Network Transport Layer: Overview; UDP; Stop-and-Wait ARQ

Qiao Xiang, Congming Gao

https://sngroup.org.cn/courses/cnns-xmuf24/index.shtml

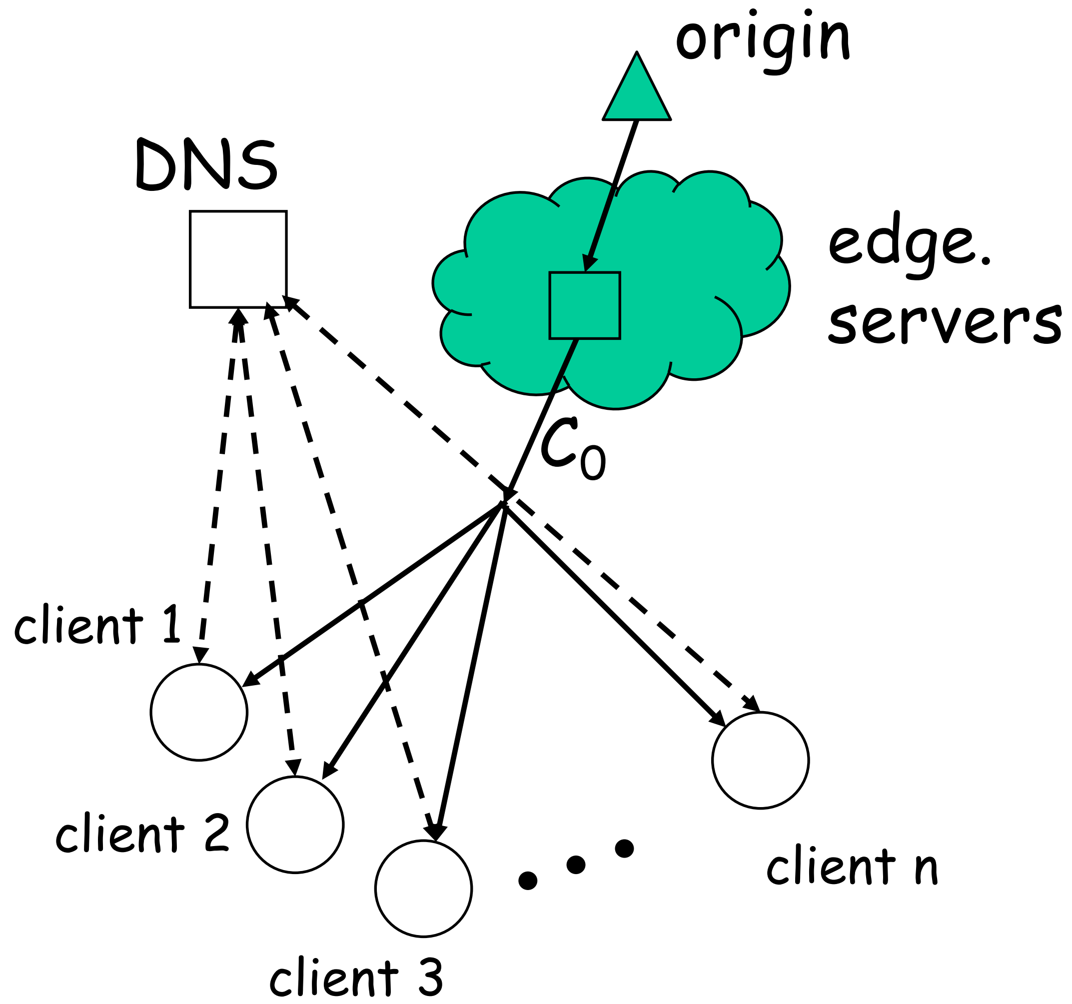
10/31/2024

This deck of slides are heavily based on CPSC 433/533 at Yale University, by courtesy of Dr. Y. Richard Yang.

Outline

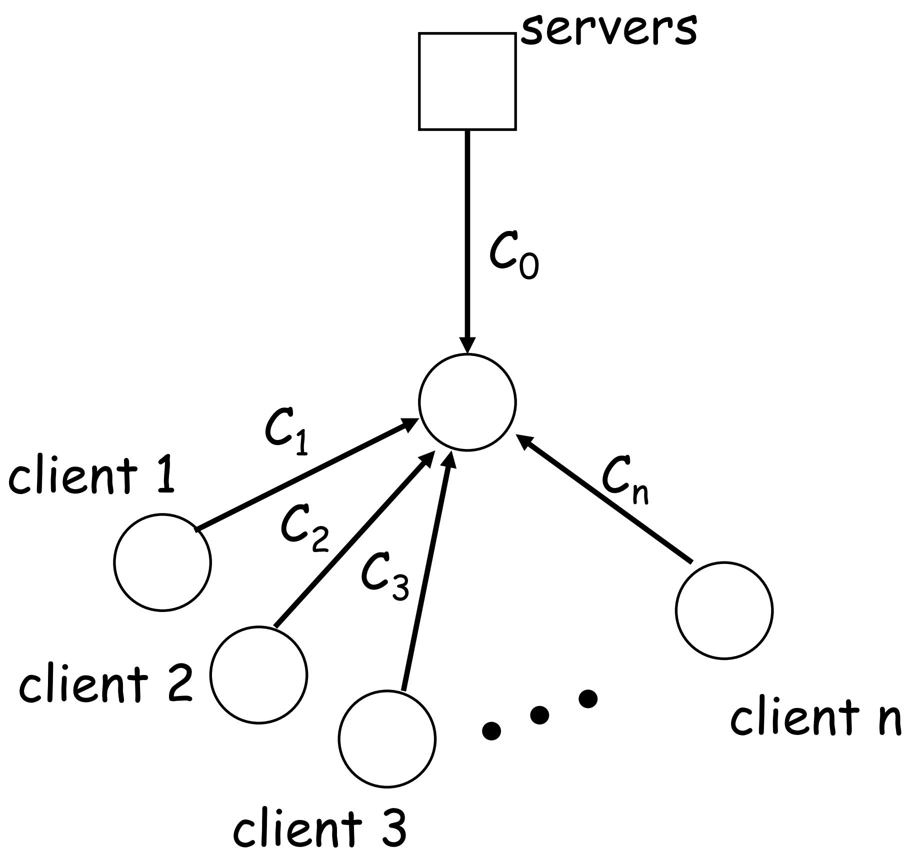
- Admin and recap
- Overview of transport layer
- UDP
- Reliable data transfer, the stop-and-go protocols

Recap: Scalability of Server-Only Approaches



Server+Host (P2P) Content Distribution: Key Design Issues

- Robustness
  - Resistant to churns and failures
- Efficiency
  - A client has content that others need; otherwise, its upload capacity may not be utilized
- Incentive: clients are willing to upload
  - Some real systems nearly 50% of all responses are returned by the top 1% of sharing hosts

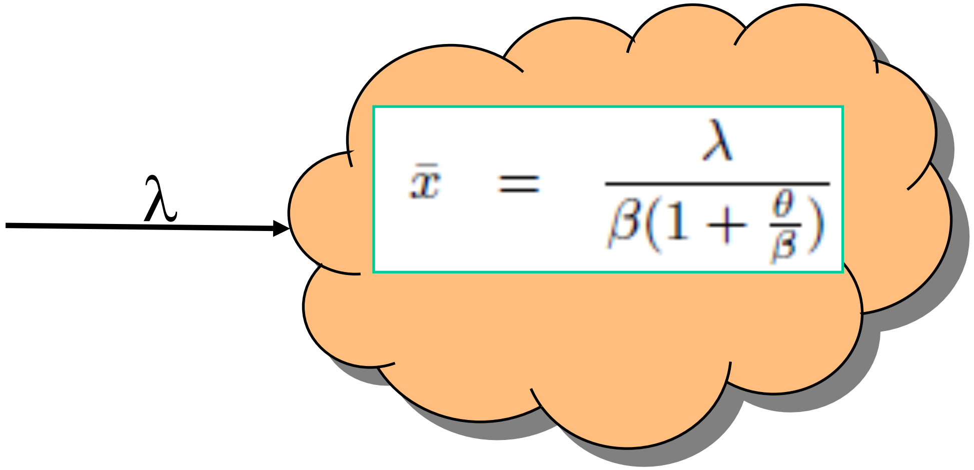


System State

$$\bar{x} = \frac{\lambda}{\beta(1+\frac{\theta}{\beta})}$$

$$\bar{y} = \frac{\lambda}{\gamma(1+\frac{\theta}{\beta})}.$$

Q: How long does each downloader stay as a downloader?



$$T = \frac{1}{\theta + \beta}$$

$$\frac{1}{\beta} = \max\{\frac{1}{c}, \frac{1}{\eta}(\frac{1}{\mu} - \frac{1}{\gamma})\}$$

Key take-away: not scaling inverse with system size (x)

- New requests comes, new bandwidth also comes

Recap

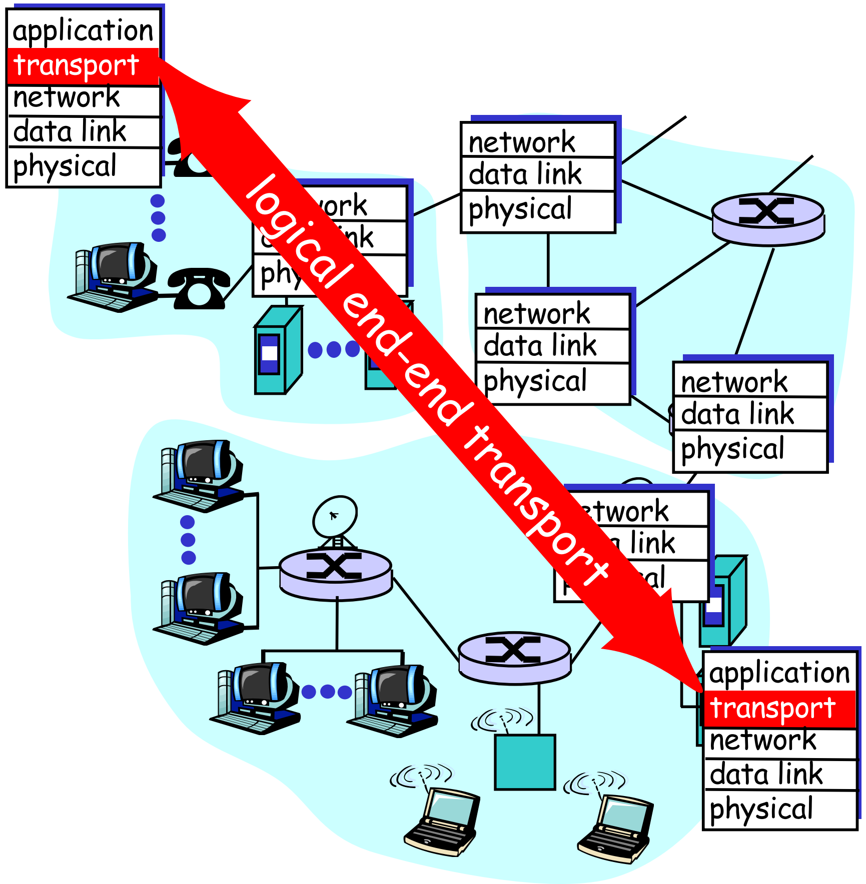
- Applications
  - Client-server applications
    - Single server
    - Multiple servers load balancing
  - Application overlays (distributed network applications) to
    - scale bandwidth/resource (BitTorrent)
    - distribute content lookup (Freenet, DHT, Chord) [optional]
    - distribute content verification (Block chain) [optional]
    - achieve anonymity (Tor) [optional]

Outline

- Admin and recap
- *Overview of transport layer*
- UDP
- Reliable data transfer, the stop-and-go protocols

Overview

- Provide *logical communication* between app' processes
- Transport protocols run in end systems
  - send side: breaks app messages into segments, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- Transport vs. network layer services:
  - *Network layer:* data transfer between end systems
  - *Transport layer:* data transfer between processes
    - relies on, enhances network layer services



Transport Layer Services and Protocols

- Reliable, in-order delivery (TCP)
  - multiplexing
  - reliability and connection setup
  - congestion control
  - flow control

- Unreliable, unordered delivery: UDP
  - multiplexing

- Services not available:
  - delay guarantees
  - bandwidth guarantees

Transport Layer: Road Ahead

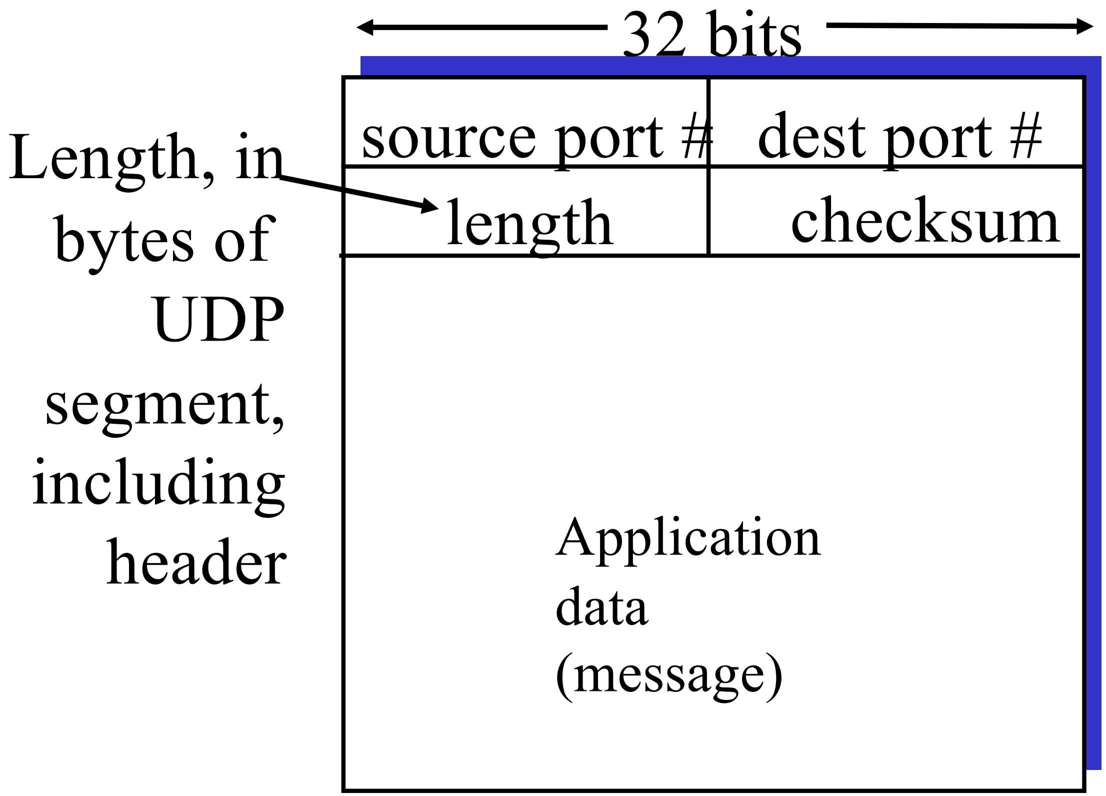
- Class 1 (today):
  - transport layer services
  - connectionless transport: UDP
  - reliable data transfer using stop-and-wait/alternating-bit protocol
- Class 2 (ready for lab assignment 4/part 1):
  - sliding window reliability
  - TCP reliability
    - overview of TCP
    - TCP RTT measurement
    - TCP connection management
- Class 3 (ready for lab assignment 4/part 2 [optional]):
  - principles of congestion control
  - TCP congestion control; AIMD; TCP Reno
- Class 4:
  - TCP Vegas, performance modeling; Nash Bargaining solution
- Class 5:
  - primal-dual as a resource allocation and analysis framework
- ...

Outline

- Admin and recap
- Overview of transport layer
- *UDP and error checking*
- Reliable data transfer, the stop-and-go protocols

UDP: User Datagram Protocol [RFC 768]

- Often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- Other UDP uses
  - DNS
  - SNMP



UDP segment format

UDP Checksum

Goal: end-to-end detection of "errors" (e.g., flipped bits) in transmitted segment

Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition of segment contents to be zero
- sender puts checksum value into UDP checksum field

Receiver:

- compute sum of segment and checksum; check if sum zero
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?*

One's Complement Arithmetic

- UDP checksum is based on one's complement arithmetic
  - one's complement was a common representation of signed numbers in early computers
- One's complement representation
  - bit-wise NOT for negative numbers
  - example: assume 8 bits
    - `00000000: 0`
    - `00000001: 1`
    - `01111111: 127`
    - `10000000: ?`
    - `11111111: ?`
  - addition: conventional binary addition except adding any resulting carry back into the resulting sum
    - Example: -1 + 2

UDP Checksum: Algorithm

- Example checksum:

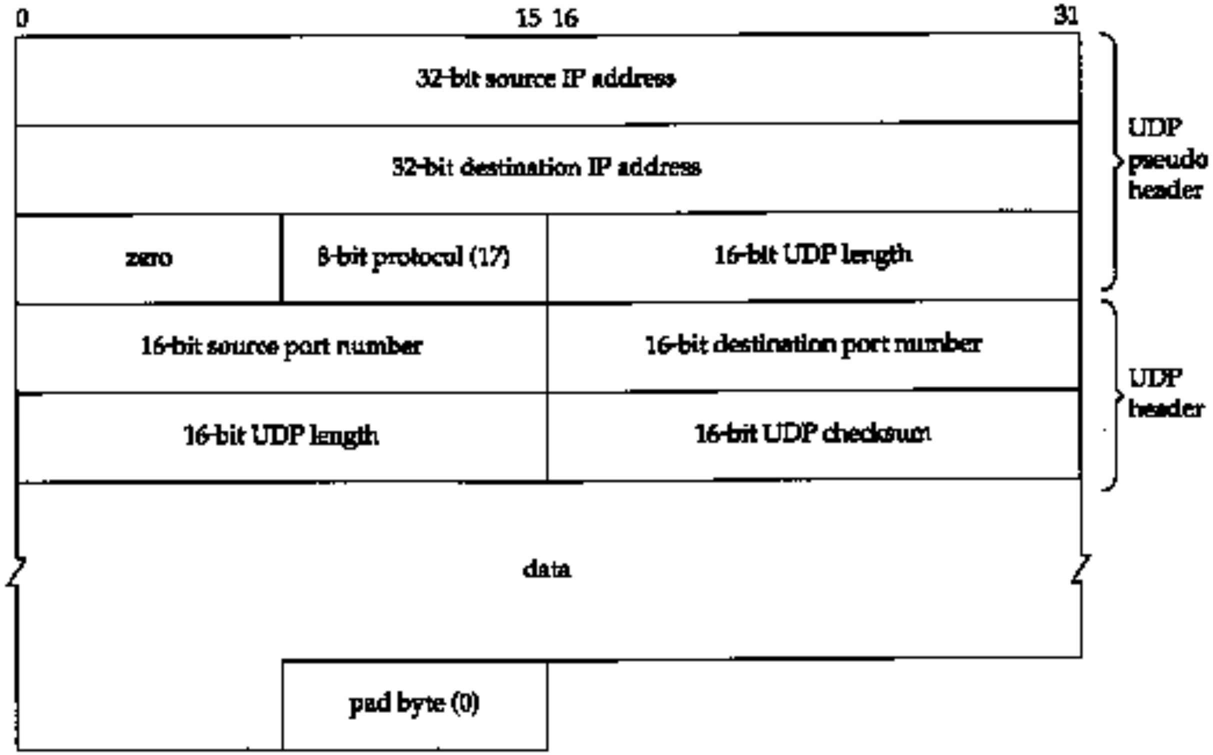
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

- For fast implementation of computing UDP checksum, see http://www.faqs.org/rfcs/rfc1071.html

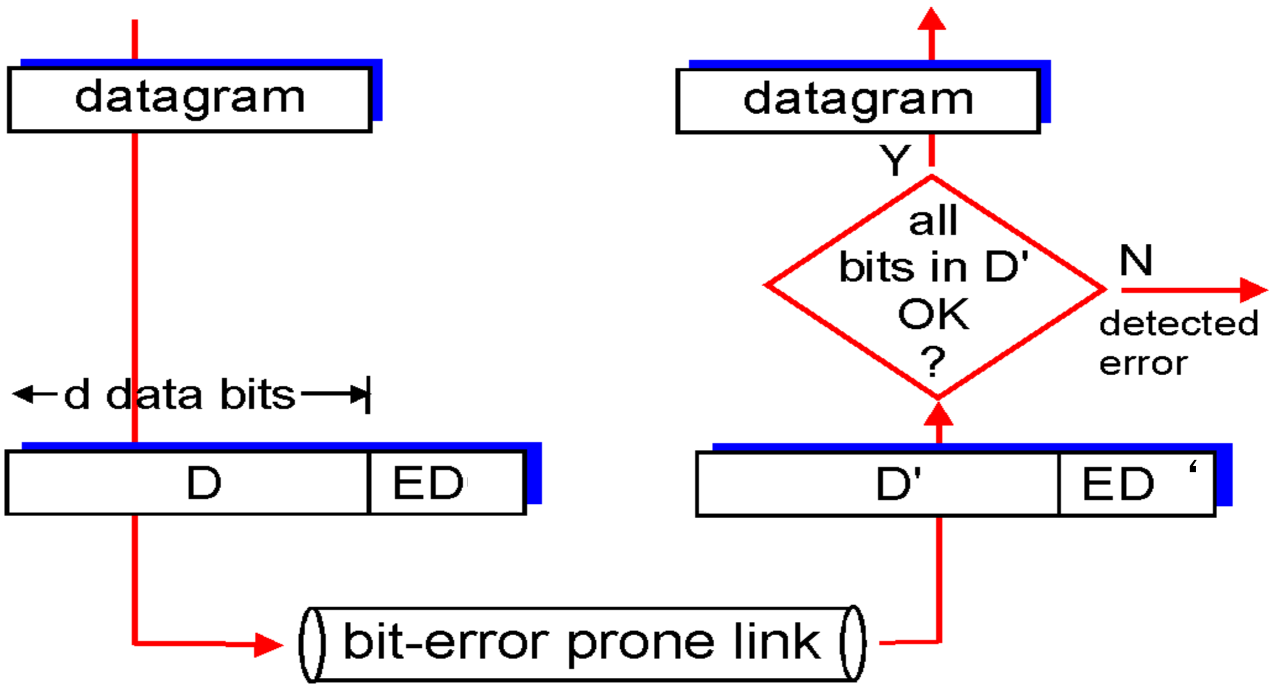
UDP Checksum: Coverage

Calculated over:

- A pseudo-header
  - IP Source Address (4 bytes)
  - IP Destination Address (4 bytes)
  - Protocol (2 bytes)
  - UDP Length (2 bytes)
- UDP header
- UDP data



General Error Detection (Checksum)



D   = Data protected by error checking, may include header fields
ED = Error Detection bits (redundancy)

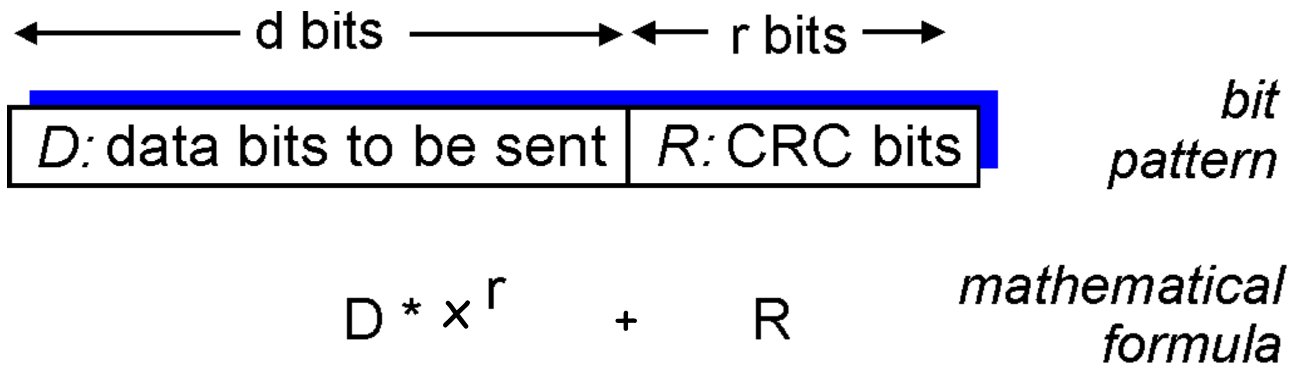
- Error detection not 100% reliable!
  - a good error detector may miss some errors, but rarely
  - larger ED field generally yields better detection

Cyclic Redundancy Check: Background

- Widely used in practice, e.g.,
  - Ethernet, DOCSIS (Cable Modem), FDDI, PKZIP, WinZip, PNG
- For a given data D, consider it as a polynomial D(x)
  - consider the string of 0 and 1 as the coefficients of a polynomial
    - e.g. consider string 10011 as x^4+x+1
  - addition and subtraction are modular 2, thus the same as xor
- Choose generator polynomial G(x) with r+1 bits, where r is called the degree of G(x)

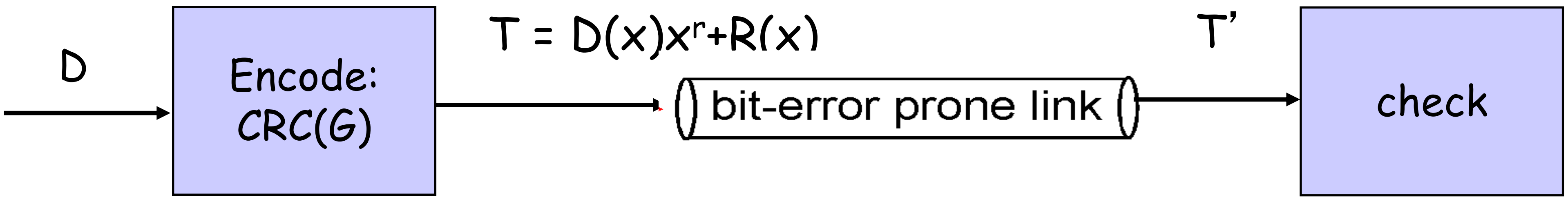
Cyclic Redundancy Check: Encode

- Given data G(x) and D(x), choose R(x) with r bits, such that
  - $D(x)x^r+R(x)$ is exactly divisible by G(x)



- The bits correspond to $D(x)x^r+R(x)$ are sent to the receiver

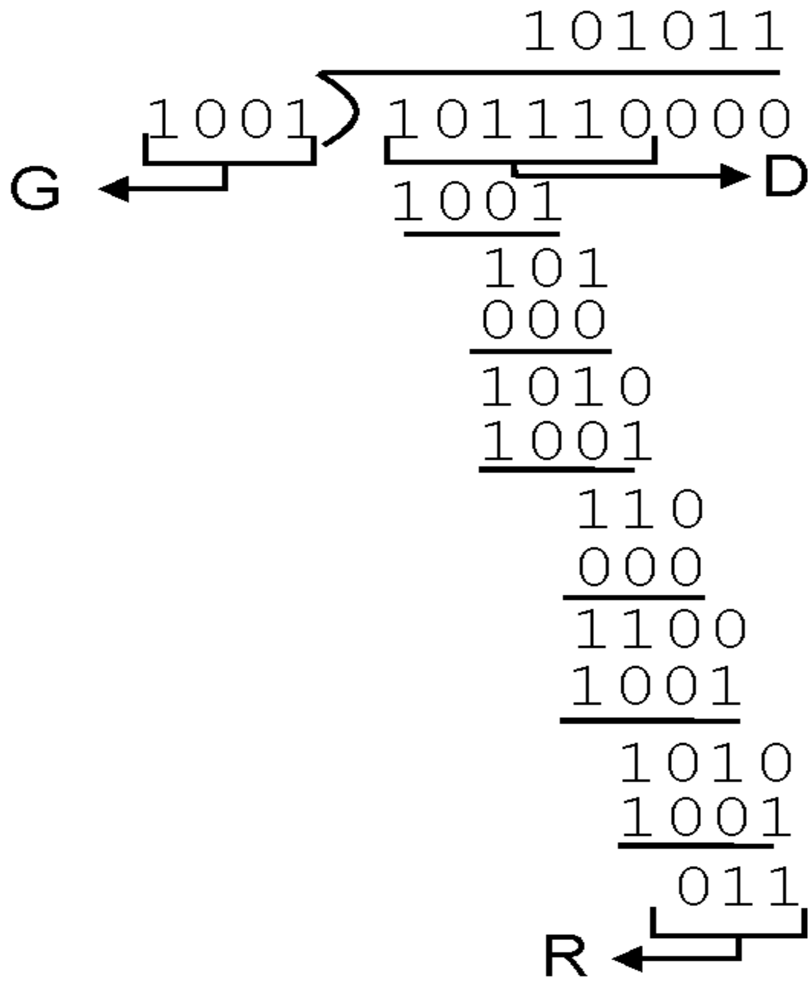
Cyclic Redundancy Check: Decode



- Since G(x) is global, when the receiver receives the transmission T' (x), it divides T' (x) by G(x)
  - if non-zero remainder: error detected!
  - if zero remainder, assumes no error

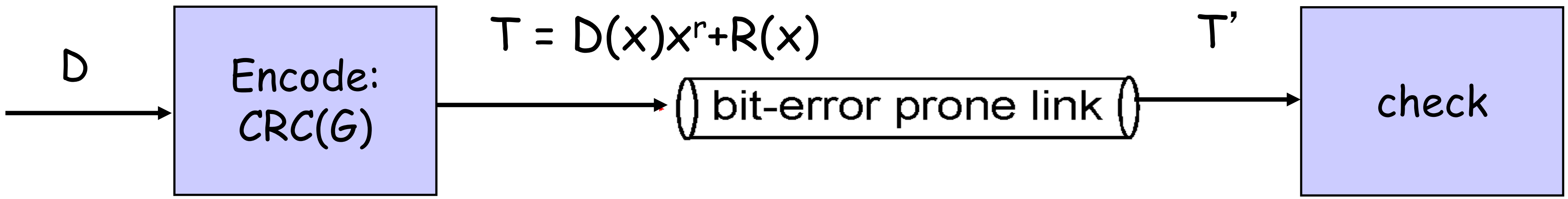
CRC: Steps and an Example

- Suppose the degree of G(x) is r
- Append r zero to D(x), i.e. consider $D(x)x^r$
- Divide $D(x)x^r$ by G(x). Let R(x) denote the reminder
- Send <D, R> to the receiver



The Power of CRC

- Let T(x) denote $D(x)x^r+R(x)$, and E(x) the polynomial of the error bits
  - the received signal is T' (x) = T(x)+E(x)



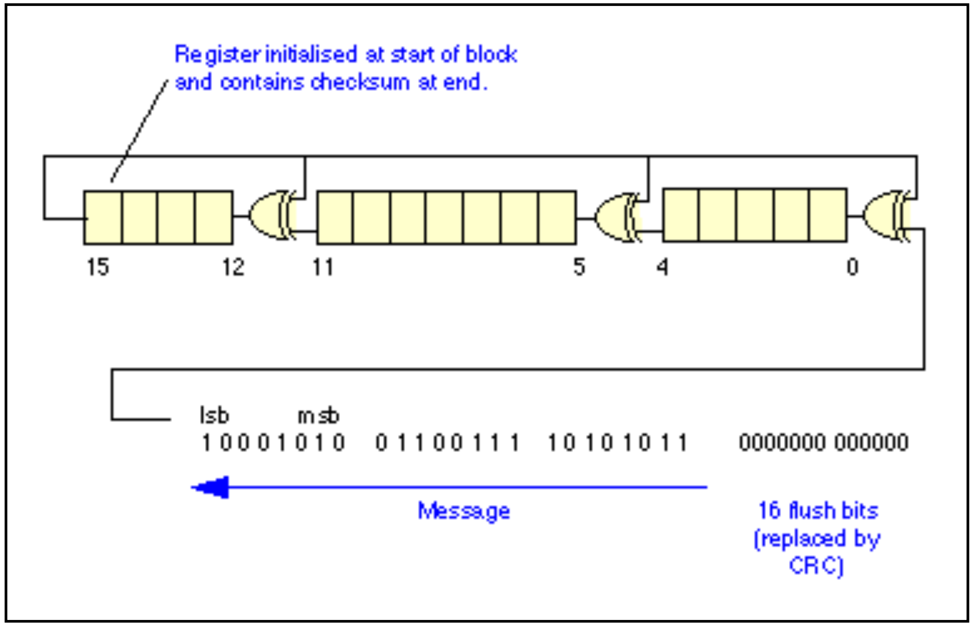
- Since T(x) is divisible by G(x), we only need to consider if E(x) is divisible by G(x)

The Power of CRC

- Detect a single-bit error: $E(x) = x^i$
  - if G(x) contains two or more terms, E(x) is not divisible by G(x)
- Detect an odd number of errors: E(x) has an odd number of terms:
  - lemma: if E(x) has an odd number of terms, E(x) cannot be divisible by (x+1)
    - suppose E(x) = (x+1)F(x), let x=1, the left hand will be 1, while the right hand will be 0
  - thus if G(x) contains x+1 as a factor, E(x) will not be divided by G(x)
- Many more errors can be detected by designing the right G(x)

Example G(x)

- 16 bits CRC:
  - CRC-16: $x^{16}+x^{15}+x^{2}+1$,
    CRC-CCITT: $x^{16}+x^{12}+x^{5}+1$
  - both can catch
    - all single or double bit errors
    - all odd number of bit errors
    - all burst errors of length 16 or less
    - >99.99% of the 17 or 18 bits burst errors



CRC-16 hardware implementation
Using shift and XOR registers

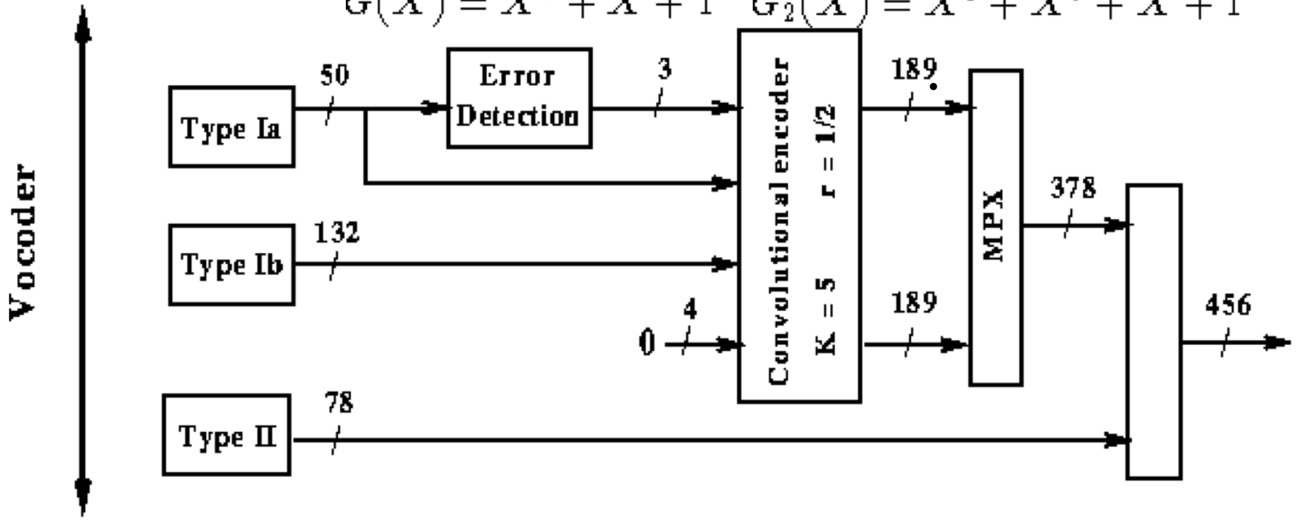
http://en.wikipedia.org/wiki/CRC-32#Implementation

Example G(x)

- 32 bits CRC:
  - *CRC32:* $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$
  - used by Ethernet, FDDI, PKZIP, WinZip, and PNG
- GSM phones

$G_1(X) = X^4 + X^3 + 1$

$G(X) = X^3 + X + 1 \quad G_2(X) = X^4 + X^3 + X + 1$



- For more details see the link below and further links it contains:
  - http://en.wikipedia.org/wiki/Cyclic_redundancy_check

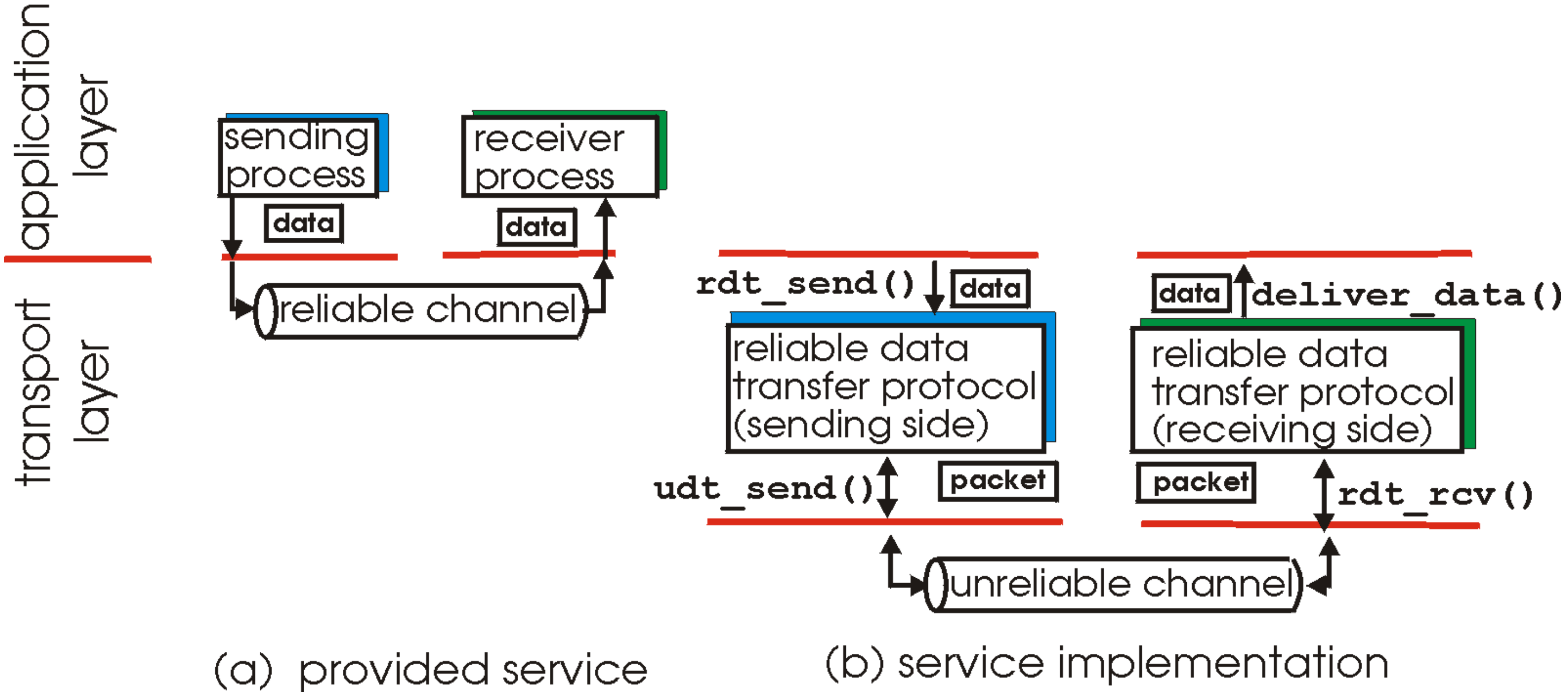
Outline

- Admin and recap
- Transport overview
- UDP
- *Reliable data transfer*

Principles of Reliable Data Transfer (RDT)

- Important in app., transport, link layers
- Foundation to other protocols
- We use the development of RDT to also better appreciate understanding distributed protocols

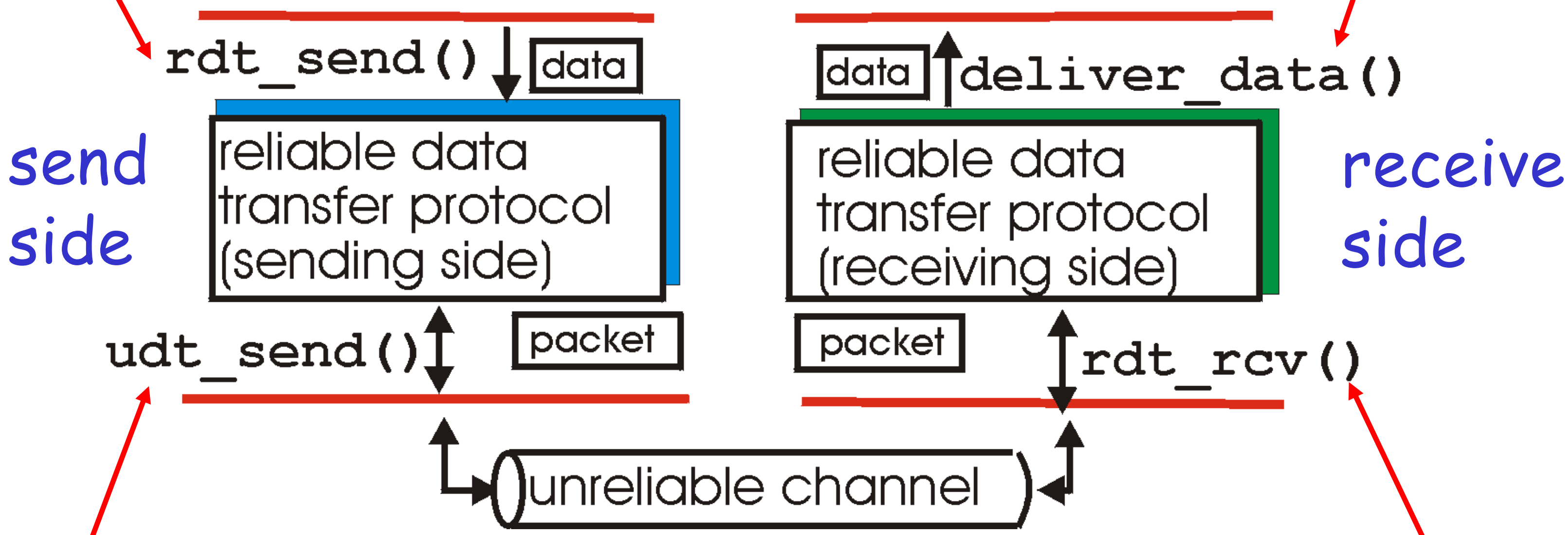
Reliable Data Transfer



Reliable Data Transfer: Getting Started

rdt_send(): called from above, (e.g., by app.)

deliver_data(): called by **rdt** to deliver data to upper



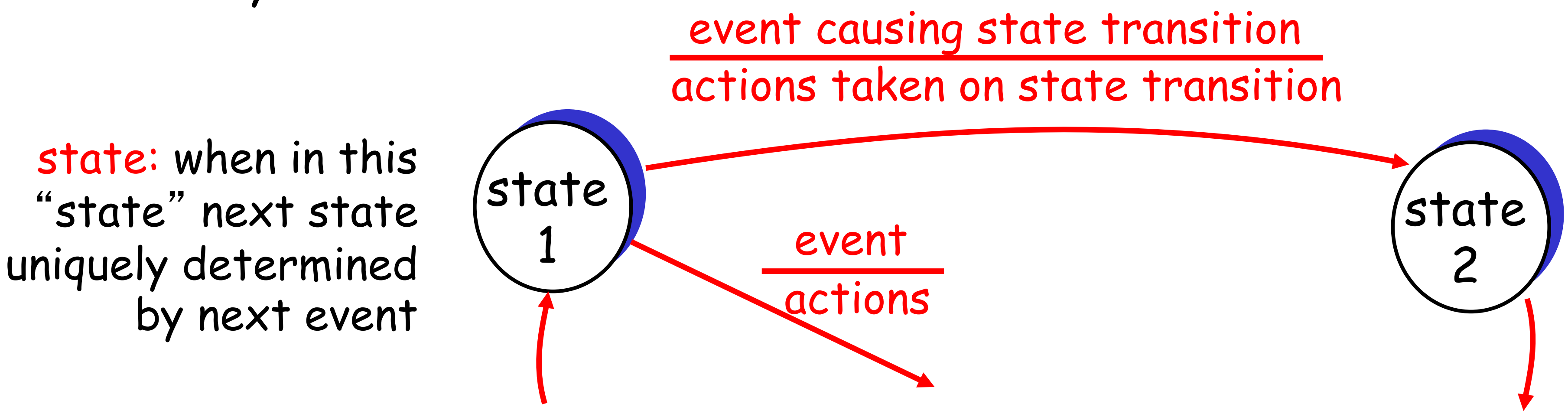
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

rdt_rcv(): called from below; when packet arrives on rcv-side of channel

Reliable Data Transfer: Getting Started

We'll:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions !
- use finite state machines (FSM) to specify sender, receiver

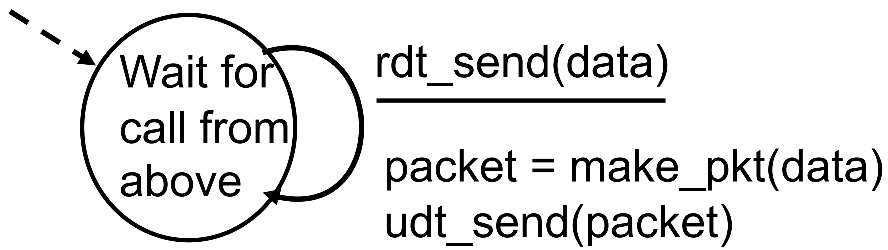


Outline

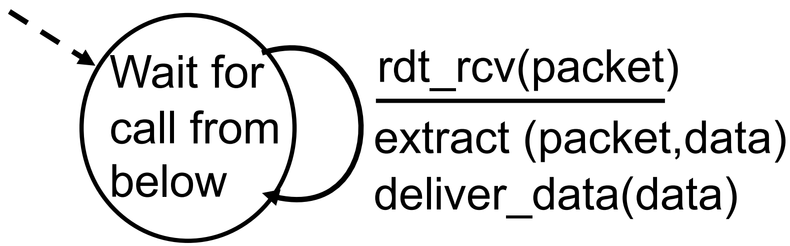
- Admin and review
- Overview of transport layer
- UDP and error checking
- Reliable data transfer
  - *perfect channel*

Rdt1.0: reliable transfer over a reliable channel

- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel



sender



receiver

Exercise: Prove correctness of Rdt1.0.

Correctness: for every single packet, one and only one copy is received by receiver correctly (no error) and in-order

Potential Channel Errors

- bit errors
- loss (drop) of packets
- reordering or duplication

Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt).