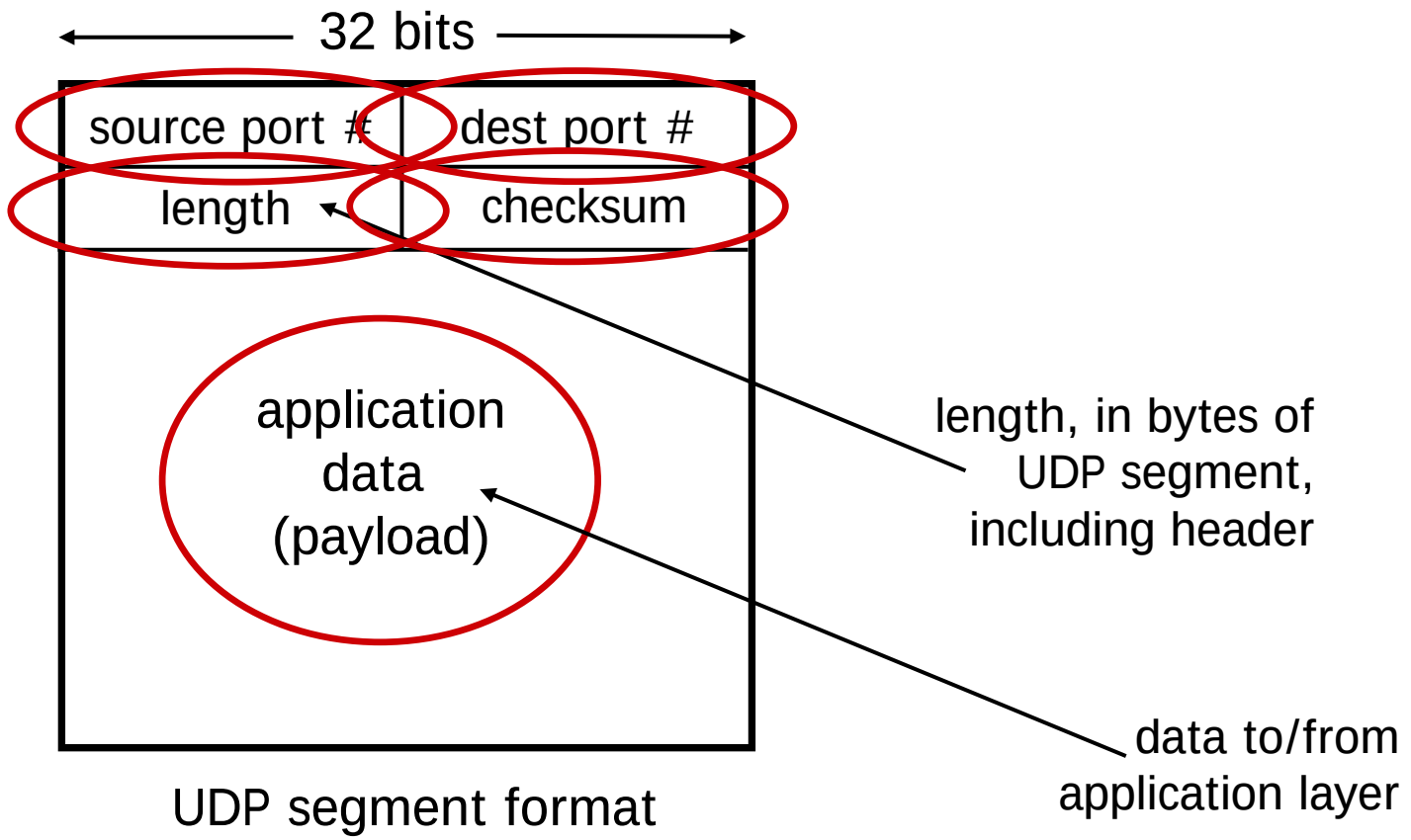
UDP: Transport Layer Actions



UDP receiver actions:

- receives segment from IP
- checks UDP checksum header value
- extracts application-layer message
- demultiplexes message up to application via socket

UDP segment header



UDP segment format

توی این اسلاید فرمت بسته UDP می بینیم:
source port# و dest port#: که برای مشخص کردن اپلیکیشن مبدا و مقصد استفاده میشه

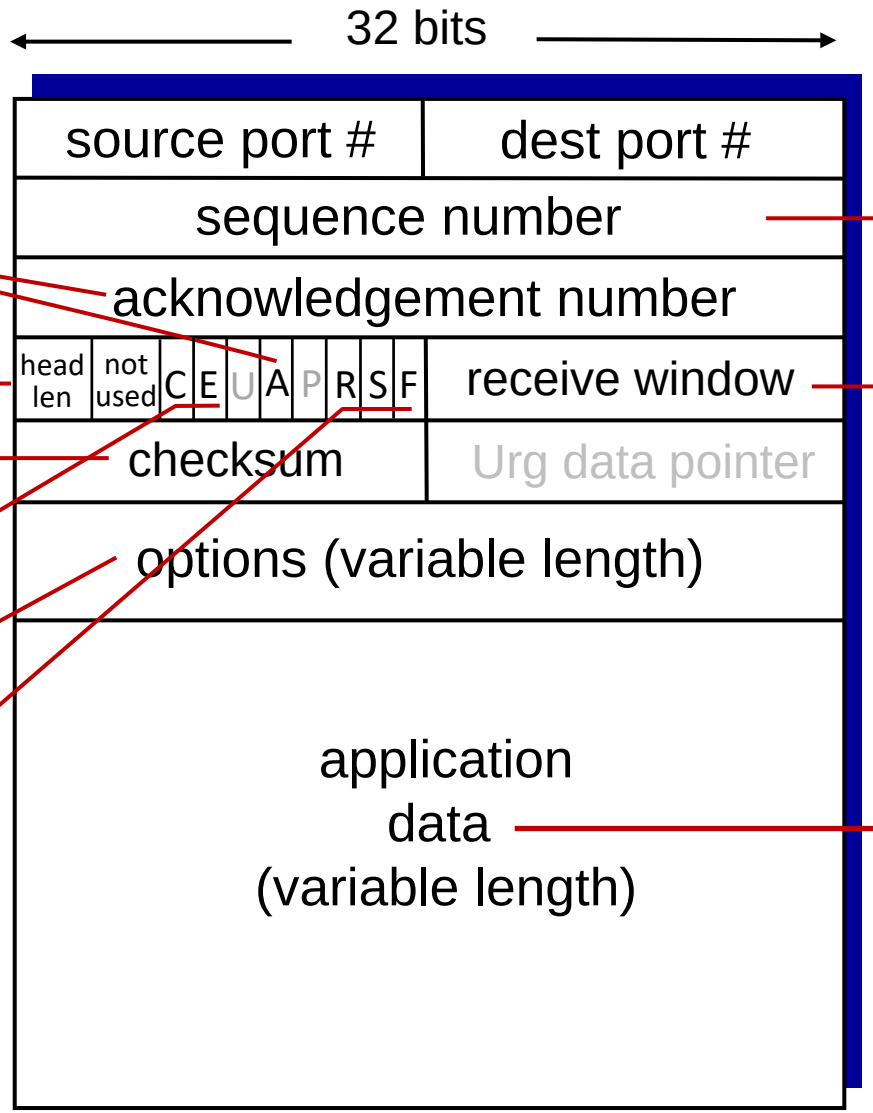
اندازه هدر UDP خیلی کوچیک میشه و این over head هدر رو خیلی کم میکنه و پردازش خیلی محدودی داره و سریعتر بسته می تونه ارسال بشه
UDP هیچ کنترلی در مورد congestion در شبکه انجام نمی ده پس به راحتی می تونه عرض باند شبکه رو استفاده بکنه

TCP: overview RFCs: 793,1122, 2018, 5681, 7323

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - no “message boundaries"
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- cumulative ACKs
- pipelining:
  - TCP congestion and flow control set window size
- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

-
در مقابل TCP یک پروتکل point-to-point است که بین دو نقطه انتهایی عمل میکنه و connection-oriented است ینی بین این دوتا نقطه کانکشن ایجاد میشه با handshaking و توی این کانکشن اطلاعات در هر دو جهت ارسال میشه
این پروتکل انتقال اطمینان و با ترتیب درست اطلاعات در اون جهت رو ینی بین مبدا و مقصد رو تنظیم می کنه پس به این ترتیب یک pipelined از بسته هایی که از مبدا تا مقصد ارسال میشن ایجاد میشه

TCP segment structure



ACK: seq # of next expected byte; A bit: this is an ACK

length (of TCP header)

Internet checksum

C, E: congestion notification

TCP options

RST, SYN, FIN: connection management

segment seq #: counting bytes of data into bytestream (not segments!)

flow control: # bytes receiver willing to accept

data sent by application into TCP socket

فرمت tcp:

TCP sequence numbers, ACKs

Sequence numbers:

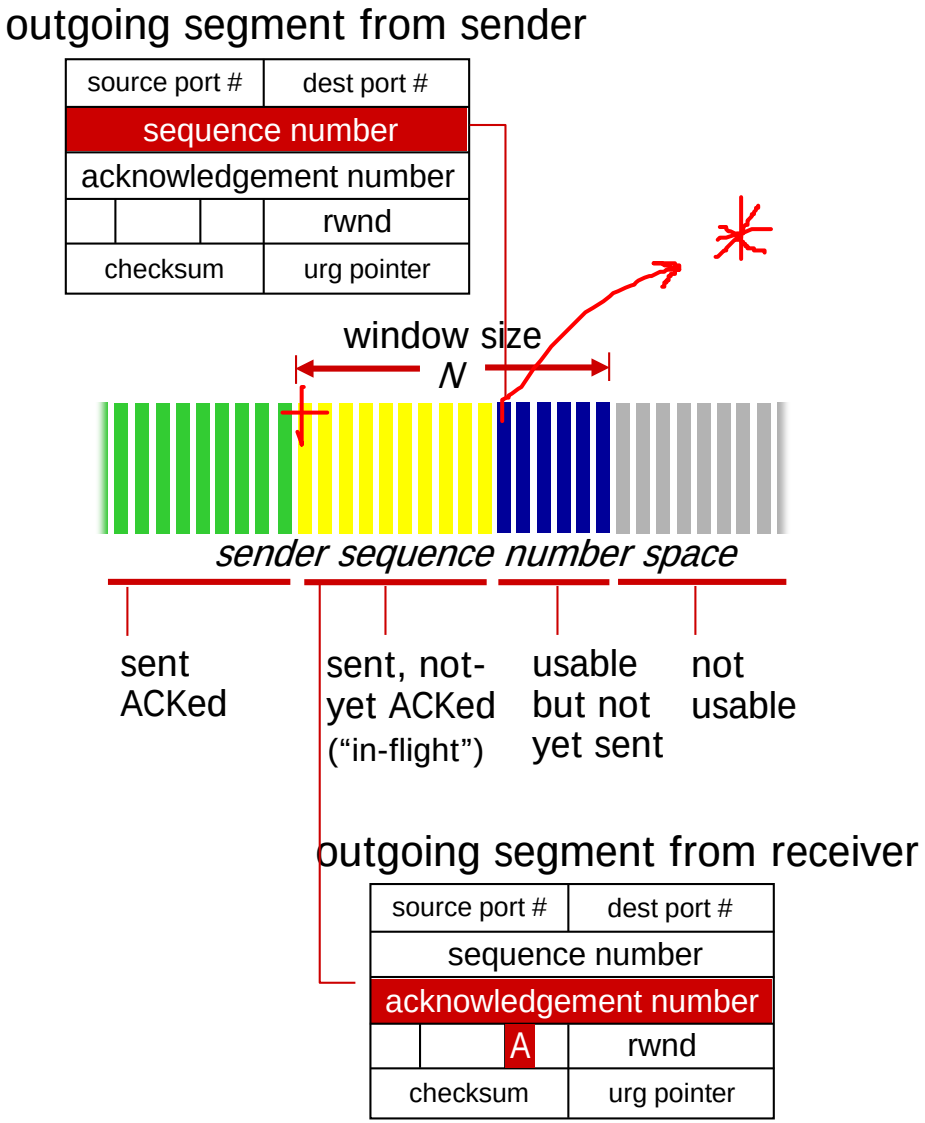
- byte stream "number" of first byte in segment's data

Acknowledgements:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- *A:* TCP spec doesn't say, - up to implementor

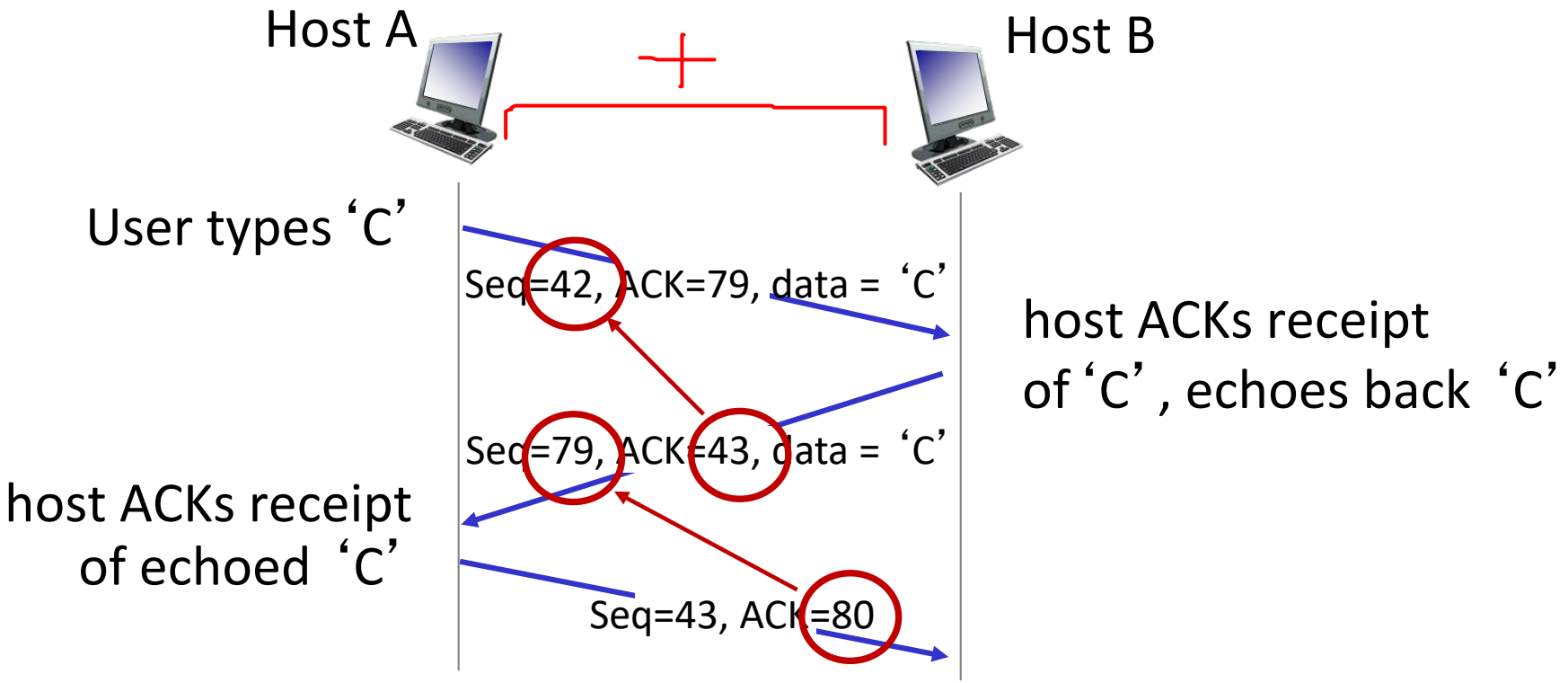


-
برای اطمینان از رسیدن بسته ها و ترتیب درست اون ها از مکانیزیم sliding window استفاده میکنیم
در مکانیزیم sliding window سمت فرستند یک پنجره داره به نام پنجره ارسال و متناسب با اون در سمت گیرنده هم یک پنجره دریافت داریم
این پنجره رو می تونیم اینطوری تصور بکنیم خونه خونه هایی داره که توی هر خونه یک بایت می تونه ذخیره بشه این خونه ها شماره گذاری میشن و هر خونه نسبت به خونه قبلی شمارش یک واحد اضافه تر میشه
بایت هایی که از اپلیکیشن دریافت میشن به ترتیب در این خونه ها چیده میشن و هر بایتی شماره پیدا میکنه و وقتی تعداد این بایت ها به اندازه کافی شد این بایت ها در قالب یک سگمنت با اضافه شدن هدر به گیرنده ارسال میشه
سگمنتی که به این ترتیب ساخته میشه و ارسال میشه یک sequence number بهش نسبت داده میشه در فیلد sequence number هدرش نوشته میشه

این sequence number در واقع شماره اولین بایتی است که داخل سگمنت گذاشته میشه
این سگمنت رو وقتی گیرنده دریافت میکنه اونو Acknowledgement میکنه و به این ترتیب ما مطمئن میشیم که بسته در مقصد دریافت شده و با استفاده از همین sequence number ترتیب بسته ها هم در مقصد چک میشه و بسته ها با ترتیب درست به اپلیکیشن های سمت گیرنده داده میشن
وقتی که یک سگمنت Acknowledgement میشه تمام بایت هایی که داخل اون هستن Acknowledgement میشن و وقتی که یک سگمنت ارسال میشه بایت های اون داخل پنجره باقی می مانن تا زمانی که Acknowledgement بشن و این برای این منظور است که اگر اطمینان حاصل نشد که بسته به مقصد رسیده بسته رو مجددا بفرستیم تا ازمانی این اطمینان به وجود بیاد ینی تا زمانی که ACK اون رو دریافت بکنیم ولی بایت هایی که ارسال شدن بعد از اینکه Acknowledgement شدن از پنجره خارج میشن برای این منظور ابتدای پنجره به اندازه تعداد بایت هایی که Acknowledgement شدن جلو می رن
در پروتکل sliding window اندازه کل پنجره : ینی تعداد کل بایت هایی که می تونن ارسال بشن و ما منتظر Acknowledgement اون ها می تونیم بمونیم تا بیاد و اینا محدوده پس اندازه پنجره ارسال محدوده که اینجا با N نشون داده شده و به همین جهت وقتی که ابتدای پنجره به اندازه سگمنتی که Acknowledgement شده جلو بره انتهای اون هم جلو می ره و به این ترتیب بایت های جدیدی داخل پنجره قرار میگیرن
بایت های توی پنجره ارسال رو به چند دسته تقسیم می کنیم:
بایت هایی که داخل پنجره هستن و بایت هایی که خارج پنجره هستن
بایت های خارج:
بایت های سبز بایت هایی هستن که قبلا توی پنجره بودن و ارسال شدن و Acknowledgement شدن و از پنجره شدن
بایت های خاکستری بایت هایی هستن که هنوز وارد پنجره نشدن ینی هنوز نمی تونیم اونارو استفاده بکنیم

بایت های داخل:
بایت زرد: اون هایی هستن که ارسال شدن و منتظر Acknowledgement شون هستیم پس ابتدای پنجره اینجاست + و این ها بایت هایی هستن که از ابتدای پنجره ارسال شدن و هنوز ACK نشدن و بایت های ابی بایت هایی هستن که هنوز ارسال نشدن این ها بایت هایی هستن که از اپلیکیشن گرفته شدن ولی تعدادشون به اندازه سگمنت نشده که ارسال بشن یا هنوز از اپلیکیشن گرفته نشدن ولی جاش هست که هنوز بیان و ارسال بشن
به این ترتیب اولین بسته ای که بعدا ارسال خواهد شد Sequence numbersاش Sequence numbers بایت * خواهد بود و ACK که بیاد کدوم بایت رو Acknowledgement خواهد کرد و کدوم بسته رو؟ بسته ای که اولین بایتش + است و Sequence numbers --> + و همه بایت هایی که توی بسته اون هستن رو ACK میکنه

TCP sequence numbers, ACKs



simple telnet scenario

مثال:
فرض میکنیم در هاست A یک کاراکتری تایپ میشه و یک سگمنت ساخته میشه و ارسال میشه و توی پنجره ارسال این شماره اش 42 است در نتیجه Sequence numbers بسته ارسالی شماره اش میشه 42
این بسته وقتی که به گیرنده برسه گیرنده اینو Acknowledge خواهد کرد و Acknowledge اون 43 است ینی 43 مشخص کننده بایت بعدیه که سمت گیرنده منتظرش است و به این ترتیب در سمت فرستند از 42 تا 43 هر چی هست که اینجا یک بایت بیشتر نیست Acknowledge میشه
در TCP مکانیزیم sliding window یک طرفه است ینی بسته ها از یک سمت ارسال میشن و در سمت مقابل Acknowledge میشن
در سمت مقابل یک همچین sliding window داریم که بسته رو ارسال میکنه و این ها Acknowledge هاشون رو توی بسته دیتایی که به سمت مقابل می فرستن قرار میدن برای همین بسته ای که از این سمت می ره Acknowledge هم توش هست
وقتی که این بسته رسید پنجره می ره جلوتر و بسته بعدی ارسال میشه و توی اون هم بسته ای که از اون طرف دریافت شد Acknowledge میشه
نکته مهمی که اینجا باید بهش توجه کرد فاصله زمانی است که طول می کشه تا بسته ای ارسال بشه و بعد Acknowledge اون از طرق مقابل دریافت بشه به این فاصله زمانی ما میگیم RTT ینی یک زمان رفت و برگشت
چرا رفت و برگشت؟ چون بسته ما از سمت فرستنده باید به گیرنده بره و برسه و بعد Acknowledge اون توسط اون ارسال بشه و بعد این طرف دریافت بشه پس یک رفت و برگشت اتفاق می افته
توی این شکل فرض می کنیم زمان به سمت پایین جلو می ره ینی هرچی زمان می گذره ما میایم پایین تر و این هم + فاصله رو نشون میده ینی از فرستنده تا گیرنده
وقتی بسته فاصله ای از فرستنده تا گیرنده رو طی میکنه این زمانی طول می کشه و بعد دوباره بسته ای که از اونور ارسال میشه تا بیاد این طرف دریافت بشه یک زمانی طول می کشه و مجموع این ها میشه زمان رفت و برگشت
این زمان رفت و برگشت یک پارامتر مهمی توی TCP هستش

TCP round trip time, timeout

Q: how to set TCP timeout value?

- longer than RTT, but RTT varies!
- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss

Q: how to estimate RTT?

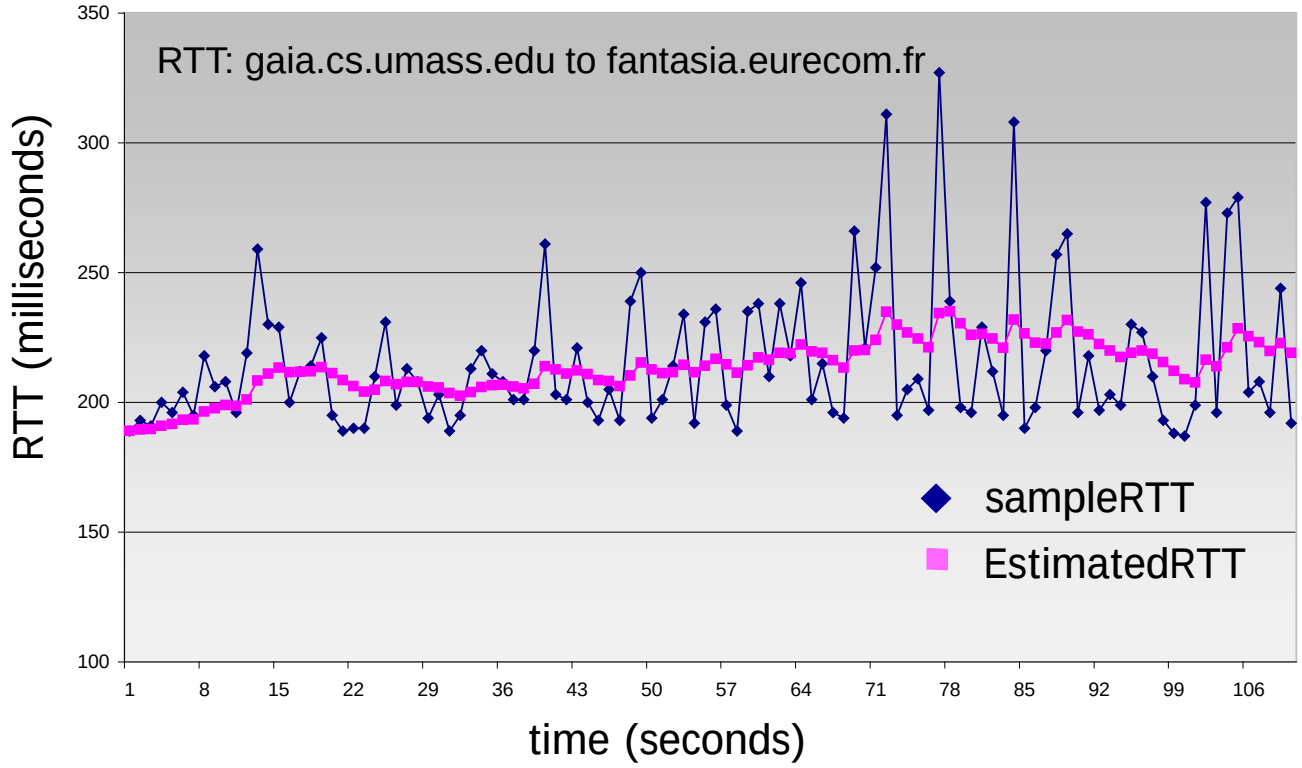
- `SampleRTT`: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- `SampleRTT` will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current `SampleRTT`

-
یکی از دلایلی که این round trip time یا RTT اهمیت پیدا میکنه مسئله timeout است
timeout چی هست؟
گفتیم ما یک بسته ای رو از سمت فرستنده به گیرنده می فرستیم منتظریم ACK اون می شیم تا مطمئن بشیم که این در سمت دیگر دریافت شده و اگر این ACK دریافت نشه و نرسه بدین ترتیب ما این اطمینان رو پیدا نخواهیم کرد و اگر اطمینان پیدا نکنیم که بسته توسط اون طرف دریافت شده باشه بسته رو باید دوباره بفرستیم
منتها ما کی به این تصمیم می رسیم که مطمئن بشیم که بسته به اون طرف نرسیده؟ ایا بسته نرسیده یا ACK تاخیر داره؟ ما یک مدتی صبر میکنیم تا مطمئن بشیم که ACK بخاطر شلوغی شبکه و تاخیر شبکه تاخیر پیدا کرده یک حدی برای این انتظار در نظر می گیریم و اگر تا این مدت ACK دریافت نشد فرض میکنیم که بسته نرسیده و اونو دوباره می فرستیم این مدت رو با یک تایمر ست میکنیم و اسمش رو می ذاریم timeout و عملا وقتی که تایمرمون timeout بکنه بسته رو مجددا می فرستیم
زمان این تایمر خیلی مهم است چون اگر ما تایمر رو کوتاه انتخاب کرده باشیم واقعا ممکنه در شرایطی که ACK می تونه با تاخیر برسه ما بسته رو مجدد ارسال بکنیم بدون اینکه نیاز به اون داشته باشیم و اگر خیلی طولانی اونو ست کرده باشیم باعث میشه وقت هایی هم که واقعا بسته نرسیده و ما باید بفرستیم اینو داریم با تاخیر می فرستیم پس این زمان مهم است
پس به همین خاطر در سمت فرستنده، فرستنده TCP میاد RTT بسته هایی که ارسال می کنه رو اندازه می گیره و همیشه یه چیزی از شبکه رو به دست میاره؟؟؟ ینی در وضع موجود یک بسته که می ره ینی مثلا یک بسته ارسال میشه و ACK اش برگردونده میشه در این لحظه تایمری رو ست میکنه برای اندازه گیری RTT و در این لحظه اونو قطع می کنه و می بینه چقدر شده به این ترتیب یک سمپل از RTT به دست میاد و سمت فرستنده این RTTهایی که اندازه گیری می کنه رو متوسط گیری میکنه و به این ترتیب یک مقدار نمونه برای RTT در شرایط فعلی به دست میاد

TCP round trip time, timeout

$$\mathbf{EstimatedRTT = (1-\alpha)*EstimatedRTT + \alpha*SampleRTT}$$

- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$



-
برای محاسبه timeout:
از این فرمول می ره
نکته: سمت فرستنده RTT رو اندازه گیری می کنه

TCP round trip time, timeout

- timeout interval: **EstimatedRTT** plus "safety margin"
  - large variation in **EstimatedRTT:** want a larger safety margin

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT

"safety margin"

- **DevRTT**: EWMA of **SampleRTT** deviation from **EstimatedRTT**:

```
DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|
```

(typically, β = 0.25)

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

TCP Sender (simplified)

event: data received from application

- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unACKed segment
  - expiration interval: **`TimeOutInterval`**

event: timeout

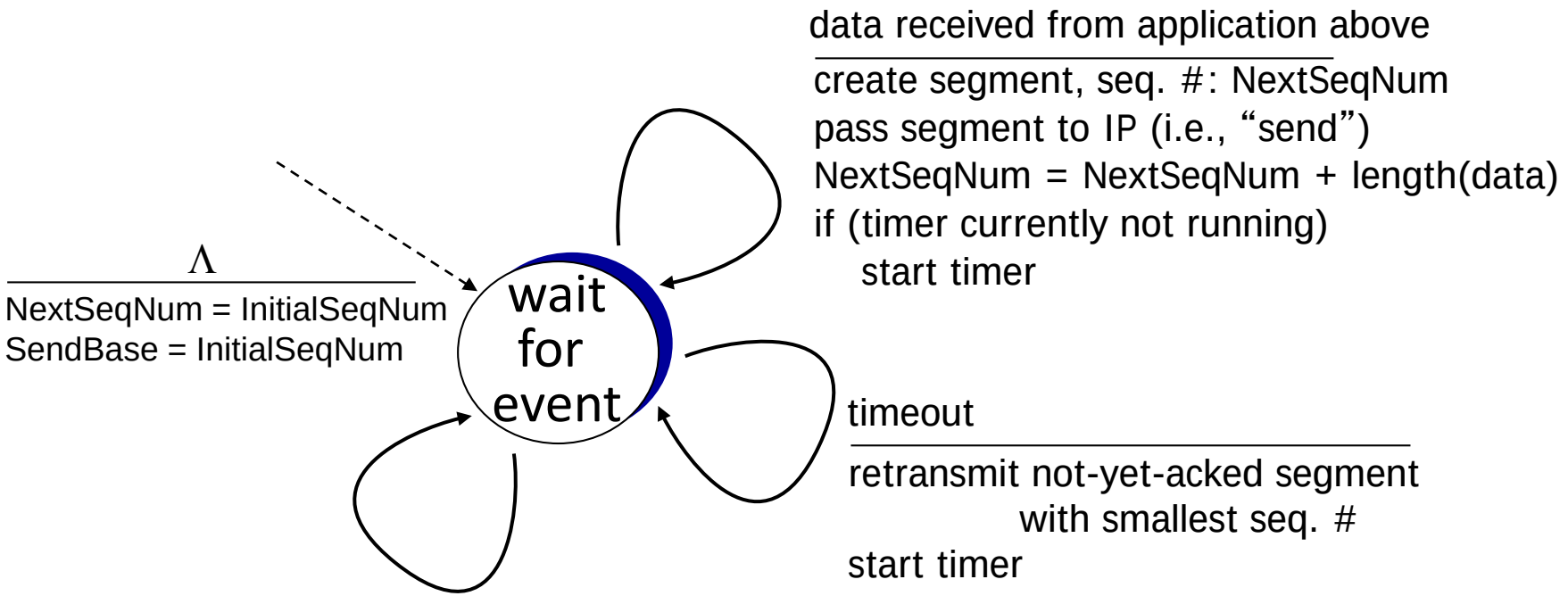
- retransmit segment that caused timeout
- restart timer

event: ACK received

- if ACK acknowledges previously unACKed segments
  - update what is known to be ACKed
  - start timer if there are still unACKed segments

-
پروتکل در مورد سمت فرستنده:
اگر دیتا از اپلیکیشن داده بشه به پروتکل TCP برای ارسال --> این دیتا جمع میشه و وقتی که به اندازه کافی بود یک سگمنت تشکیل میشه و Sequence numbers یا seq# روش زده میشه و ارسال میشه
و اگر این سگمنت اولین سگمنت توی پنجره است که داره ارسال میشه ینی هیچ سگمنت قبلیه توی پنجره در حال انتظار نیست برای ACK در اینصورت تایمر ست میشه
و event که می تونه اتفاق بیوفته timeout است و اگر timeout اتفاق بیوفته همین سگمنت دوباره ارسال میشه و تایمر دوباره ست میشه و اگر این سگمنت اولین سگمنت توی پنجره نباشه و اگر timeout اتفاق بیوفته قدیمی ترین سگمنتی که توی پنجره بوده دوباره ارسال میشه
و این قدیمی ترین سگمنت = همون اولین سگمنت میشه که این قدیمی ترین سگمنت توی پنجره باید تایمر داشته باشه
event بعدی که می تونه اتفاق بیوفته اینه که ack دریافت بشه از سمت گیرنده --> اگر Ack دریافت بشه همه بسته هایی Ack نشه تا این ack که دریافت شده acknowledges میشن ینی از پنجره خارج میشن و پنجرمون اپدیت میشه به این ترتیب ابتدای پنجره جلو میره تا جایی که acknowledges شده و اگر بعد از اون هنوز بسته ای است که acknowledges نشده در این صورت تایمر مجددا ستاپ میشه و اگر بسته دیگری نباشد لزومی برای ستاپ کردن تایمر نیست

TCP sender (simplified)



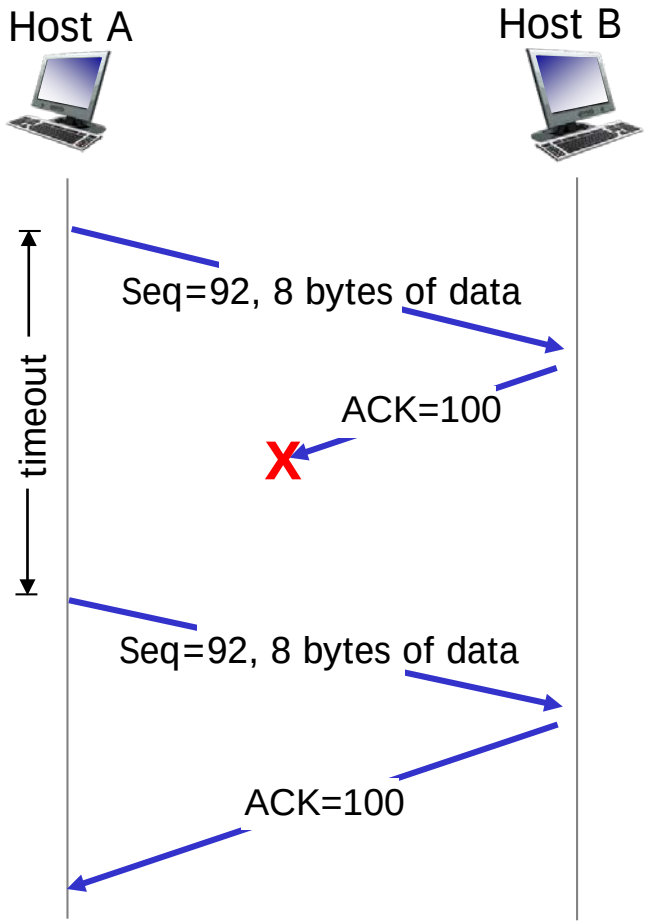
این استیت ماشین خلاصه حالت هایی است که الان گفتیم

TCP Receiver: ACK generation [RFC 5681]

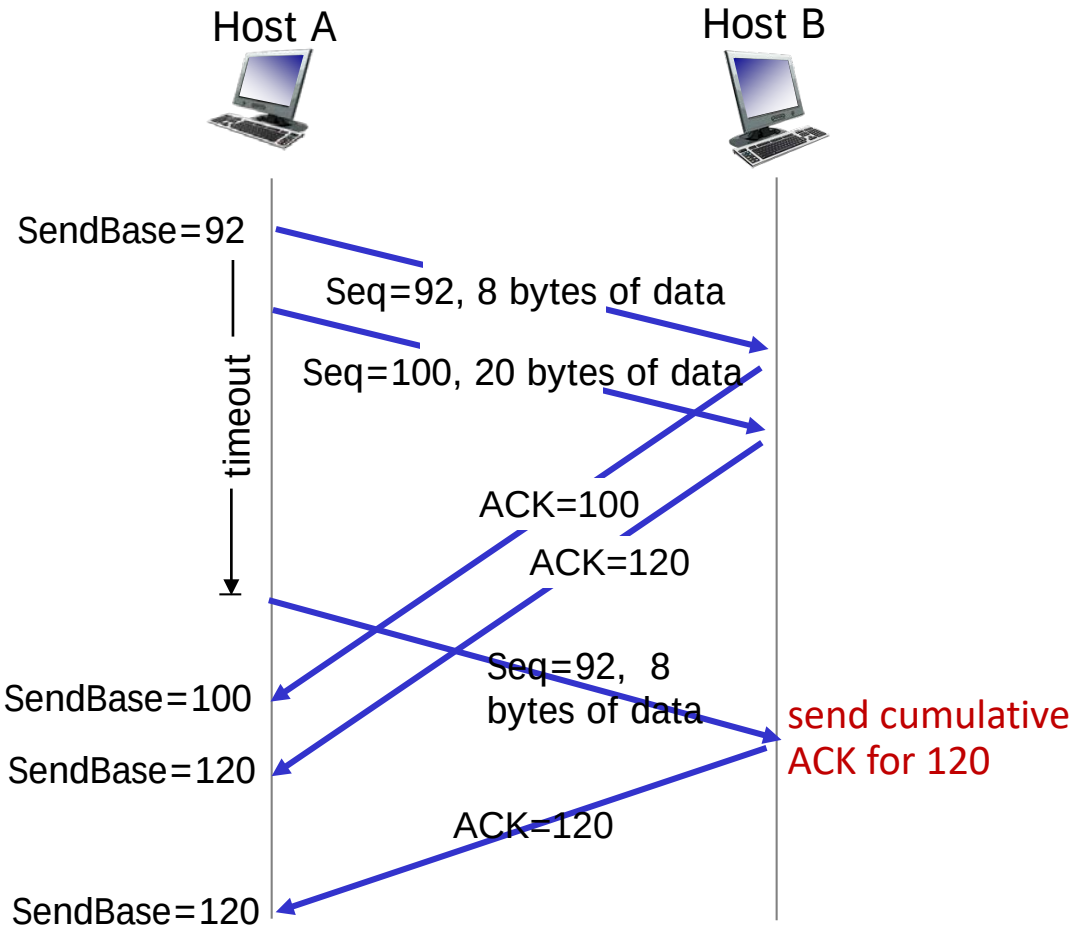
event at receiver	*TCP receiver action*
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *duplicate ACK*, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

در سمت گیرنده چه اتفاق هایی می افته؟
وقتی یک بسته با ترتیب درست دریافت بشه --> الزاما بلافاصله ack داده نمیشه بلکه یک مقداری صبر می کنه سمت گیرنده
و بسته بعدی که دریافت شد اونوقت ack برای دوتاش می فرسته برای اینکه در تعداد Ack ها صرفه جویی بکنه
اگر یک بسته ای خارج از ترتیب دریافت بکنه در این صورت اخرین ack اش رو تکرار میکنه یعنی duplicate ack می فرسته چرا؟ چون یک بسته خارج از ترتیب نشون میده که یک بسته دیگری قبل از این بوده که نرسیده
سمت گیرنده در پروتکل tcp الزامی نداره که بسته های خارج از ترتیب رو نگهداری بکنیم ولی به هرحال می تونیم این کار رو هم انجام بدیم و اگر این کارو انجام داده باشه یک فاصله ای اینجا وجود داره که بسته هایی هستن که هنوز نرسیدن البته بسته خارج از ترتیب رو نمی تونه به اپلیکیشن بده بلکه نگهداری میکنه حالا اگر بسته ای توی این فاصله اومد که این فاصله رو پر بکنه این هم اضافه میکنه اگر این بسته ابتدای gap رو پر بکنه این ack می ده وگرنه duplicate ack مجددا ارسال میشه

TCP: retransmission scenarios



lost ACK scenario

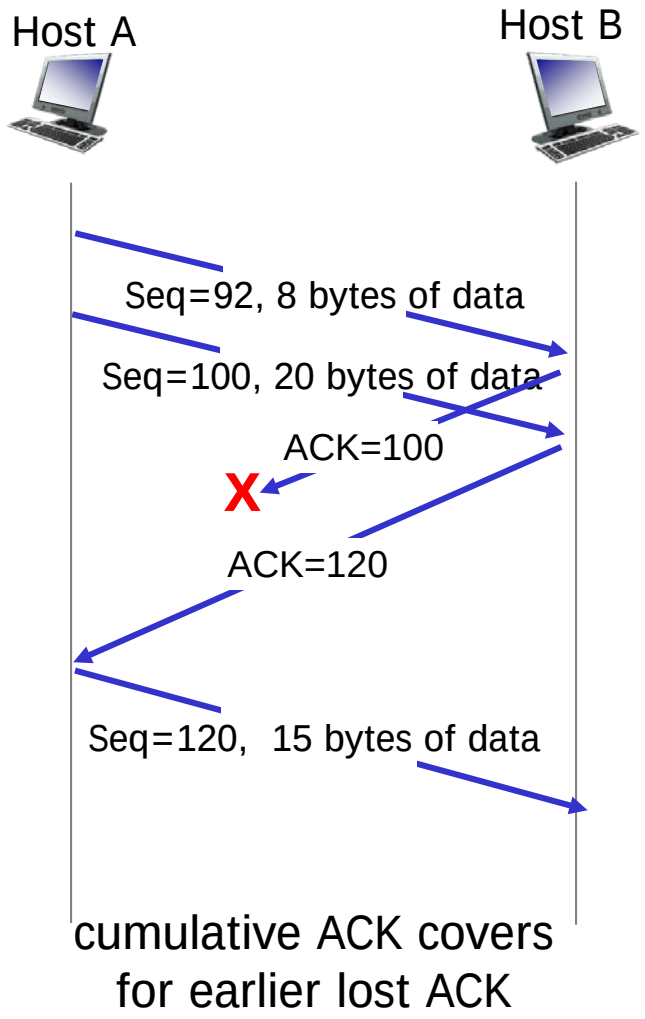


premature timeout

سناریو:
اگر یک سگمنت ارسال بشه و بسته به مقصد رسیده ولی ack اون گم میشه ینی بسته رسیده به مقصد ولی Ack اش نرسیده به فرستنده در اینصورت timeout اتفاق می افته و بسته دوباره ارسال میشه و بسته تکراری سمت گیرنده تشخیص میده ازروی چی ؟ از روی seq و بعد همون ack قبلی رو می فرسته باز و Ack اگه برسه بسته بعدی ارسال میشه

سناریو بعدی:
بسته های متوالی ارسال میشن و ackها به فرستنده داده میشه ولی تاخیر زیاد است و timeout زودتر اتفاق می افته و بسته مجدد ارسال میشه و باز هم اینجا بسته تکراری است و در سمت گیرنده مشکلی پیش نمیاد و گیرنده اخرین ack که فرستاده بود رو دوباره تکرار میکنه ینی مشخص میکنه بسته بعدی که باید ارسال بشه چی هست و اگر اینو سمت فرستنده دریافت کرد بسته بعدی رو ارسال میکنه

TCP: retransmission scenarios



سناریوی بعدی:
دو تا بسته متوالی ارسال شدن و اولی Ack اش گم شده ولی دومی ack اش گم نشده و به فرستنده برگردونده شده و timeout هم هنوز اتفاق نیوفته پس هیچ مشکلی پیش نمیاد ینی گم شدن ack قبلی هیچ نقشی اینجا نداره

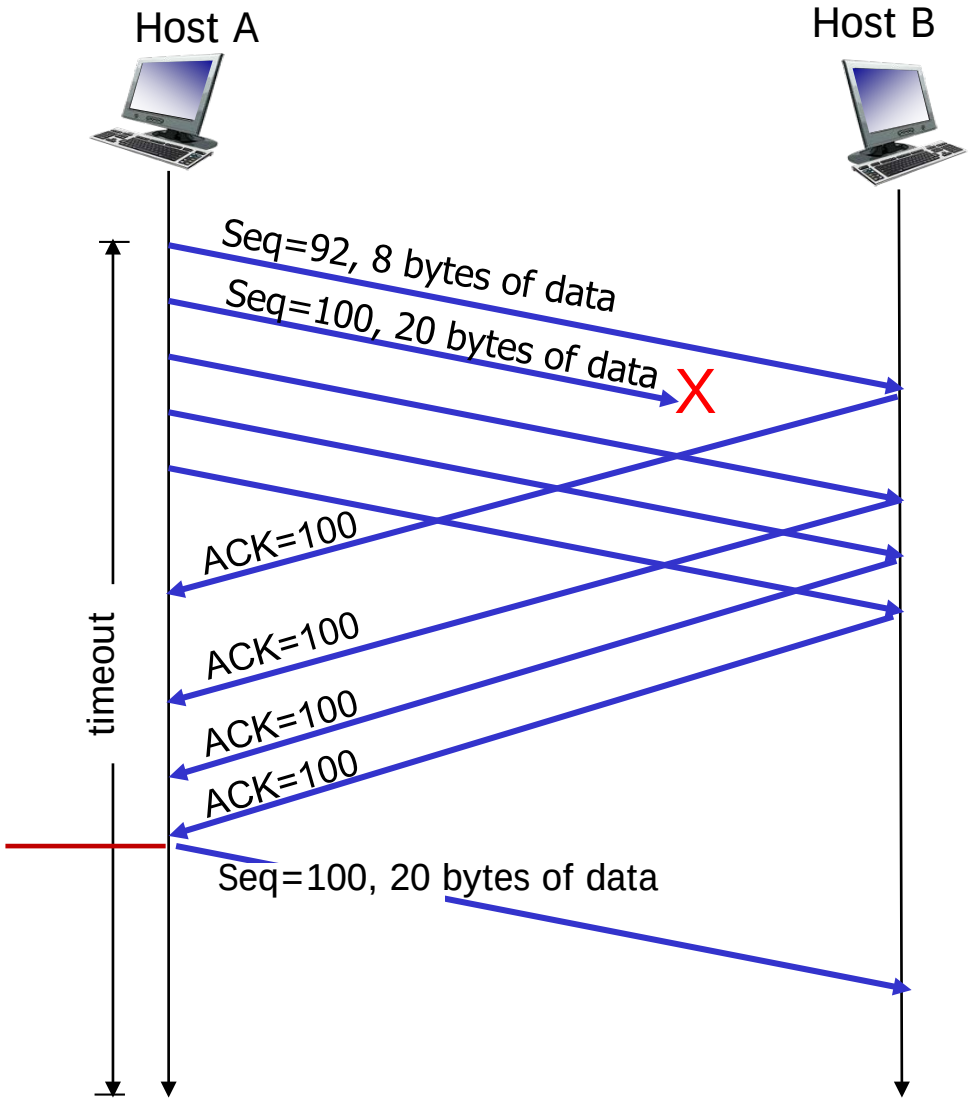
TCP fast retransmit

TCP fast retransmit

if sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #

- likely that unACKed segment lost, so don't wait for timeout

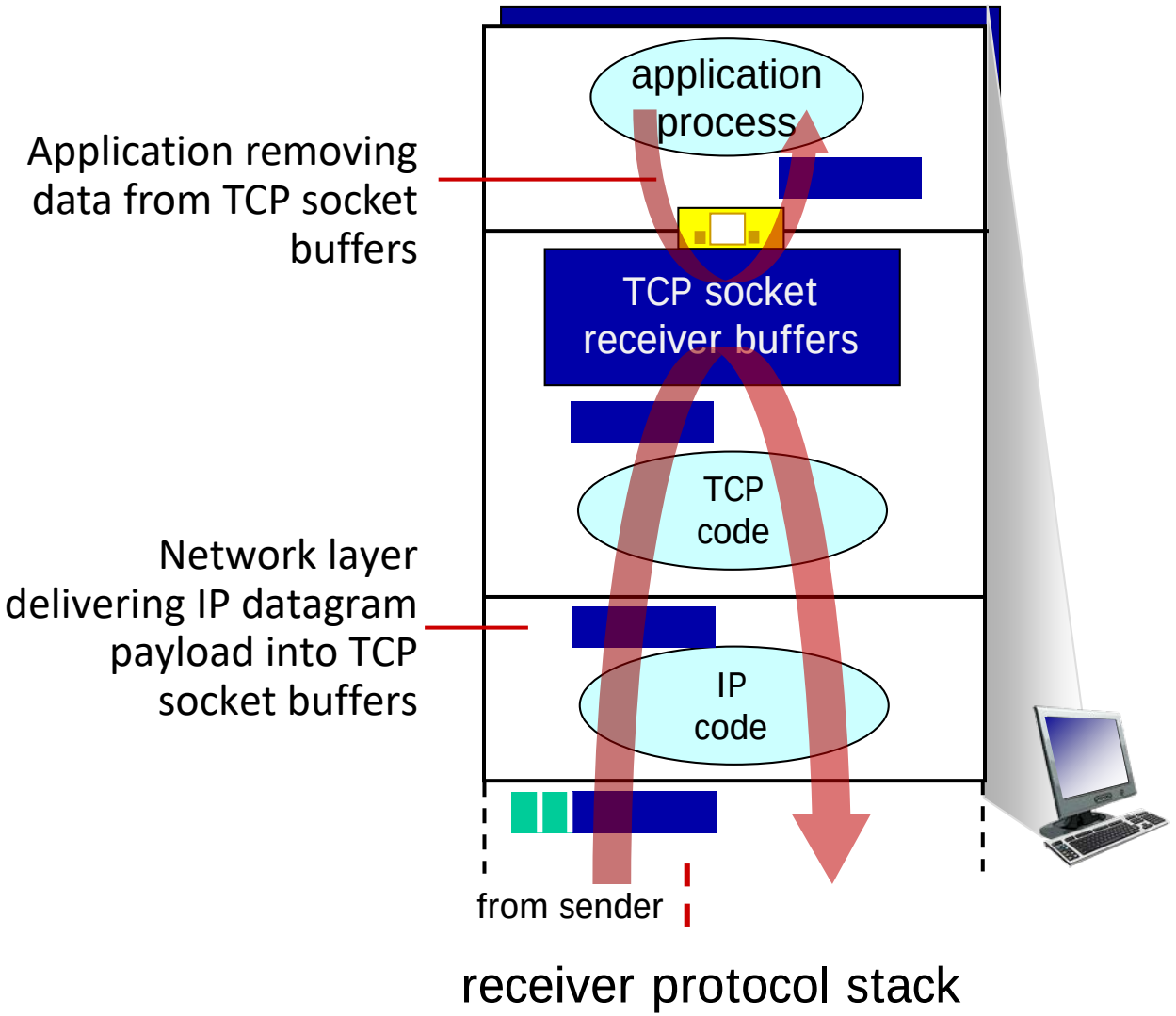
Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!



-
در پروتکل سمت گیرنده:
گفتیم اگر گیرنده یک بسته ای رو دریافت بکنه که Seq اون به ترتیب نیست متوجه میشه که یک بسته ای احتمالا گم شده و برای همین اخرین seq که دریافت کرده بود و Ack کرده بود رو ack اون رو تکرار میکنه بدین ترتیب سمت فرستنده یک Ack تکراری دریافت میشه و سمت فرستنده می تونه این برداشت رو داشته باشه که یکی از بسته هایی که فرستاده به مقصد نرسیده و صبر می کنه تا Ack های بعدی رو ببینه و در سمت گیرنده اگر بسته ها هم بدون ترتیب بیان ینی اون بسته ای که نرسیده هنوز نرسیده باشه اونا هم همون Ack رو تکرار میکنن بنابراین در سمت فرستنده تعداد ack تکراری دریافت میشه و حالا اینو می تونه تفسیر بکنه که یکی از بسته هاش نرسیده و اون بسته کدوم می تونه باشه؟ اولین بسته ای که Ack اش نیومده پس اون رو میتونه دوباره بفرسته بدون اینکه منتظر timeout بشه چون timeout رو گفتم معمولا زمانش رو خیلی زیاد ست میکنیم که مشکل ایجاد نکنه پس به این میگیم fast retransmit ینی بسته ای که باید retransmit بشه خیلی سریعتر اینو می فرسته
تعداد ack هایی برای duplicate ACK ها پیش بینی شده 3 تاست ینی اگر 3 تا ack تکراری دیده بشه به این معناست که بسته نرسیده و ان را دوباره ارسال بکن
مثال:
یک بسته ارسال شده و Ack اش برگشته و ack که اومده seq بسته بعدی که گیرنده انتظارش رو داره مشخصا ack میکنه
بسته بعدی ارسال شده ولی فرض میکنیم گم شده ولی بسته های بعدی تر دریافت شدن توسط گیرنده --> بسته بعدی که توسط گیرنده دریافت بشه گیرنده می بینه که ترتیبش درست نیست و یک بسته این توسط دریافت نشده برای همین اخرین ack رو تکرار میکنه و بسته بعدی و بعدی تر هم به همین شکل و باز این اخرین Ack رو تکرار میکنه و به این ترتیب 3 تا ack تکراری و سمت فرستنده مطمئن میشه که اون بسته گم شده و اون بسته ای که ack داره براش میاد رو دوباره می فرسته

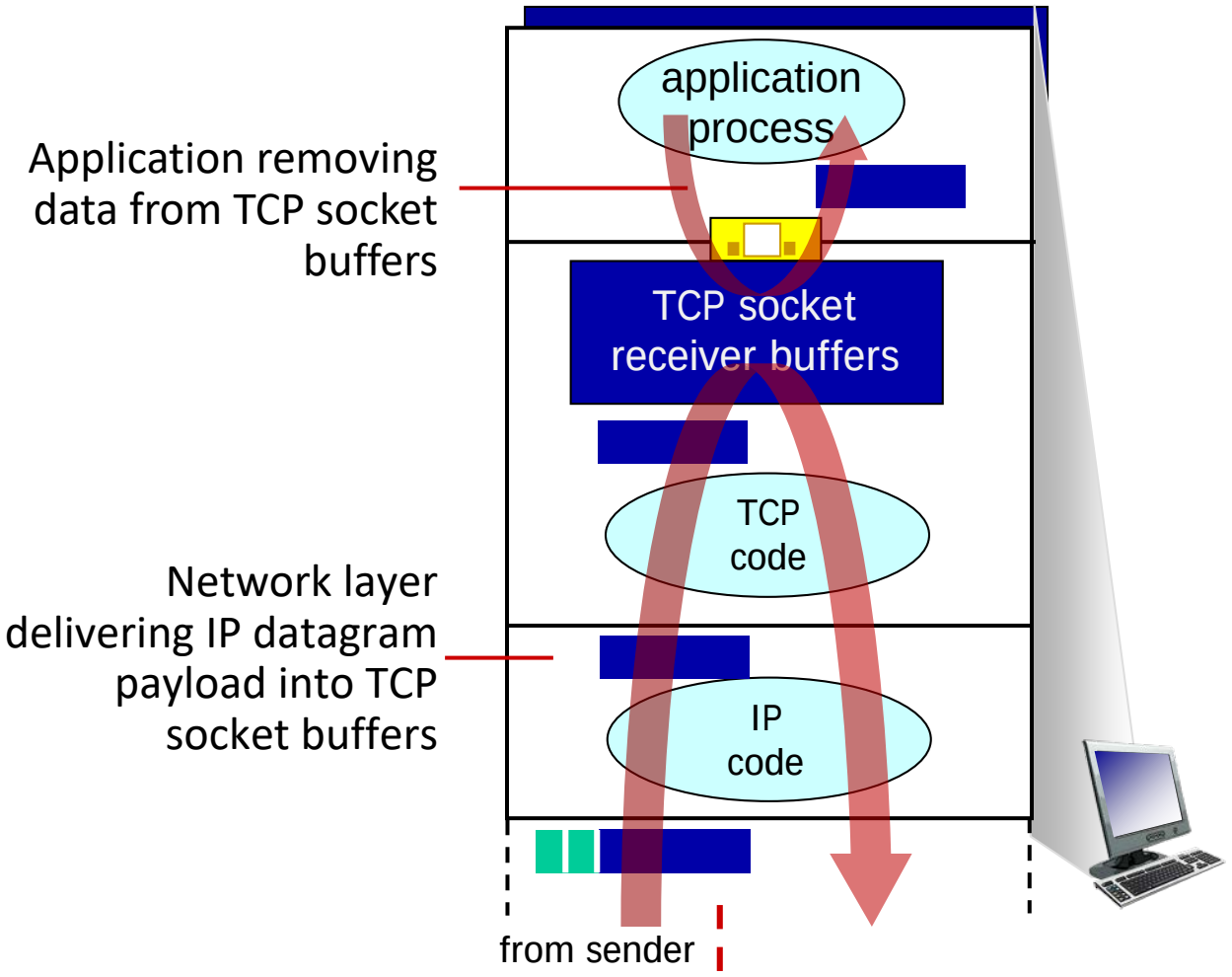
TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



TCP flow control

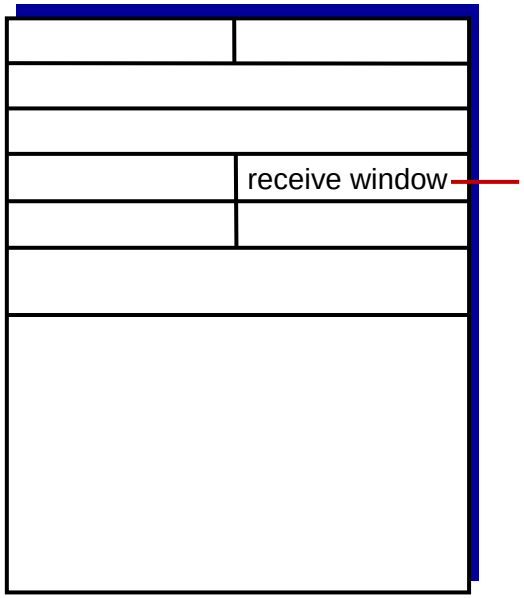
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



--

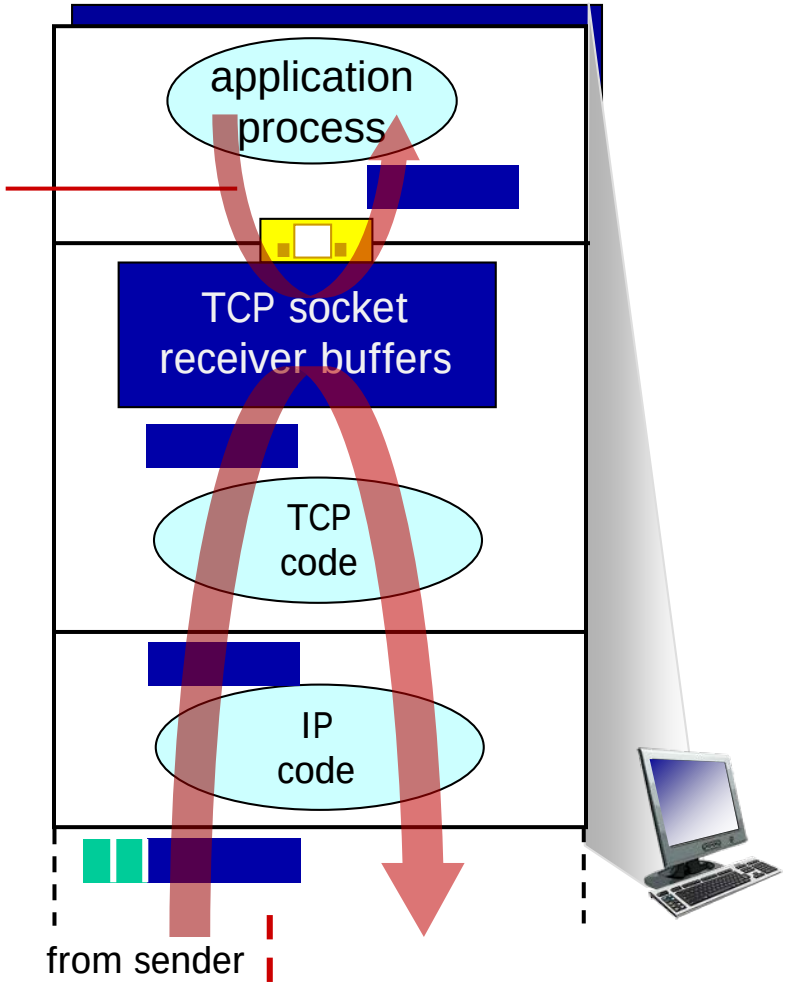
TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



flow control: # bytes receiver willing to accept

Application removing data from TCP socket buffers



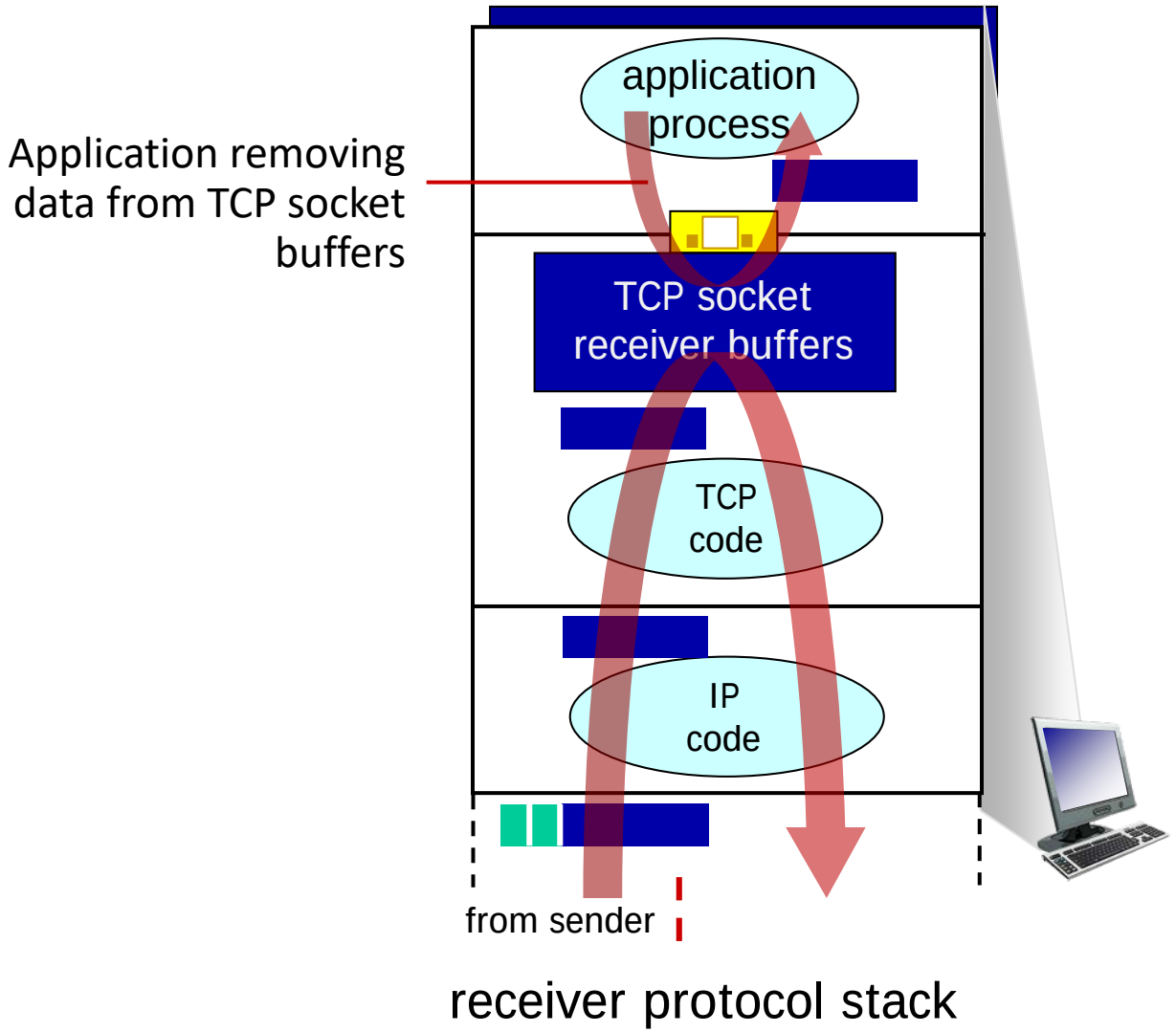
receiver protocol stack

TCP flow control

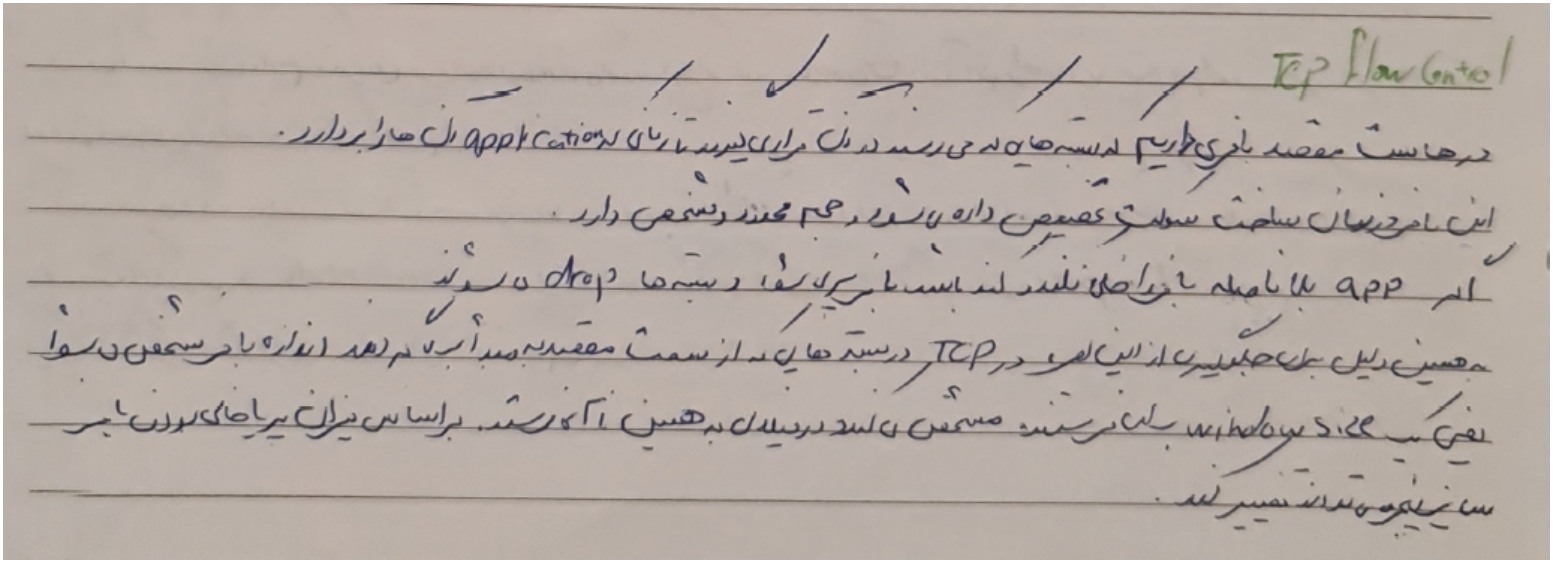
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

flow control

receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

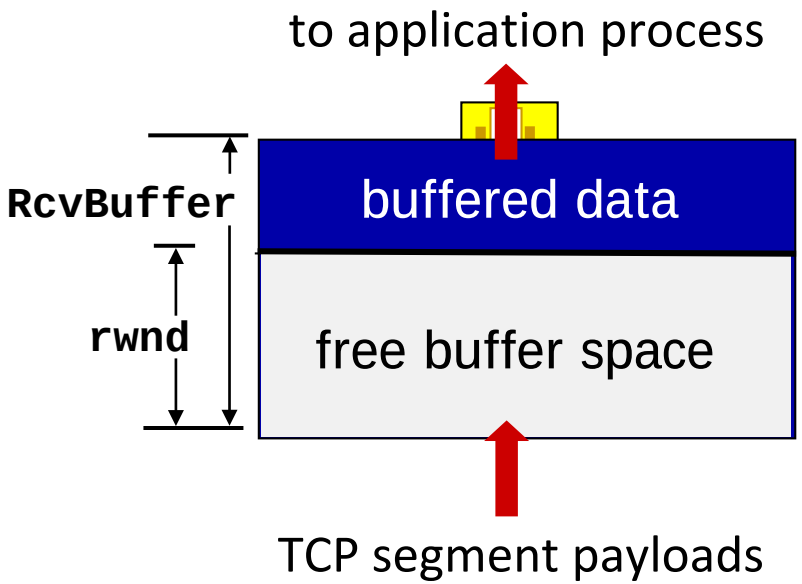


....
گیرنده فرستنده را کنترل می کند، بنابراین فرستنده با ارسال بیش از حد و خیلی سریع، بافر گیرنده را سرریز نمی کند.



TCP flow control

- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow

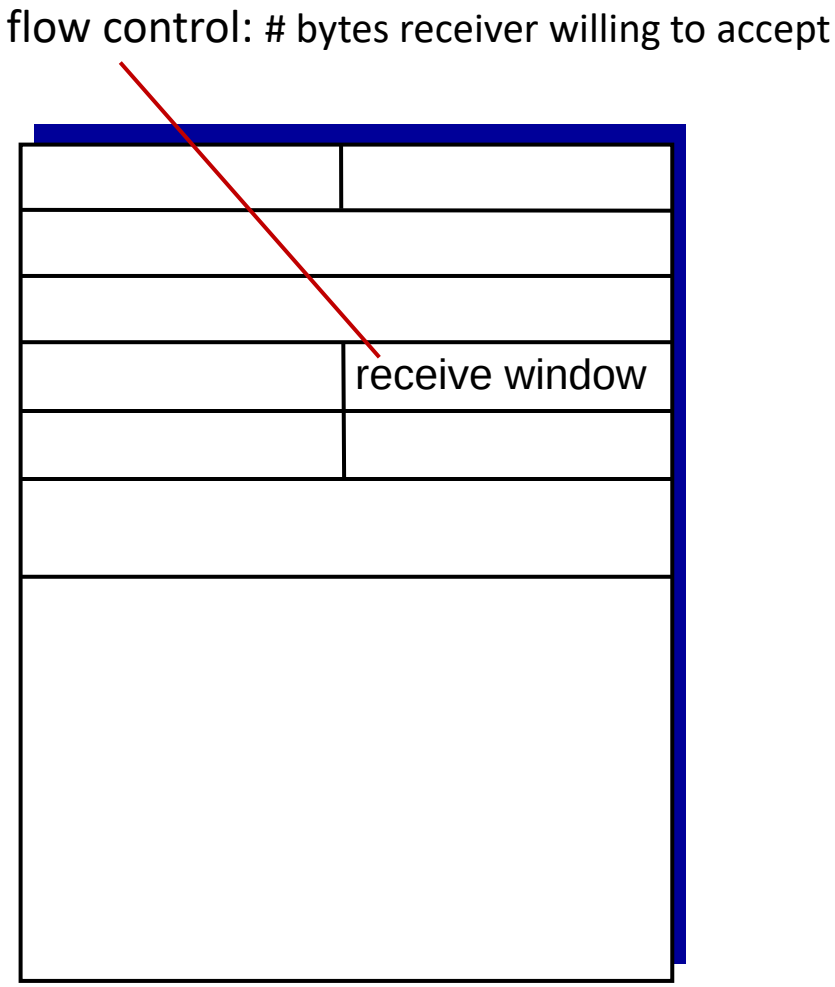


TCP receiver-side buffering

-
مقصد rwnd رو در window size در هدر بسته های tcp به مبدا اعلام می کنه

TCP flow control

- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow



TCP segment format
