Application Overlays (P2P); Network Transport Layer: Overview; UDP; Stop-and-Wait ARQ

Qiao Xiang

https://qiaoxiang.me/courses/cnns-xmuf22/index.shtml

10/18/2022

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

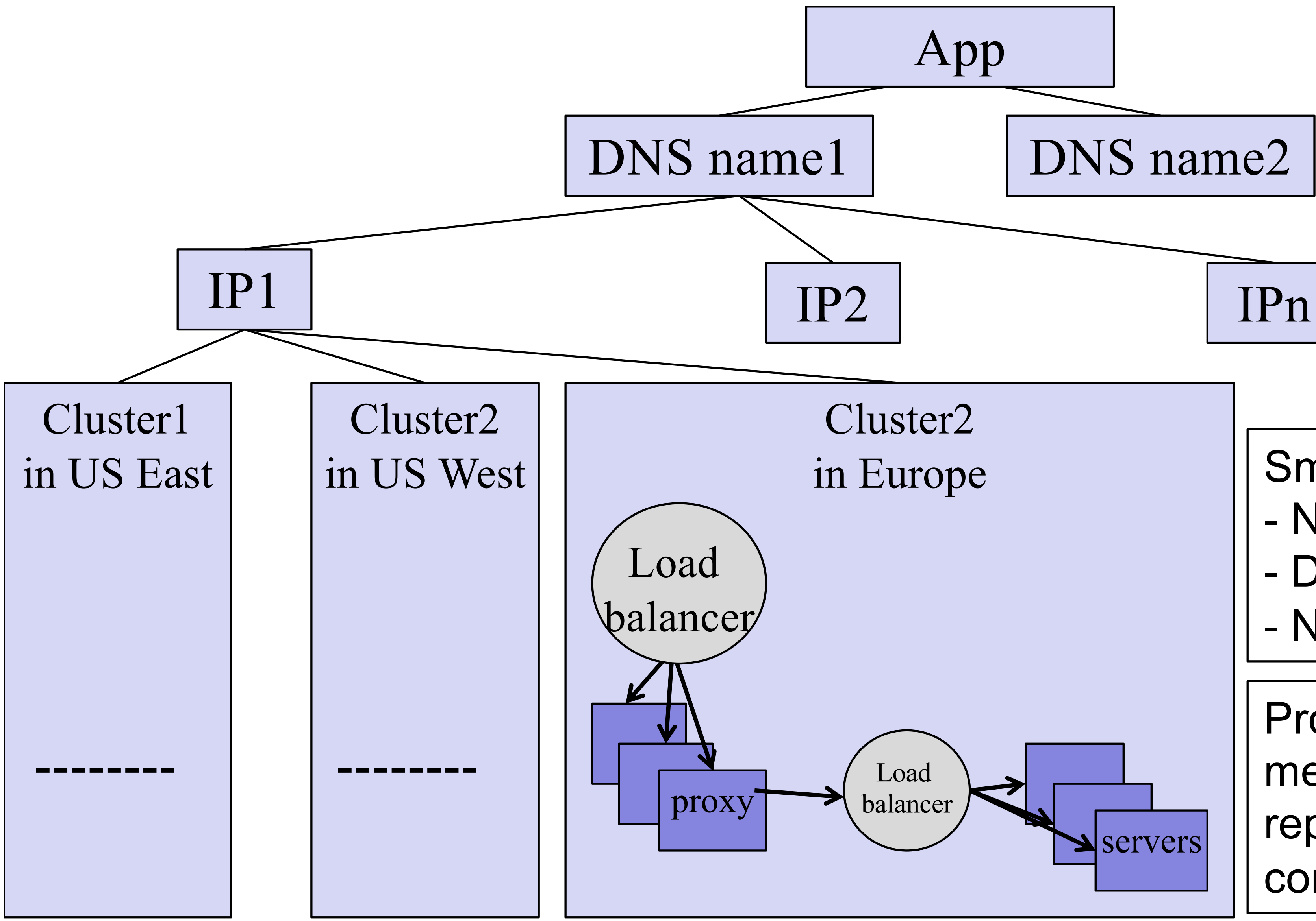
Outline

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

Admin

- Lab assignment 3 due on Nov. 8
- Midterm exam on Nov. 10 (during lab class)
    - cover from introduction to application layer
    - 15-16 subjective questions over 100 minutes
    - 1-page cheat sheet allowed
- Class project: please check the project topic list and talk to mentors to decide which project you want to work on.
    - https://qiaoxiang.me/courses/cnns-xmuf22/files/projects/index.html

Recap: Direction Mechanisms



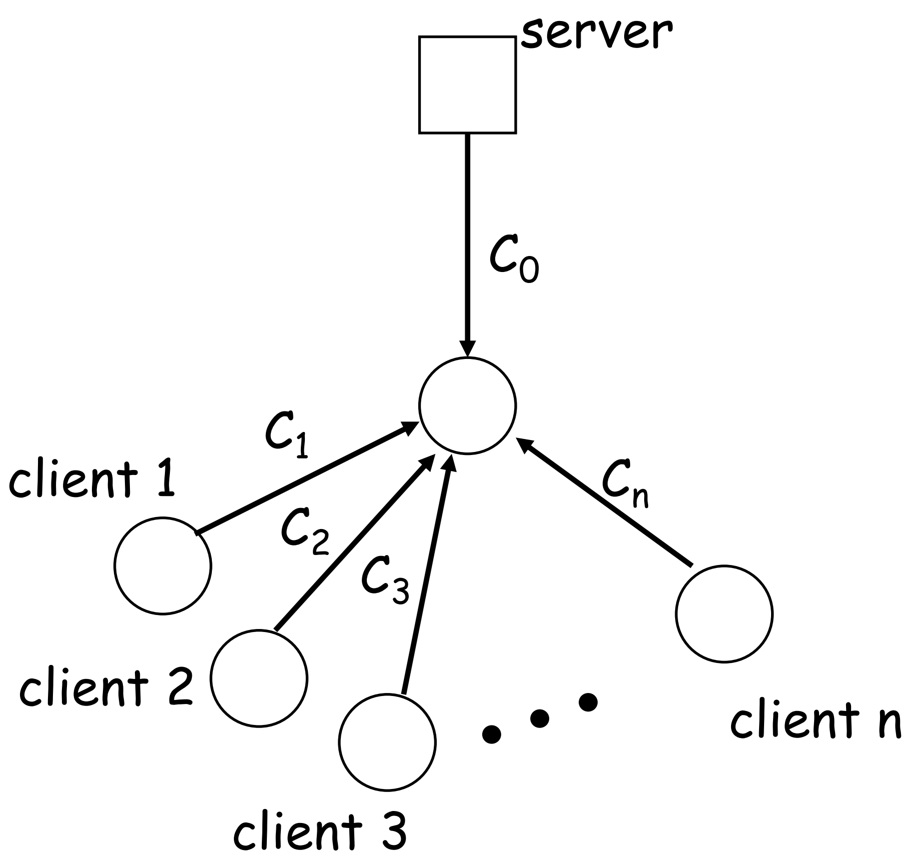
- cname
- hierarchy

Smart switch/LB
- NAT rewrite
- Direct reply
- NLB

Proxy as a mechanism for replica content consistency

An Upper Bound on Scalability

- Idea: use resources from both clients and the server
- Assume
  - need to achieve same rate to all clients
  - only uplinks can be bottlenecks
- What is an upper bound on scalability?

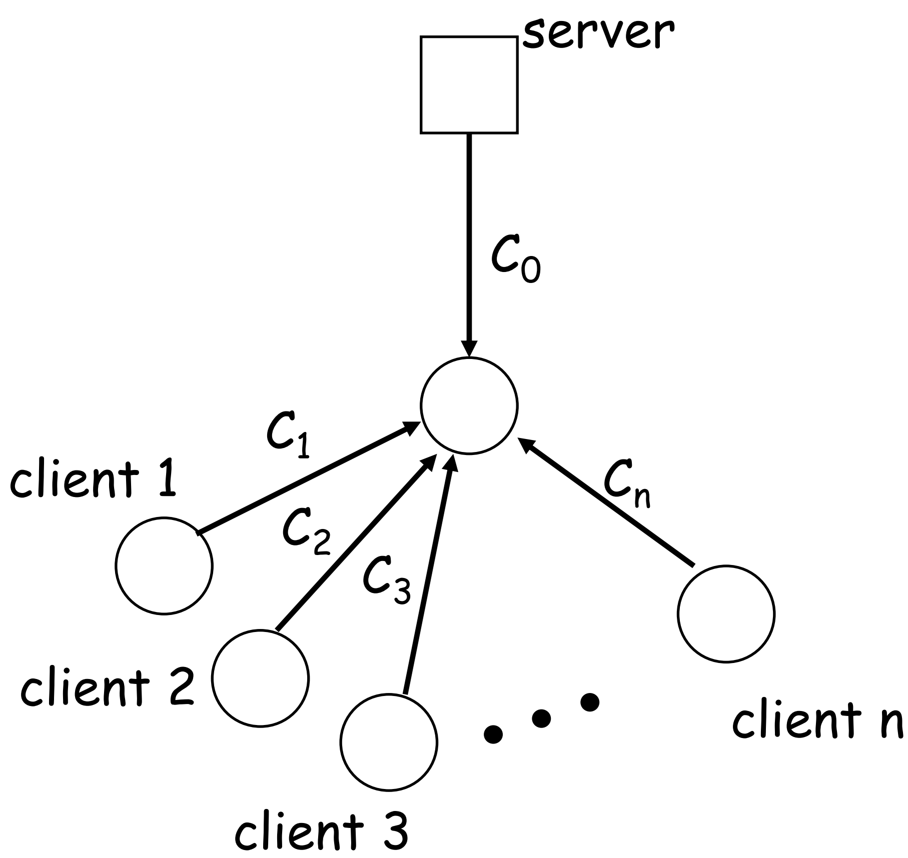


The Scalability Problem

- Maximum throughput

  $R = \min\{C_0, (C_0+\Sigma C_i)/n\}$

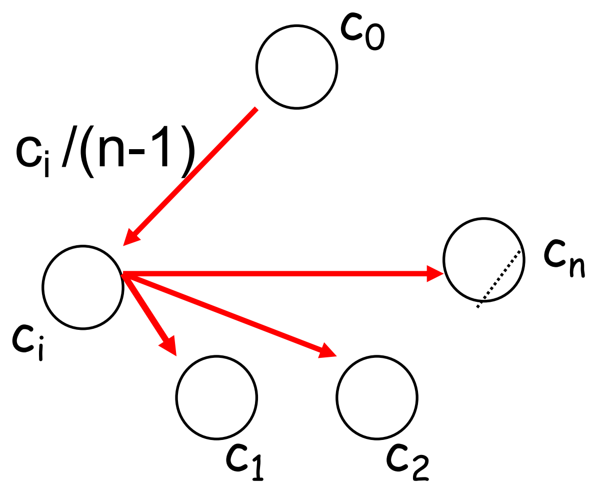
- The bound is theoretically approachable



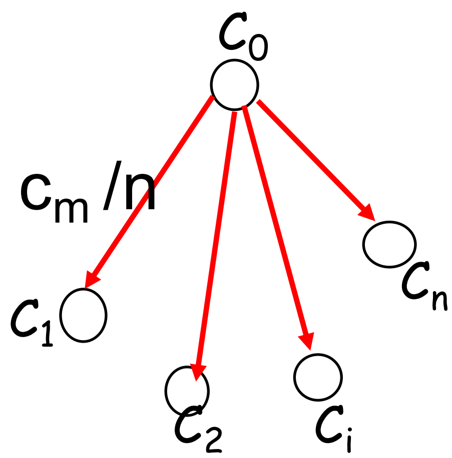
Theoretical Capacity: upload is bottleneck

$R = \min\{C_0, (C_0+\Sigma C_i)/n\}$

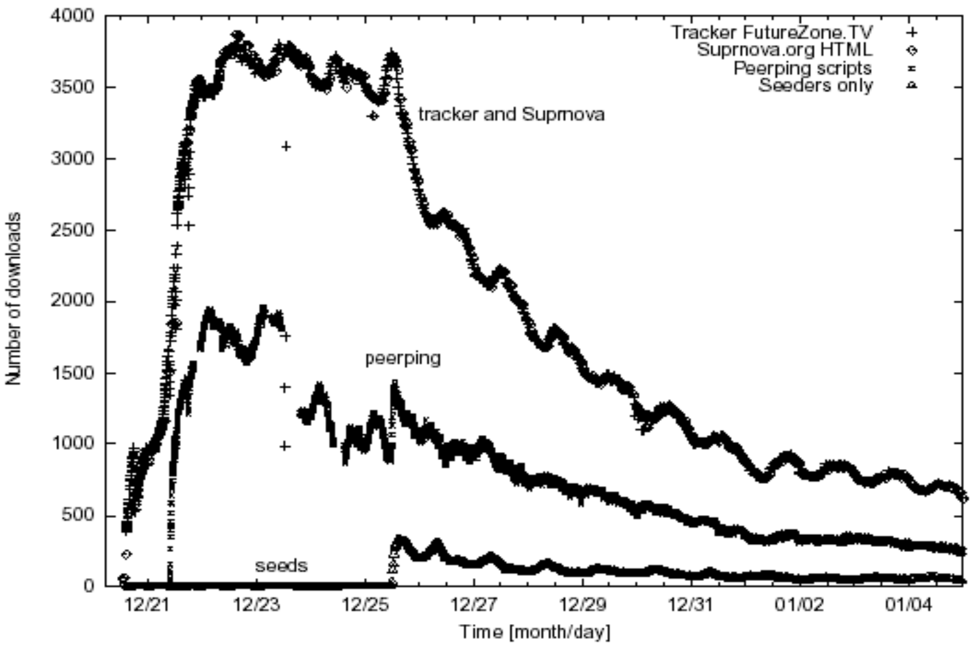
- Assume $c_0 > (C_0+\Sigma C_i)/n$
- Tree i:
  server → client i: $c_i/(n-1)$
  client i → other n-1 clients

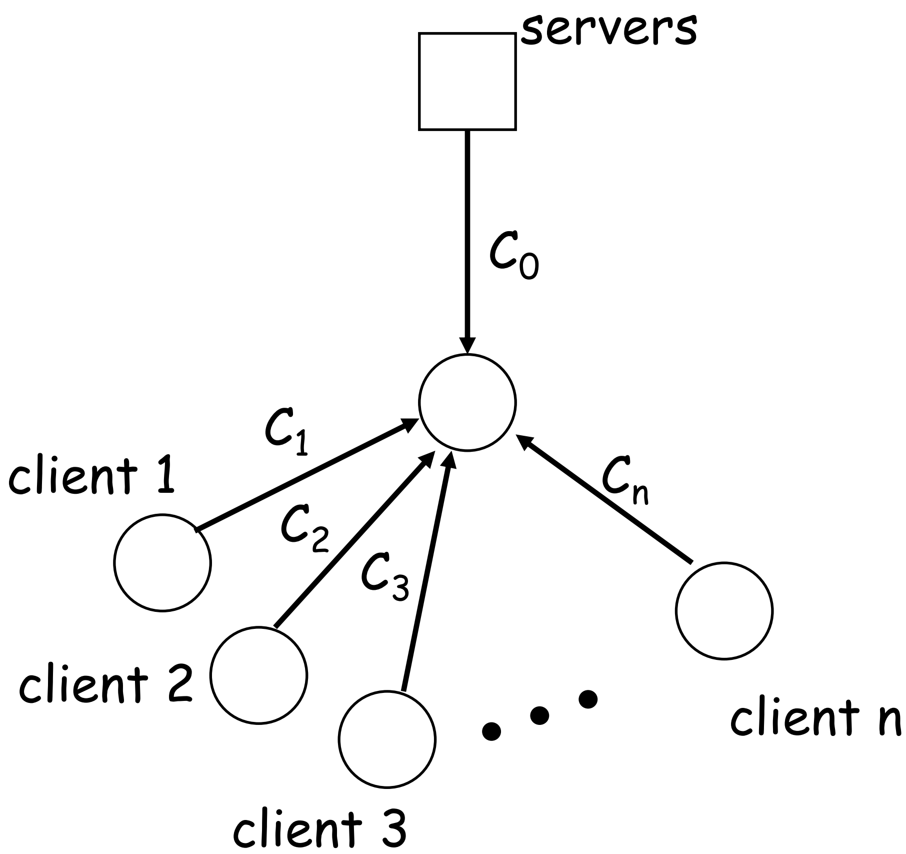


- Tree 0:
  server has remaining
  $c_m = c0 - (c1 + c2 + \ldots cn)/(n-1)$
  send to client i: c_m/n



Why not Building the Trees?

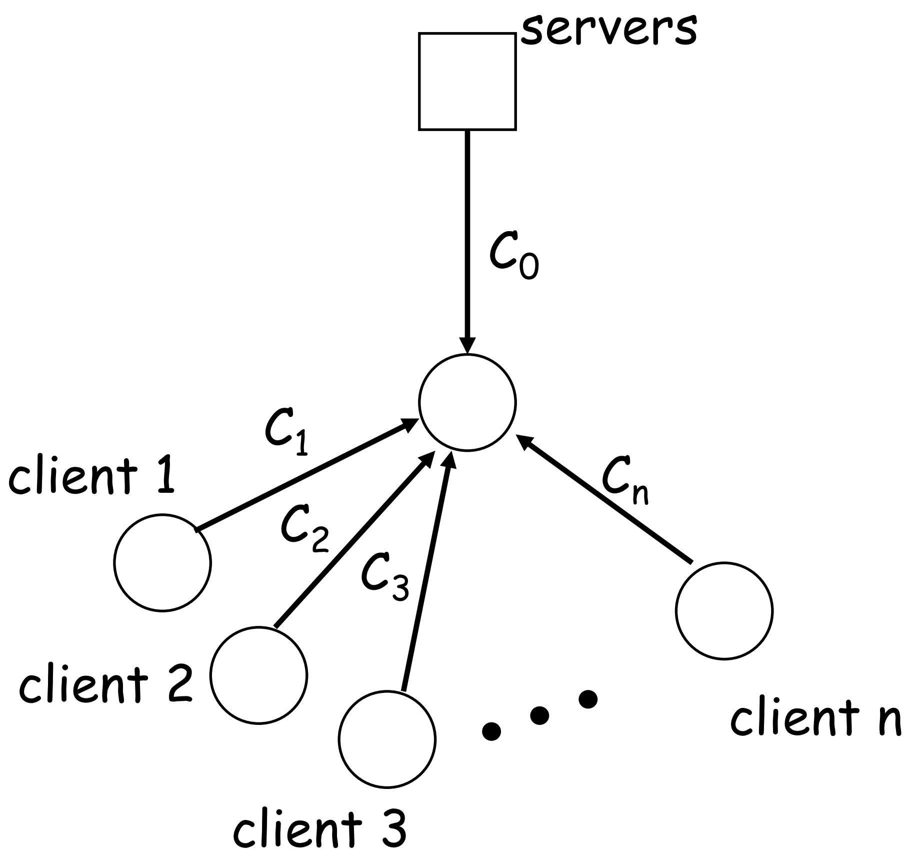




- Clients come and go (churns): maintaining the trees is too expensive
- Each client needs N connections

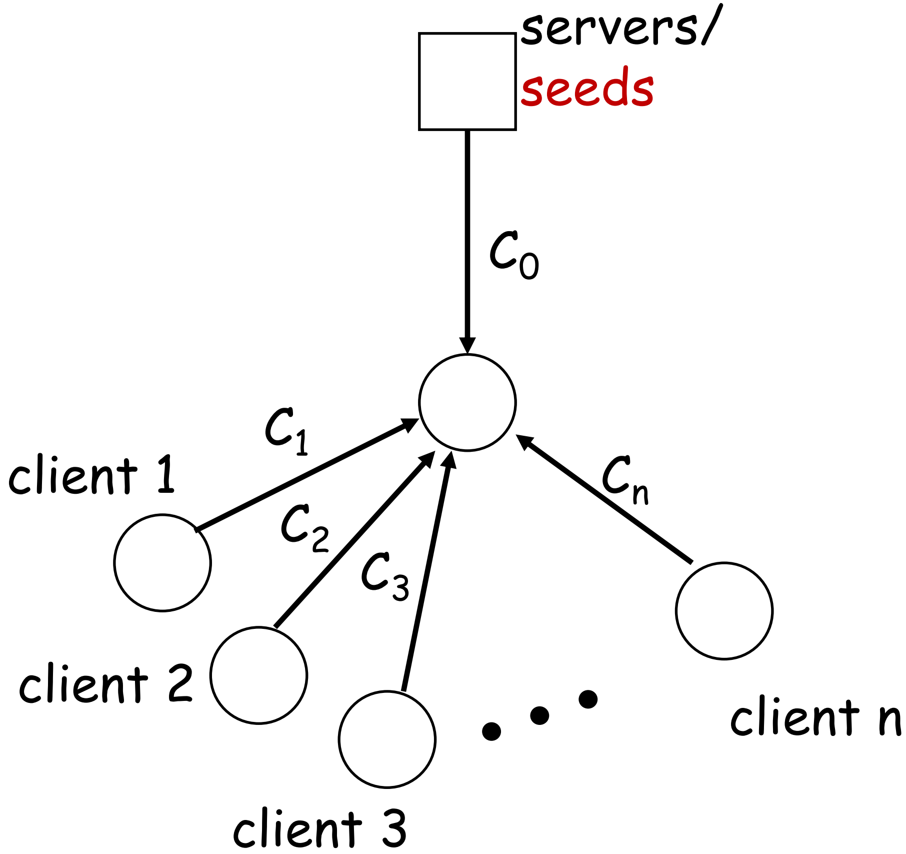
Server+Host (P2P) Content Distribution: Key Design Issues

- 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



Discussion: How to handle the issues?

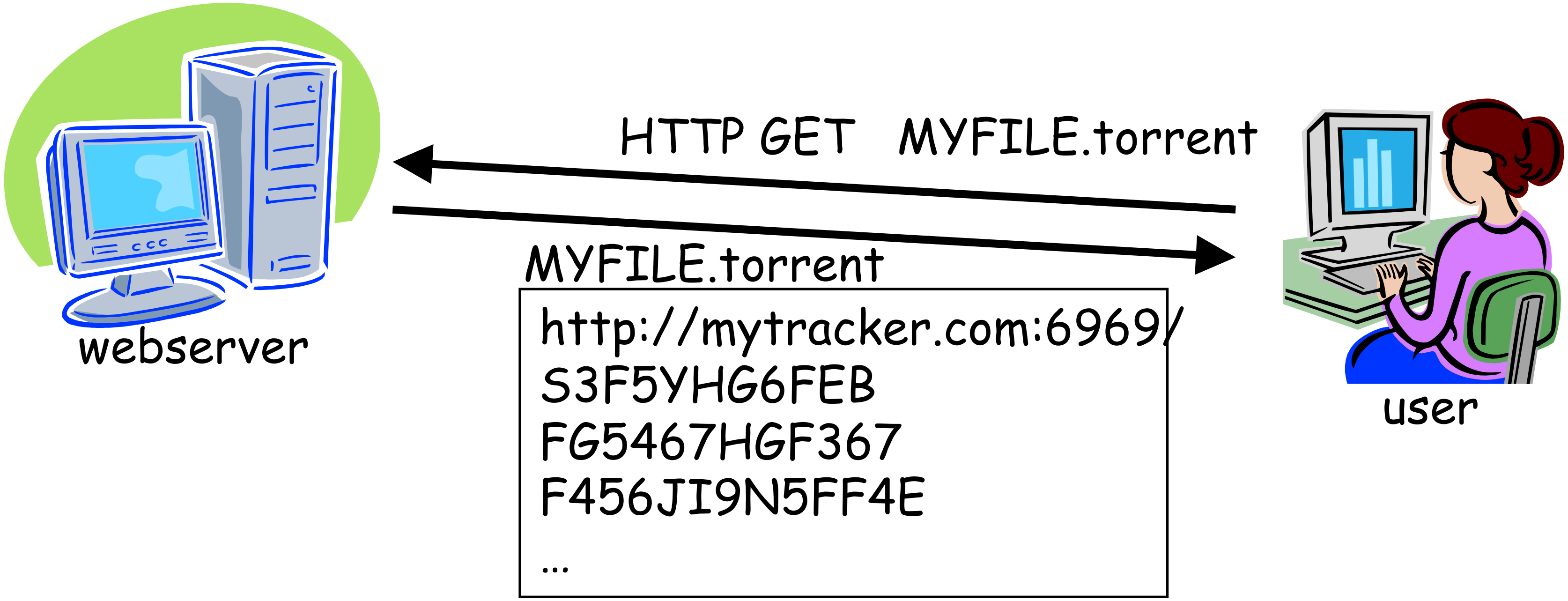
- Robustness
- Efficiency
- Incentive



Example: BitTorrent

- A P2P file sharing protocol
- Created by Bram Cohen in 2004
  - Spec at bep_0003: http://www.bittorrent.org/beps/bep_0003.html

BitTorrent: Lookup



Metadata (.torrent) File Structure

- Meta info contains information necessary to contact the tracker and describes the files in the torrent
  - URL of tracker
  - file name
  - file length
  - piece length (typically 256KB)
  - SHA-1 hashes of pieces for verification
  - also creation date, comment, creator, …

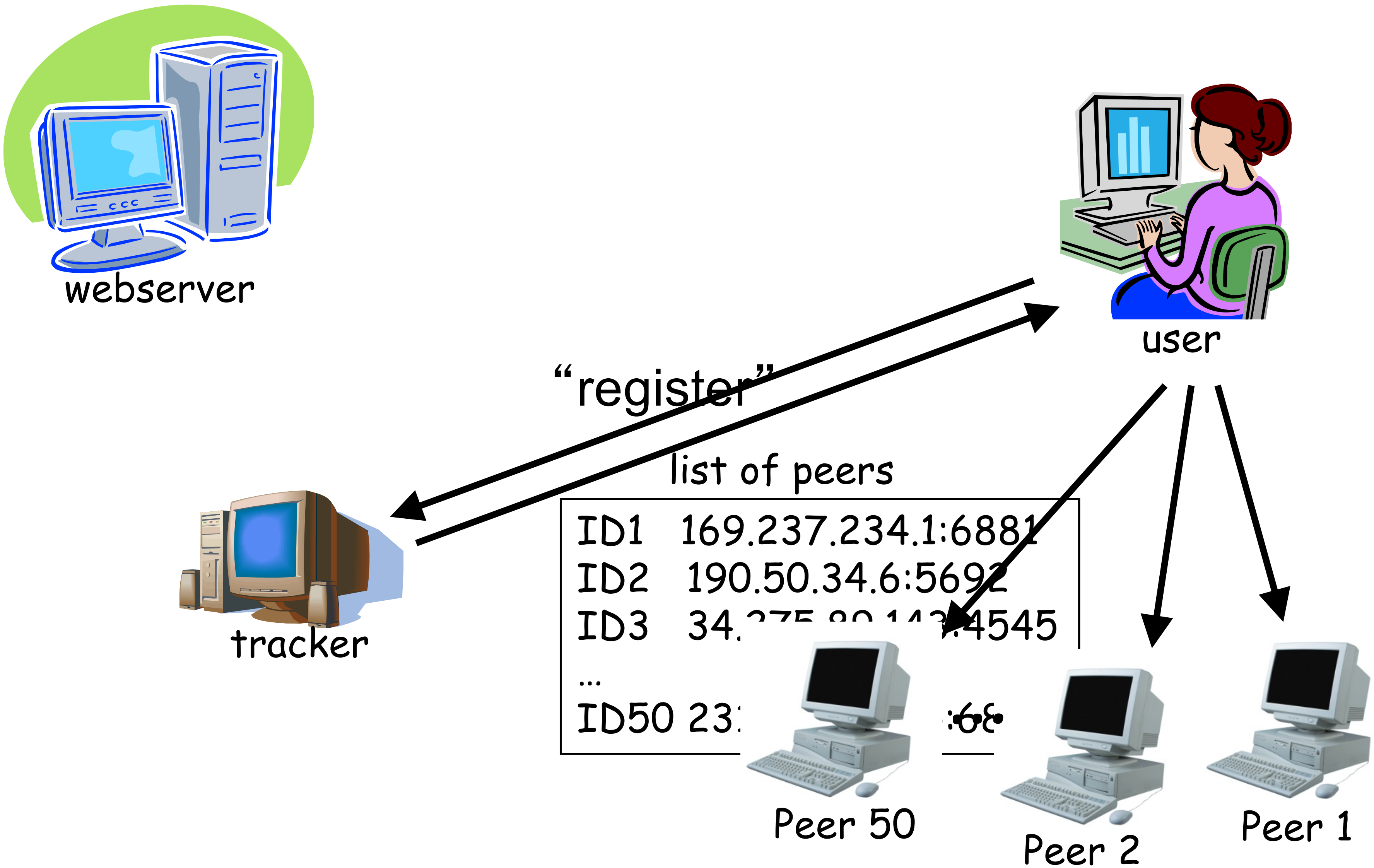
Tracker Protocol

- Communicates with clients via HTTP/HTTPS

- Client GET request
  - info_hash: uniquely identifies the file
  - peer_id: chosen by and uniquely identifies the client
  - client IP and port
  - numwant: how many peers to return (defaults to 50)
  - stats: e.g., bytes uploaded, downloaded

- Tracker GET response
  - interval: how often to contact the tracker
  - list of peers, containing peer id, IP and port
  - stats

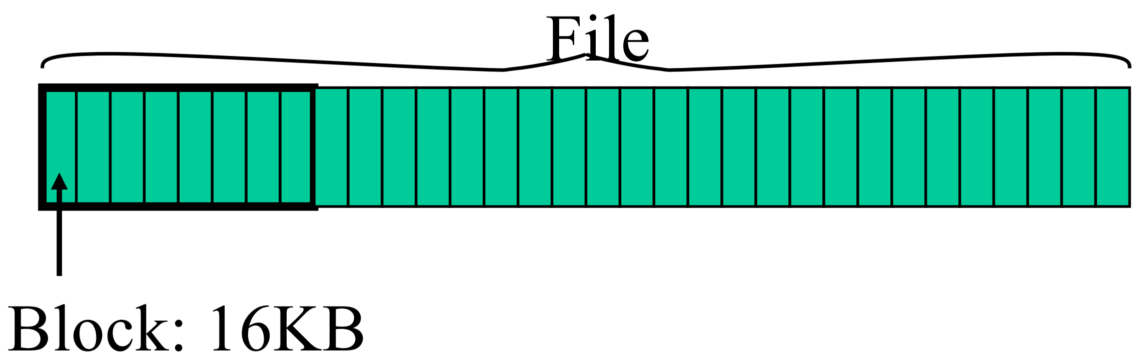
Tracker Protocol



Robustness and efficiency: Piece-based Swarming

- Divide a large file into small blocks and request block-size content from different peers (why?)

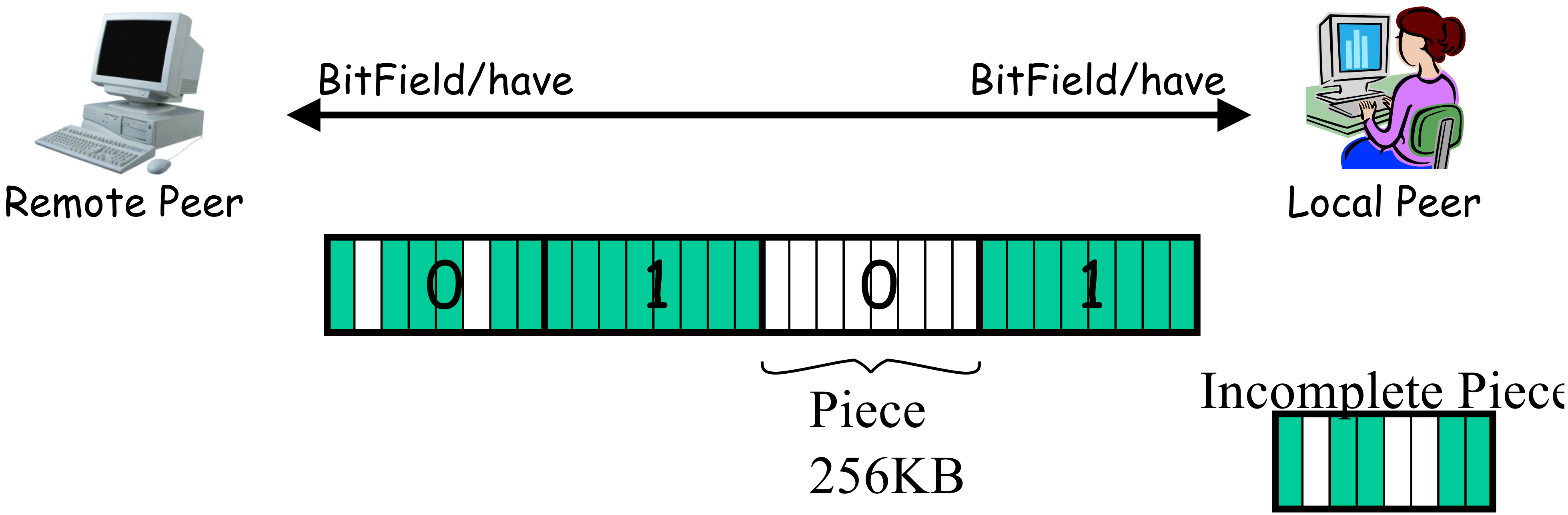
Block: unit of download



- If do not finish downloading a block from one peer within timeout (say due to churns), switch to requesting the block from another peer

Detail: Peer Protocol

(Over TCP)

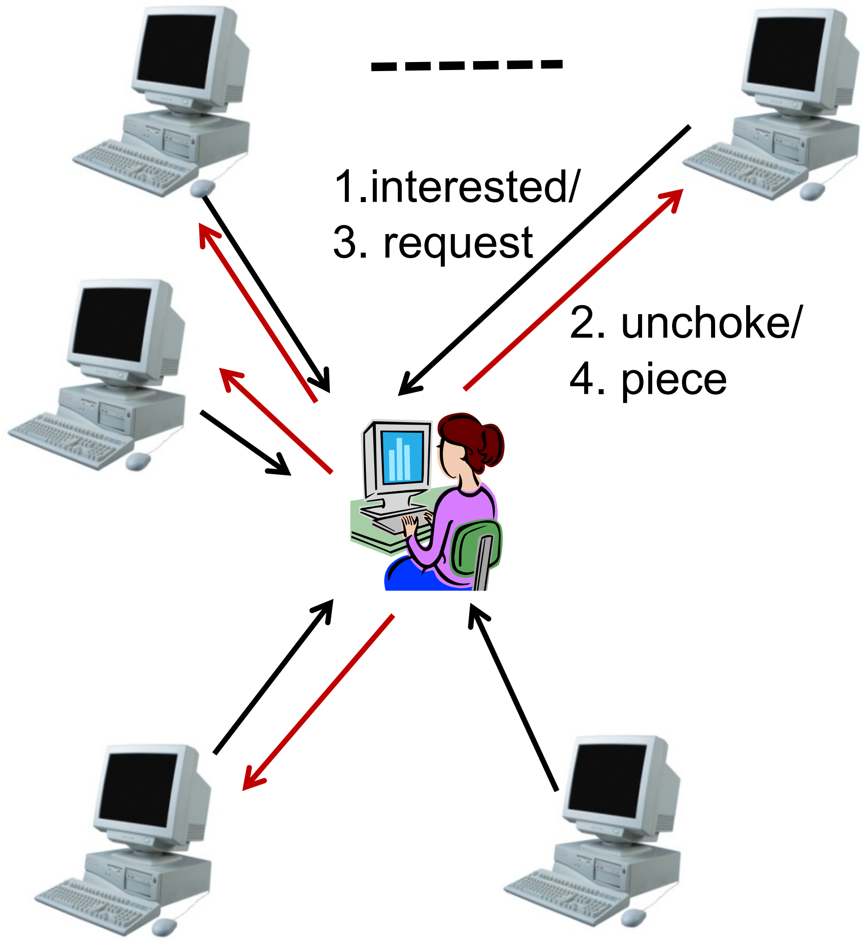


- Peers exchange bitmap representing content availability
  - `bitfield` msg during initial connection
  - `have` msg to notify updates to bitmap
  - to reduce bitmap size, aggregate multiple blocks as a piece

Peer Request

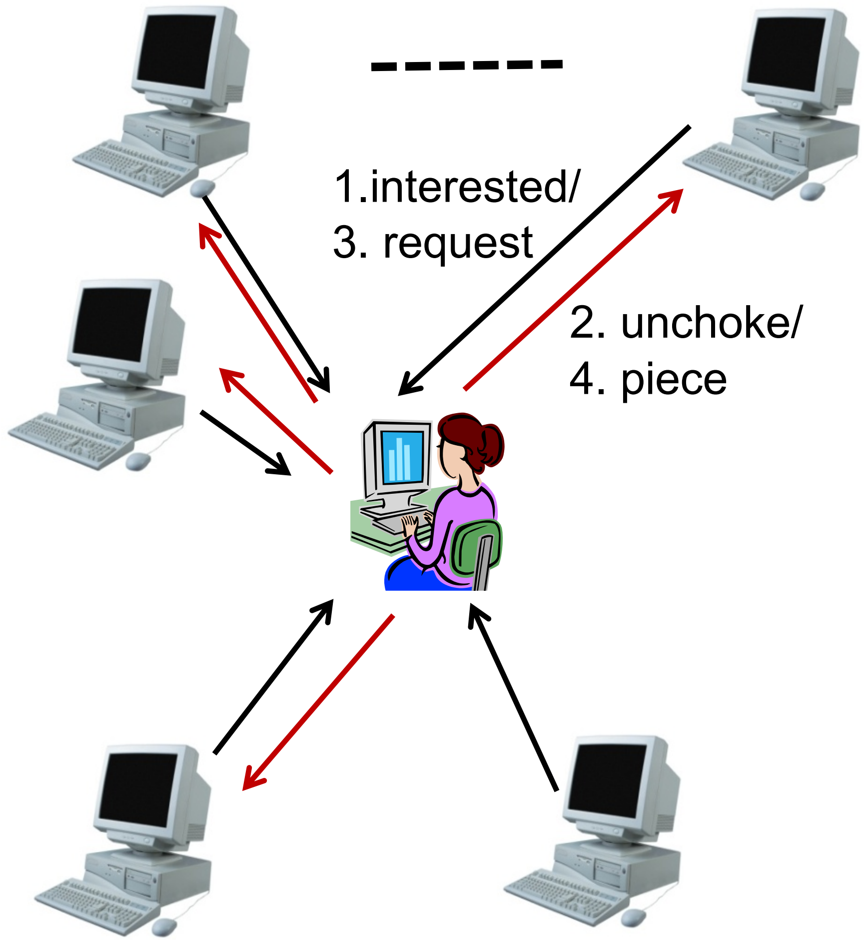
http://www.bittorrent.org/beps/bep_0003.html

- If peer A has a piece that peer B needs, peer B sends `interested` to A
- `unchoke`: indicate that A allows B to request
- `request`: B requests a specific block from A
- `piece`: specific data



Key Design Points

- request:
  - which data blocks to request?
- unchoke:
  - which peers to serve?



Request: Block Availability

- Request (local) rarest first
  - achieves the fastest replication of rare pieces
  - obtain something of value

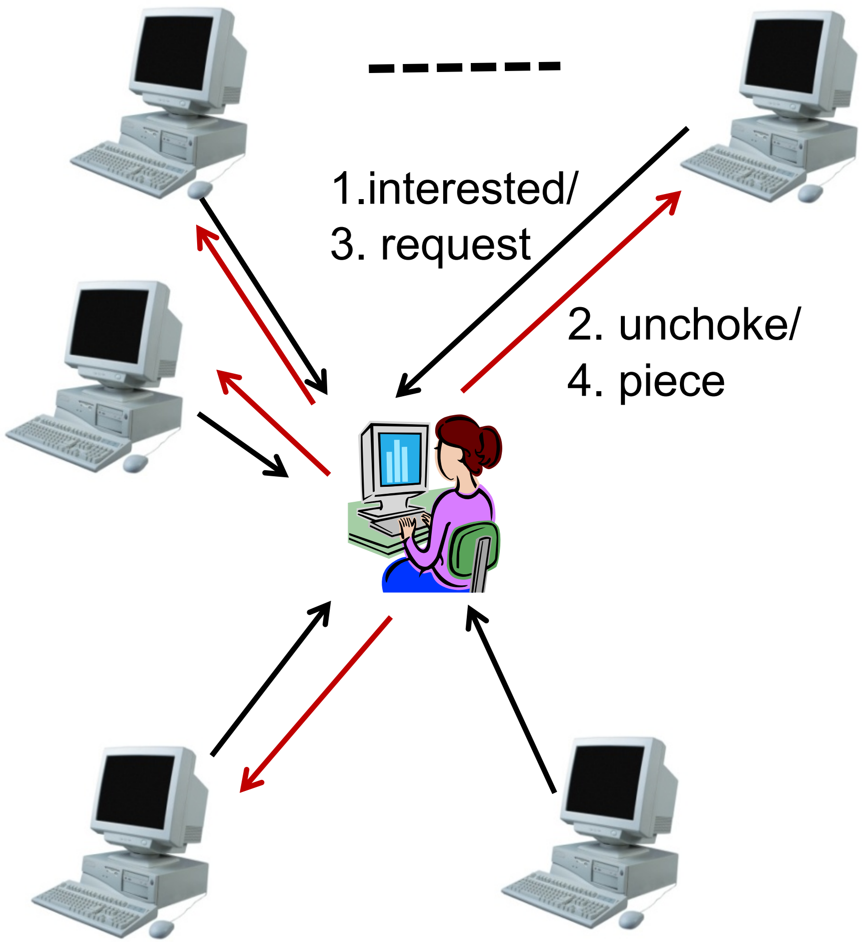
Block Availability: Revisions

- When downloading starts (first 4 pieces): choose at random and request them from the peers
  - get pieces as quickly as possible
  - obtain something to offer to others

- Endgame mode
  - defense against the "last-block problem": cannot finish because missing a few last pieces
  - send requests for missing pieces to all peers in our peer list
  - send `cancel` messages upon receipt of a piece

BitTorrent: Unchoke

- Periodically (typically every 10 seconds) calculate data-receiving rates from all peers
- Upload to (*unchoke*) the fastest
  - constant number (4) of unchoking slots
  - partition upload bw equally among unchoked



commonly referred to as "tit-for-tat" strategy

Optimistic Unchoking

- Periodically select a peer at random and upload to it
  - typically every 3 unchoking rounds (30 seconds)

- Multi-purpose mechanism
  - allow bootstrapping of new clients
  - continuously look for the fastest peers (exploitation vs exploration)

BitTorrent Fluid Analysis

- Normalize file size to 1
- x(t): number of downloaders (also known as leechers) who do not have all pieces at time t.
- y(t): number of seeds in the system at time t.
- λ: the arrival rate of new requests.
- μ: the uploading bandwidth of a given peer.
- c: the downloading bandwidth of a given peer, assume $c \geq \mu$.
- θ: the rate at which downloaders abort download.
- γ: the rate at which seeds leave the system.
- η: indicates the effectiveness of downloader sharing, η takes values in [0, 1].

System Evolution

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \lambda - \theta x(t) - \min\{cx(t), \mu(\eta x(t) + y(t))\},$$
$$\frac{\mathrm{d}y}{\mathrm{d}t} = \min\{cx(t), \mu(\eta x(t) + y(t))\} - \gamma y(t),$$

Solving steady state: $\frac{\mathrm{d}x(t)}{\mathrm{d}t} = \frac{\mathrm{d}y(t)}{\mathrm{d}t} = 0$

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

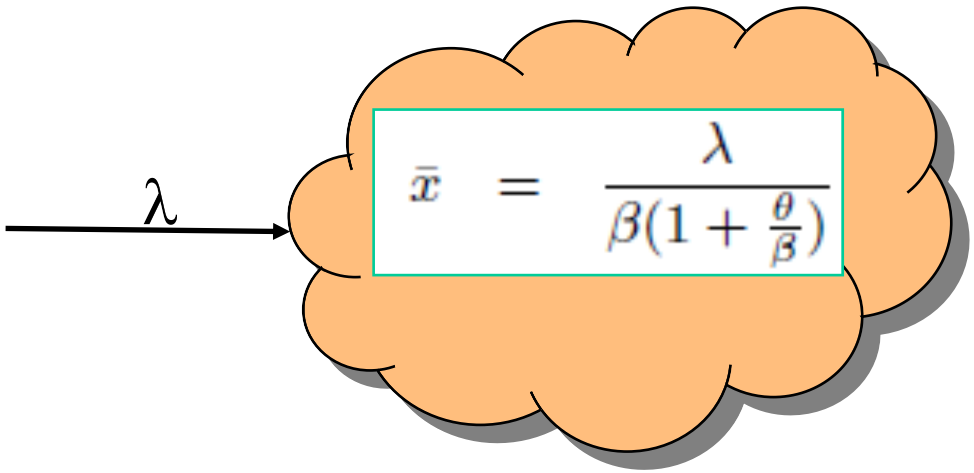
$$\boxed{\begin{aligned} \bar{x} &= \frac{\lambda}{\beta(1 + \frac{\theta}{\beta})} \\ \bar{y} &= \frac{\lambda}{\gamma(1 + \frac{\theta}{\beta})}. \end{aligned}}$$

"Modeling and Performance Analysis of BitTorrent-Like Peer-to-Peer Networks", SIGCOMM'04
https://conferences.sigcomm.org/sigcomm/2004/papers/p444-qiu1.pdf

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

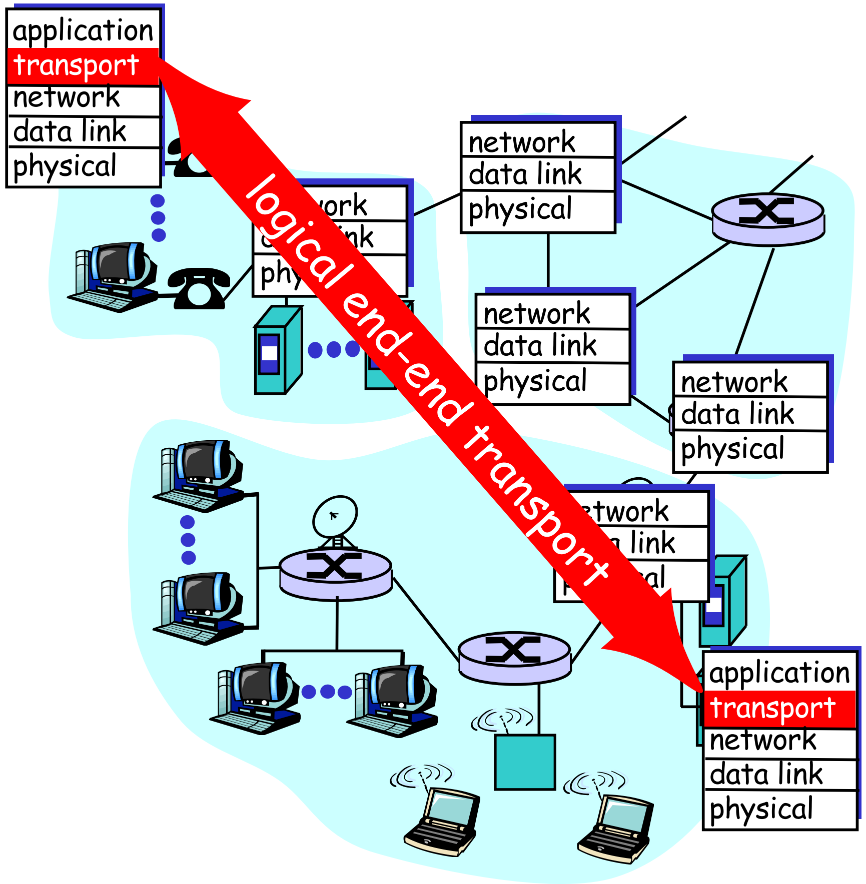
- Applications
  - Client-server applications
    - Single server
    - Multiple servers load balancing
  - Application overlays (distributed network applications) to
    - scale bandwidth/resource (BitTorrent)
    - distribute content lookup (Freenet, DHT, Chord) [optional]
    - distribute content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

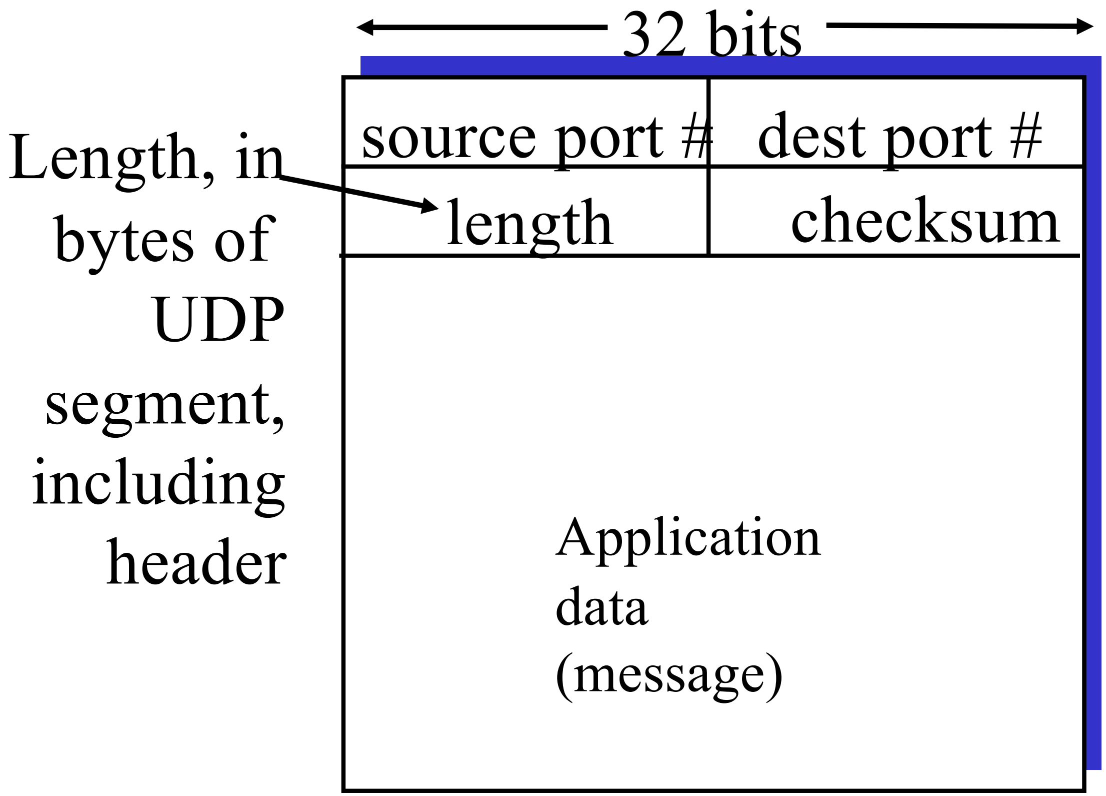
- Class 1 (today):
  - transport layer services
  - connectionless transport: UDP
  - reliable data transfer using stop-and-wait/alternating-bit protocol
- Class 2 (ready for lab assignment 4/part 1):
  - sliding window reliability
  - TCP reliability
    - overview of TCP
    - TCP RTT measurement
    - 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:

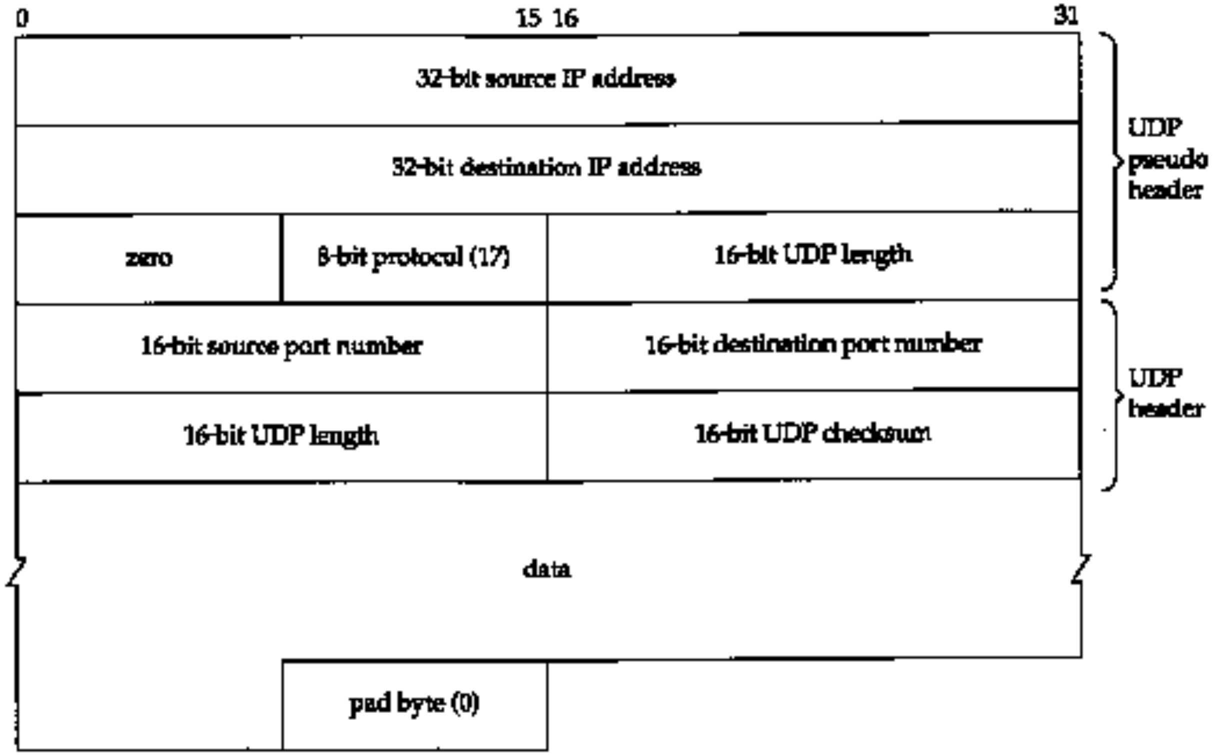
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

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

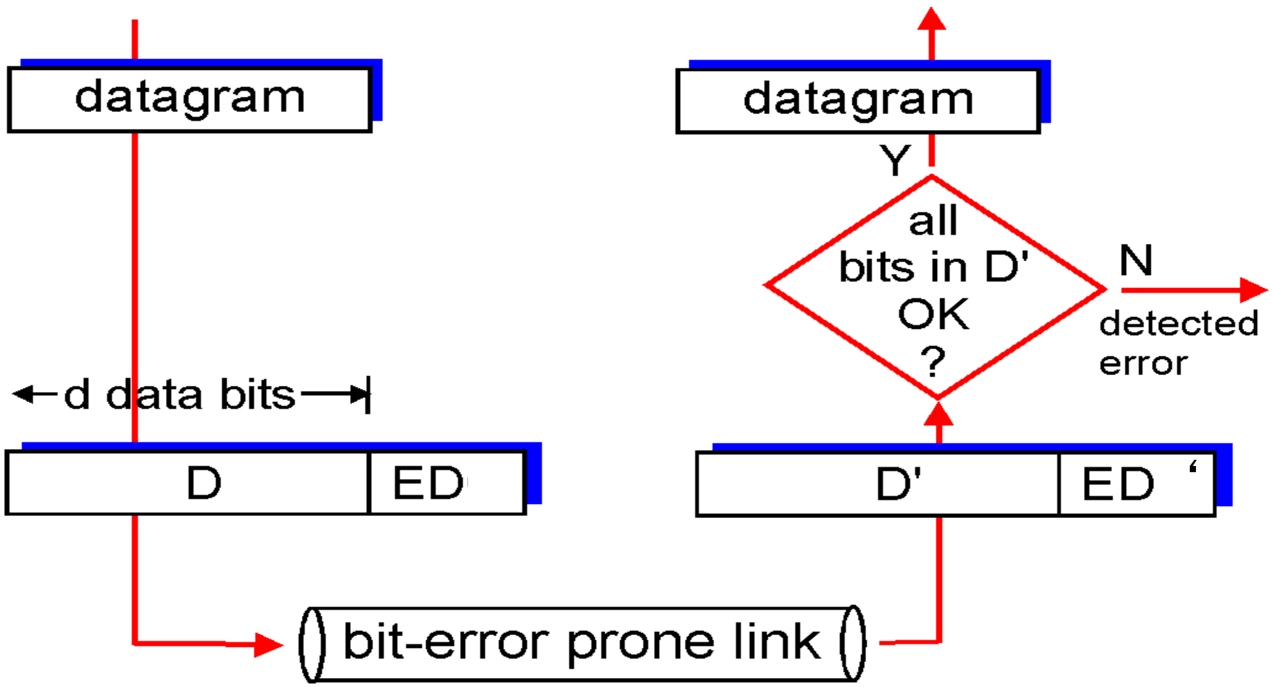
UDP Checksum: Coverage

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)

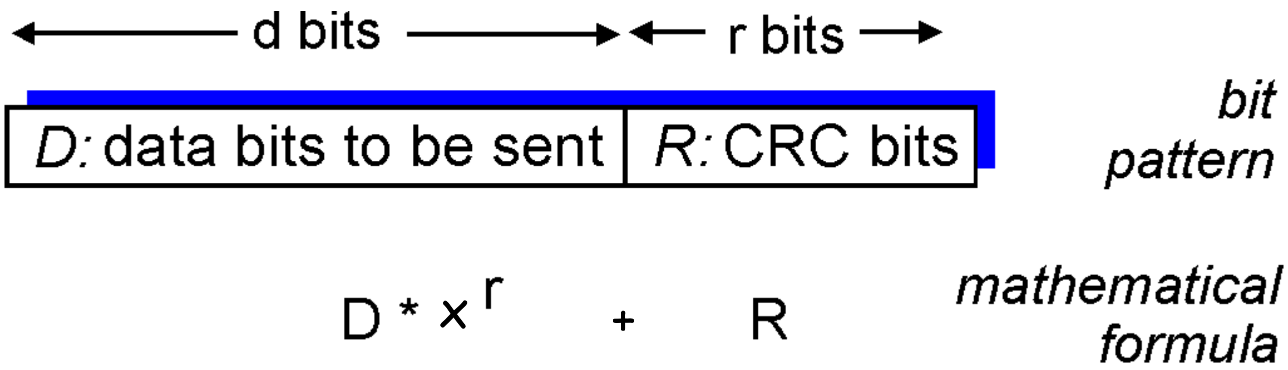
- Error detection not 100% reliable!
    - a good error detector may miss some errors, but rarely
    - larger ED field generally yields better detection

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)

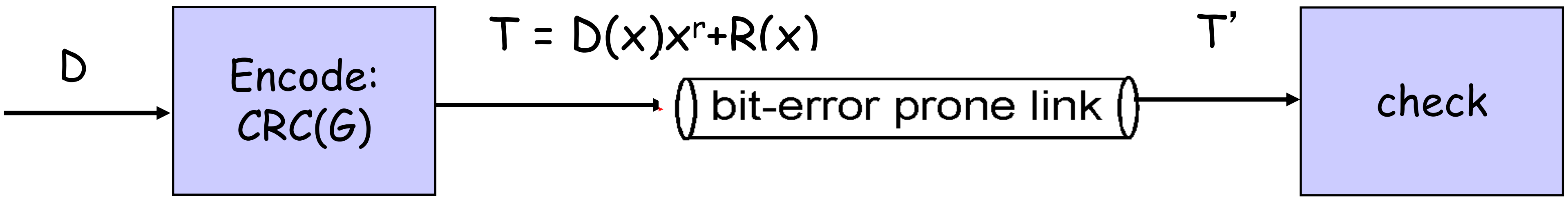
Cyclic Redundancy Check: Encode

- 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

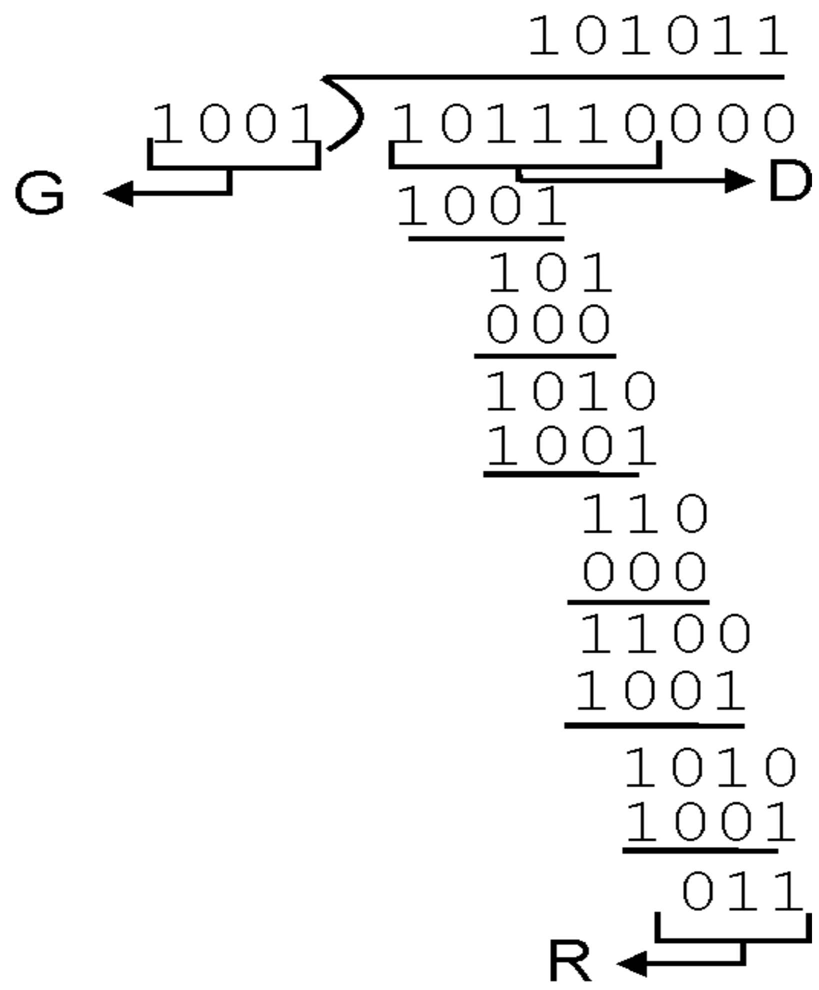
Cyclic Redundancy Check: Decode



- 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

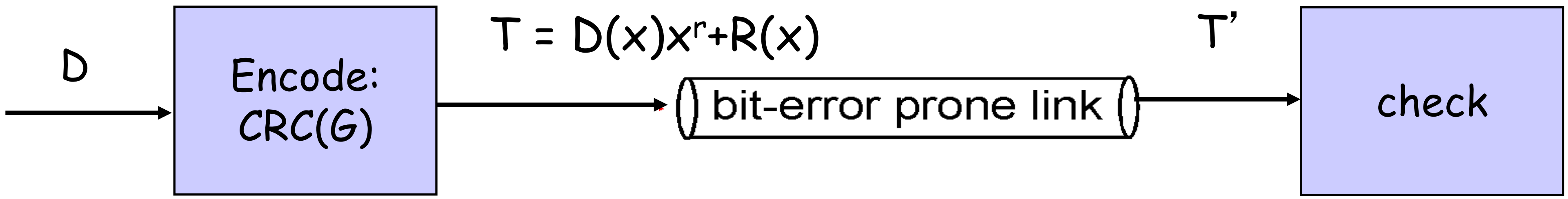
CRC: Steps and an Example

- 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

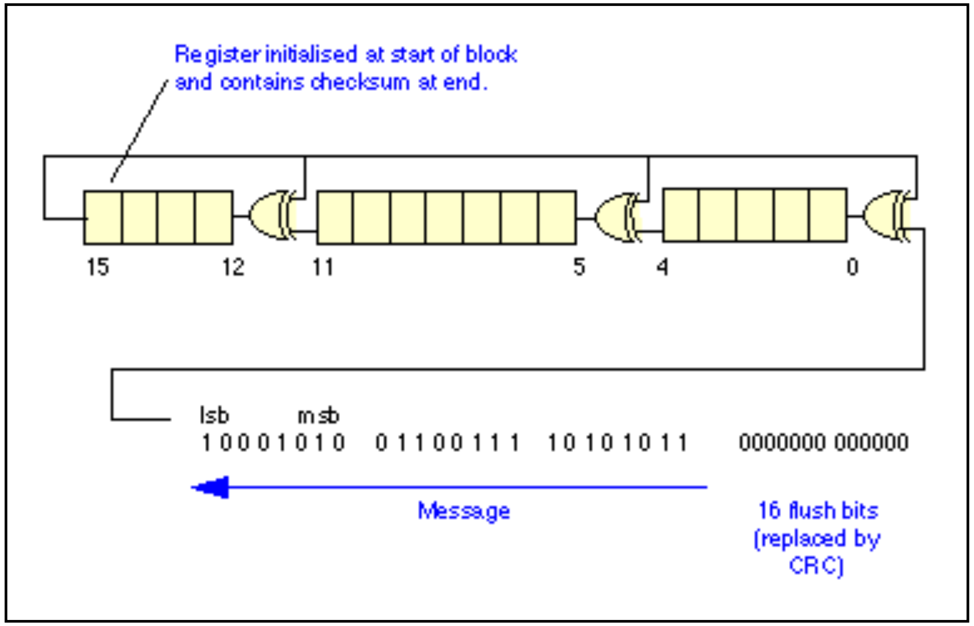
- Detect a single-bit error: $E(x) = x^i$
  - if $G(x)$ contains two or more terms, $E(x)$ is not divisible by $G(x)$

- 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

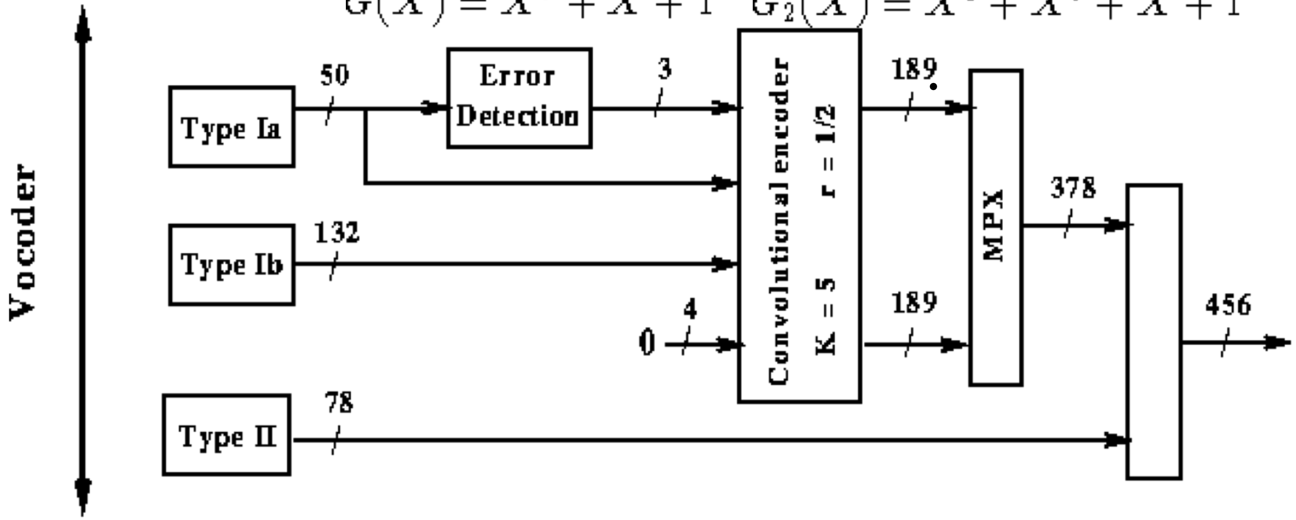
http://en.wikipedia.org/wiki/CRC-32#Implementation

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