Washington University in St. Louis

Engineering

9. Peer to Peer Applications

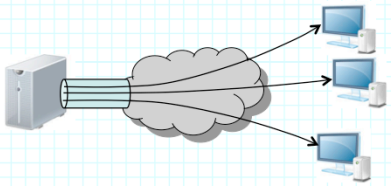
- Peer-to-peer file sharing – BitTorrent
- Distributed hash tables

Jon Turner

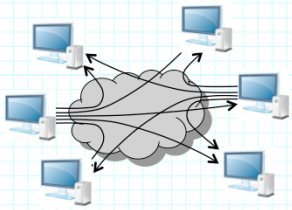
Client-Server vs. Peer-to-Peer

- Cient-server applications
  - » servers typically operated by a well-known organization that is well-motivated to operate responsibly
- Peer-to-peer applications
  - » "peers" are typically computers owned by individuals
    - little basis for trust, users have mixed motivations
  - » technical advantage: group of peers can complete transfer of a large file faster than can a single server
  - » cost advantage: no expensive server, peers are "free"
  - » legal/ethical concerns: can be enabler for large-scale piracy
- Why is peer-to-peer faster?
  - » server sending same file to *N* hosts must send every bit *N* times
  - » peers can forward pieces of file to other peers, as they arrive
    - so, total sending rate grows with the number of peers

Simplified Transfer Time Analysis



- Client server case
  - » server upload rate: u_s b/s
  - » client download rate: d_c b/s
  - » # of clients: N
  - » file size: F bits
  - » time to transfer file to all clients:
    $T=\max\{FN/u_s, F/d_c\}$ seconds
    so, if u_s=1 Gb/s, d_c=1Mb/s, $N=10^4$, F=10 Gb, T=100,000 seconds



- Peer-to-peer case
  - » peer upload rate: u_p b/s
  - » peer download rate: d_p b/s
  - » originating peer sends piece of file to every other peer
  - » each peer, sends its piece to all others
  - » if $u_p<d_p$ then
    $T=F/u_p$ seconds
    so, if u_p=500 Kb/s, F=10 Gb, T=20,000 seconds

3

Understanding the P2P Speedup

- If T_S is file transfer time using server and T_P is transfer time for p2p, T_S/T_P is the peer-to-peer *speedup*

$$\max\{FN/u_s, F/d_c\}/F/u_p = u_p \max\{N/u_s, 1/d_c\}$$

- So, when there are many peers ($N > u_s/d_c$)

$$speedup = Nu_p/u_s$$

  - » that is, what matters most is the total upload rate, whether it comes from a single server or from N peers
  - » well-provisioned server may well outperform a small number of peers
  - » for large enough N, peer-to-peer approach can be faster
- P2P methods have been used for distributing software updates (and other large files) in data centers
  - » many peers, with high upload rates

Exercises

1. Suppose a movie studio wants to distribute a new movie as a digital file to 1,000 movie theaters across country. Assume the file is 5 GB long, and that the studio's server has 1 Gb/s internet connection and that the theaters each have a 5 Mb/s DSL connection. Approximately, how much time is needed to distribute the file to all the servers, using the client-server method?
2. Reconsider the scenario from the last question, assuming peer-to-peer distribution is used. To simplify, assume that both the studio's server and the theaters have DSL connections with a 5 Mb/s download rate and a 2 Mb/s upload rate. What is the total "upstream bandwidth" in this case? How long does it take to distribute the file to all of the theaters, under ideal conditions?
3. The analysis for the peer-to-peer case assumes idealized conditions, in which each peer gets a portion of the file from the server, then redistributes that part to all others. This requires that each peer maintain large numbers of simultaneous connections. Suppose each peer is limited to 10 simultaneous connections. Is it still possible to achieve the idealized transfer time? If so, how? If not, why not?

BitTorrent

- In a real file-distribution system, peers come and go over time and may be greedy
  - » protocol must accommodate changing set of peers
  - » and must encourage "socially beneficial" behavior
- In BitTorrent
  - » to distribute a file, a source creates a *torrent file* that contains metadata about the file and the URL of a *tracker* site
  - » new peers register with the tracker, which provides each new peer with a list of other peers (a random subset)
  - » files are broken up into smaller *chunks* and the chunks are the basic unit of distribution
  - » a peer connects to others peers, learns what chunks they have and then requests the chunks it doesn't have
  - » a host will typically have tens of peers
    - the set of peers changes continuously

Some Details

- Peers request the *rarest chunks first*
  - » this increases number of copies of rare chunks, enabling more peers to get a copy
- A peer responds to chunk requests by favoring other peers that have supplied it with the most data recently
  - » a peer measures number of bits received from its neighbors and preferentially responds to requests from the "best suppliers"
  - » a peer also randomly selects other neighbors and responds to their requests in order to expand set of "trading partners"
- Torrent files contain crytpographic hashes of each file chunk to prevent malicious corruption of files
  - » peers compute the hash of chunks they receive and compare against the torrent file, discarding chunks that don't match
- Variety of BitTorrent clients, with varying behavior

Exercises

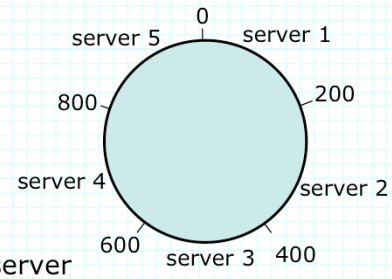
1. At any moment in time, the peer-to-peer connections in a torrent form a graph. How does the diameter of this graph affect the time taken to distribute a given chunk? Suppose the connections among the hosts are purely random and that each has 20 peers. Estimate the diameter of the graph, assuming there are 10,000 hosts in the torrent.
2. The rarest-chunk-first policy tries to increase the availability of chunks held by only a small number of hosts. Describe a situation in which this policy may not have the desired effect. In general, how is the effectiveness of this policy affected by the number of peers that each host has at one time?
3. Consider a network that supports multicast communication, in which a single host can send packets to many receivers simultaneously (with the network routers copying packets as necessary). Write an expression for the time to transfer an F bit file to N clients using such a network. Express your result in terms of the server and client upload/download rates. Compare to the original client-server analysis and to the peer-to-peer analysis.

Distributed Hash Tables

- A hash table can implement a *map* data structure
  - » stores set of (*key*, *value*) pairs, where each *key* is unique
  - » *get*(*key*) operation returns associated *value*,
  - » *put*(*key*,*value*) adds the pair (*key*,*value*) to the set, possibly replacing an existing pair
- In a DHT, the (*key*,*value*) pairs are distributed among a large (possibly changing) collection of servers
  - » typically, each server is responsible for a sub-set of the keys, determined by a hash function applied to the key value
    - e.g., a server might handle keys k, for which $1 \leq h(k) \leq 1000$
  - » seeks to balances the load among servers roughly equally
- Key design issues for DHTs
  - » how to find server for a given key
  - » how to reconfigure DHT as servers come and go

9

Simple Circular DHT



- Assign each server a range of hash values
- Each server "knows" IP address of its successor
- A "client" can query the DHT by
  - » sending *get*(*key*) message to any server
  - » the server computes *h*=*hash*(*key*)
    - if *h* is in server's range of hash values, it does a lookup in its local data structure (typically a hash table) and responds with value
    - otherwise, it forwards message to its successor
    - when message reaches "responsible server", it responds
- For a DHT with *n* servers, can take *O*(*n*) time to respond
  - » also, each server does some work for about half the queries
    - so, not much higher query volume than a single server

Adding Shortcuts to Circular DHT

- Can improve performance dramatically by having each server store IP address of multiple other servers
  - » if each server knows address and range of *all* other servers, at most two servers are involved in responding to each query
    - so *n* server DHT can process about *n*/2 times as many queries as a single server
    - can be reasonable choice if *n* is not too large and servers are reasonably stable
  - » alternatively, use lg *n* (=$\log_2 n$) shortcuts
    - each server stores routes for servers that are 1, 2, 4, 8,... hops away
    - when forwarding query, send directly to "nearest known server"
    - so, each hop reduces circular distance to destination by at least a factor of 2 – implies at most lg *n* hops
  - » intermediate strategies also possible

Choosing Shortcuts

- With $(2n)^{1/2}$ routes per server can space more closely
  - » choose routes to servers that are 1, 2, 4, 7, 11, 16,... hops away (differences increase by 1)
  - » this way, each hop reduces the circular distance to destination by a factor of $(n/2)^{1/2}$ – so $O(\lg \lg n)$ hops
- Learn routes by requesting them from other servers
  - » by repeatedly querying "furthest-known-server", can acquire $\lg n$ routes in $\lg n$ "rounds"
- Or, acquire routes passively by observing responses
  - » responses routed through "first-contact-server"

Exercises

1. Consider a circular DHT with 10 nodes numbered 0..9 and that server i is responsible for (*key*,*value*) pairs with hashes in the range [10*i*,10(*i*+1)). Also, assume that if *i* is a prime number, there is a shortcut route from i to i+3 mod 10. For all other i, there is a shortcut route from i to i+5 mod 10. Draw a graph of this DHT, showing all routes. Suppose node 6 receives a *get* request for a key that hashes to 47. What sequence of nodes does this request pass through before going back to the client?
2. Consider a DHT with 1,000 servers. Assume that each has a 100 Mb/s connection to the internet, and that it can process 100K packets per second (this includes receiving the packet, processing it and either sending a reply or forwarding it to another host). Assume that each server has routes to those hosts that are 1, 2, 4, 8,..., 512 hops away, but that it does not cache (*key*,*value*) pairs. What is the maximum number of requests this DHT can respond to in a second? What traffic pattern produces this peak performance? What is the minimum number of requests it can respond to in a second? What traffic pattern produces this performance? What is the minimum if each server is the target for an equal number of requests?

Adding and Removing Servers

- In some DHT applications, servers come and go frequently
  - » even in more stable applications, servers may crash or be taken out of service for maintenance
- To maintain routing information, must adjust successor information
  - » requires some redundancy: each server tracks suc_1, suc_2
    - servers send "are you there" messages to suc_1, suc_2 periodically to detect departures
  - » suppose server s, having predecessors p_1 and p_2 leaves
    - p_1 obtains information about new $suc_2(p_1)$ from (old) $suc_2(p_1)$ and p_2 obtains information about new $suc_2(p_1)$ from p_1
  - » to insert server s following p_1 (with predecessor p_2)
    - p_1 informs s about $suc_1(p_1)$ and $suc_2(p_1)$; p_1 informs p_2 about s

Maintaining Data in Dynamic DHTs

- To avoid data loss, each server maintains copy of its successor's data
  - » allows departing server's predecessor to take over its role in the DHT (expanding its range of hash keys)
  - » requires that copy of data owned by departing server now be transferred to its predecessor's predecessor
  - » when server *s* joins DHT following node *p*, *p* splits its range of hash key values and sends associated data to *s*
  - » *s* obtains copy of data owned by *suc*(*s*) and *p* may now discard redundant data it must no longer maintain
- Redundant data must also be maintained during normal update operations
  - » requires responsible server to inform its predecessor of updates
- Server holding copy of data can also respond to *gets*

Caching in DHTs

- DHTs perform best if query load is distributed evenly among servers
  - » use of hashing contributes to load balancing
  - » but overloads can occur if some data items are much more "popular" than others
  - » can improve performance of popular items by distributing copies to other servers and letting them respond to queries
- One approach
  - » *target-server t* sends query response to *first-contact-server f*, allowing *f* to respond to client and place copy in its cache
    - subsequent queries through *f* can be handled directly
    - *f* could also respond to queries during forwarding
  - » this approach can quickly distribute many copies of popular items, enabling higher query rates for such items
  - » cached copies removed if timestamp expires, or space needed

16

Exercises

1. Consider a DHT with 1000 servers, in which servers are added and removed as described on slide 12.
   - » What happens if two adjacent servers leave the DHT at the same time?
   - » Suppose that servers leave the DHT only when they fail, that the average time between failures for one server is one month and that the time to update the DHT when a server fails is 10 seconds. How often will an update fail?
   - » Suppose server owners collude to disrupt the DHT by joining, then leaving in a coordinated fashion? How might you protect the DHT from such behavior?
2. Consider a DHT that starts with a single server *A*, with a hash range of 0-1023. Suppose that whenever a new server joins the DHT, the target of the join splits its hash range in half, sending the second half to the new server. Now suppose 9 servers join *A*'s DHT, and that they all do so by sending a join message to A. Draw a circular diagram of the resulting 10 node DHT showing the hash range for each of the servers. Suggest a modification of the join procedure that would produce a more even distribution of the hash ranges?