Overview

- Web
  - Protocol interactions
  - Caching
  - Cookies
  - Consistent hashing
  - Peer-to-peer
  - CDN
  - Video

Web history

  - Describes the idea of a distributed hypertext system.
  - A “memex” that mimics the “web of trails” in our minds.
- 1989: Tim Berners-Lee (CERN) writes internal proposal to develop a distributed hypertext system
  - Connects “a web of notes with links”.
  - Intended to help CERN physicists in large projects share and manage information.

Web history (cont)

- 1992
  - NCSA server released
  - 26 WWW servers worldwide
- 1993
  - Marc Andreessen releases first version of NCSA Mosaic
  - Web (port 80) traffic
  - Over 200 WWW servers worldwide
- 1994
  - Andreessen and (Communications
Typical Workload (Web Pages)

- Multiple (typically small) objects per page
- File sizes
  - Heavy-tailed
  - Pareto distribution for tail
  - Lognormal for body of distribution
- Embedded references
- Number of embedded objects also Pareto
  \[ Pr(X>x) = \left(\frac{x}{x_m}\right)^{-k} \]
- This plays havoc with performance. Why?
- Solutions?

HTTP 0.9/1.0

- One request/response per TCP connection
  - Simple to implement
- Disadvantages
  - Multiple connection setups → three-way handshake each time
  - Several extra round trips added to transfer
  - Multiple slow starts

Must Consider Network Layer - Example: TCP

1: TCP connections need to be set up
   - “Three Way Handshake”:
   - Client opens TCP connection
   - Client sends HTTP request for HTML
   - Client parses HTML
   - Client opens TCP connection
   - Client sends HTTP request for image
   - Image begins to arrive

2: TCP transfers start slowly and then ramp up the bandwidth used – bandwidth is variable

Single Transfer Example
Performance Issues

- Short transfers are hard on TCP
  - Stuck in slow start
  - Loss recovery is poor when windows are small
- Lots of extra connections
  - Increases server state/processing
  - Servers also hang on to connection state after the connection is closed
  - Why must server keep these?
  - Tends to be an order of magnitude greater than # of active connections, why?

Netscape Solution

- Mosaic (original popular Web browser) fetched one object at a time!
- Netscape uses multiple concurrent connections to improve response time
  - Different parts of Web page arrive independently
  - Can grab more of the network bandwidth than other users
  - Does not necessarily improve response time
  - TCP loss recovery ends up being timeout dominated because windows are small

Persistent Connection Solution

- Multiplex multiple transfers onto one TCP connection
  - Also allow pipelined transfers
- How to identify requests/responses
  - Delimiter → Server must examine response for delimiter string
  - Content-length and delimiter → Must know size of transfer in advance
  - Block-based transmission → send in multiple length delimited blocks
  - Store-and-forward → wait for entire response and then use content-length
  - Solution → use existing methods and close connection otherwise

Persistent Connection Solution

Client sends HTTP request for HTML
Client parses HTML
Client sends HTTP request for image
Image begins to arrive
Other Problems

- Serialized transmission
  - Much of the useful information in first few bytes
    - May be better to get the 1st 1/4 of all images than one complete image (e.g., progressive JPEG)
  - Can “packetize” transfer over TCP, e.g., range requests
- Application specific solution to transport protocol problems. :(
  - Could fix TCP so it works well with multiple simultaneous connections
  - More difficult to deploy
- Persistent connections can significantly increase state on the server
  - Allow server to close HTTP session
  - Client can no longer send requests

Web Proxy Caches

- User configures browser: Web accesses via cache
- Browser sends all HTTP requests to cache
  - Object in cache: cache returns object
  - Else cache requests object from origin server, then returns object to client

No Caching Example (1)

Assumptions
- Average object size = 100,000 bits
- Avg. request rate from institution’s browser to origin servers = 15/sec
- Delay from institutional router to any origin server and back to router = 2 sec

Consequences
- Utilization on LAN = 15%
- Utilization on access link = 100%
- Total delay = Internet delay + access delay + LAN delay
  = 2 sec + minutes + milliseconds

No Caching Example (2)

Possible solution
- Increase bandwidth of access link to, say, 10 Mbps
  - Often a costly upgrade

Consequences
- Utilization on LAN = 15%
- Utilization on access link = 15%
- Total delay = Internet delay + access delay + LAN delay
  = 2 sec + msecs + msecs
With Caching Example (3)

Install cache
• Suppose hit rate is .4
Consequence
• 40% requests will be satisfied almost immediately (say 10 msec)
• 60% requests satisfied by origin server
• Utilization of access link reduced to 60%, resulting in negligible delays
• Weighted average of delays
  \[ \times .6 \times 2 \text{ sec} + .4 \times 0.01 \text{msec} < 1.3 \text{ secs} \]

HTTP Caching

• Clients often cache documents
• Challenge: update of documents
• If-Modified-Since requests to check
  • HTTP 0.9/1.0 used just date
  • HTTP 1.1 has an opaque “entity tag” (could be a file signature, etc.) as well
• When/how often should the original be checked for changes?
  • Check every time?
  • Check each session? Day? Etc?
  • Use Expires header
    • If no Expires, often use Last-Modified as estimate

Example Cache Check Request

GET / HTTP/1.1
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
If-Modified-Since: Mon, 29 Jan 2001 17:54:18 GMT
If-None-Match: "7a11f-10ed-3a75ae4a"
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
Host: www.intel-iris.net
Connection: Keep-Alive

Example Cache Check Response

HTTP/1.1 304 Not Modified
Date: Tue, 27 Mar 2001 03:50:51 GMT
Server: Apache/1.3.14 (Unix) (Red-Hat/Linux) mod_ssl/2.7.1 OpenSSL/0.9.5a DAV/1.0.2 PHP/4.0.1pl2 mod_perl/1.24
Connection: Keep-Alive
Keep-Alive: timeout=15, max=100
ETag: "7a11f-10ed-3a75ae4a"
Problems

- Well over 50% of all HTTP objects are not cacheable
  - Why?
  - Major exception?
- This problem will not go away
  - Dynamic data ➔ stock prices, scores, web cams
  - CGI scripts ➔ results based on passed parameters
- Other less obvious examples
  - SSL ➔ encrypted data is not cacheable
    - Most web clients don’t handle mixed pages well ➔ many generic objects transferred with SSL
  - Cookies ➔ results may be based on past data
    - Hit metering ➔ owner wants to measure # of hits for revenue, etc.
- What will be the end result?

Caching Proxies – Sources for Misses

- Capacity
  - How large a cache is necessary or equivalent to infinite
  - On disk vs. in memory ➔ typically on disk
- Compulsory
  - First time access to document
  - Non-cacheable documents
    - CGI-scripts
    - Personalized documents (cookies, etc)
    - Encrypted data (SSL)
- Consistency
  - Document has been updated/expired before reuse
  - Conflict
    - No such misses

Cookies: Keeping “state”

Many major Web sites use cookies.

**Four components:**
1) Cookie header line in the HTTP response message
2) Cookie header line in HTTP request message
3) Cookie file kept on user’s host and managed by user’s browser
4) Back-end database at Web site

**Example:**
- Susan accesses Internet always from the same PC
- She visits a specific e-commerce site for the first time
- When initial HTTP requests arrives at the site, the site creates a unique ID and creates an entry in a backend database for that ID
Overview

- Web
- Consistent hashing
- Peer-to-peer
- CDN
- Video

Distributing Load across Servers

- Given document XYZ, we need to choose a server to use
- Suppose we use simple hashing: modulo of a hash of the name of the document
- Number servers from 1…n
  - Place document XYZ on server (XYZ mod n)
  - What happens when a servers fails? n → n-1
    - Same if different people have different measures of n
  - Why might this be bad?

Consistent Hash: Goals

- “view” = subset of all hash buckets that are visible
- Correspond to a real server
- Desired features
  - Smoothness – little impact on hash bucket contents when buckets are added/removed
  - Spread – small set of hash buckets that may hold an object regardless of views
  - Load – all hash buckets have a similar number of objects assigned to them

Consistent Hash – Example

- Construction
  - Assign each of C hash buckets to random points on mod $2^n$ circle, where, hash key size = $n$.
  - Map object to random position on unit interval
  - Hash of object = closest bucket
  - Monotone → addition of bucket does not cause movement between existing buckets
  - Spread & Load → small set of buckets that lie near object
  - Balance → no bucket is responsible for large number of objects
Consistent Hashing

- **Main idea:**
  - map both keys and nodes to the same (metric) identifier space
  - find a "rule" how to assign keys to nodes

  **Ring is one option.**

Consistent Hashing

- The consistent hash function assigns each node and key an \( m \)-bit identifier using SHA-1 as a base hash function

  - **Node identifier:** SHA-1 hash of IP address
  - **Key identifier:** SHA-1 hash of key

Identifiers

- \( m \) bit identifier space for both keys and nodes

  - **Key identifier:** SHA-1(key)
    
    \[ \text{Key=“LetItBe” → SHA-1 → ID=60} \]

  - **Node identifier:** SHA-1(IP address)
    
    \[ \text{IP=“198.10.10.1” → SHA-1 → ID=123} \]

  - How to map key IDs to node IDs?

Consistent Hashing Example

*Rule:* A key is stored at its **successor:** node with next higher or equal ID

- IP=“198.10.10.1”
  - N123
  - K101
  - K5
  - K20
  - N32
  - K60
  - N90

- Key=“LetItBe”

- Circular 7-bit ID space
Consistent Hashing Properties

- **Load balance**: all nodes receive roughly the same number of keys
  - For \( N \) nodes and \( K \) keys, with high probability
    - Each node holds at most \((1+\varepsilon)K/N\) keys
    - Provided that \( K \) is large compared to \( N \)
  - When server is added, it receives its initial work load from “neighbors” on the ring
    - No other servers are affected
    - Similar property and a server is removed

Finer Grain Load Balance

- Redirector knows all server IDs
- It can also track approximate load (or delay)
- To balance load:
  - \( W_i = \text{Hash(URL, ip of } s_i) \) for all \( i \)
  - Sort \( W_i \) from high to low
  - Find first server with low enough load
- Benefits?
  - How should “load” be measured?

Consistent Hashing Used in Many Contexts

- Distribute load across servers in a data center
  - The redirector sits in data center
  - Finding storage cluster for an object in a CDN uses centralized knowledge
    - Why?
    - Can use consistent hashing in the cluster
  - Consistent hashing can also be used in a distributed setting
    - P2P systems can use it find files

Overview

- Web
- Consistent hashing
- Peer-to-peer
  - Motivation
  - Architectures
  - TOR
  - Skype
- CDN
- Video
Scaling Problem

- Millions of clients ⇒ server and network meltdown

P2P System

- Leverage the resources of client machines (peers)
  - Computation, storage, bandwidth

Why p2p?

- Harness lots of spare capacity
  - 1 Big Fast Server: 1Gbit/s, $10k/month++
  - 2,000 cable modems: 1Gbit/s, $ ??
  - 1M end-hosts: Uh, wow.
  - Capacity grows with the number of users!
- Build self-managing systems / Deal with huge scale
  - Same techniques attractive for both companies / servers / p2p
    - E.g., Akamai’s 14,000 nodes
    - Google’s 100,000+ nodes

Outline

- p2p file sharing techniques
  - Downloading: Whole-file vs. chunks
  - Searching
    - Centralized index (Napster, etc.)
    - Flooding (Gnutella, etc.)
    - Smarter flooding (KaZaA, …)
    - Routing (Freenet, etc.)
- Uses of p2p - what works well, what doesn’t?
  - Servers vs. arbitrary nodes
  - Hard state (backups!) vs soft-state (caches)
- Challenges
  - Fairness, freeloading, security, …
Framework

- Common Primitives:
  - **Join**: how do I begin participating?
  - **Publish**: how do I advertise my file?
  - **Search**: how do I find a file?
  - **Fetch**: how do I retrieve a file?

Searching – How?

- Needles vs. Haystacks
  - Searching for top 40, or an obscure punk track from 1981 that nobody ever heard of?
- Search expressiveness
  - Whole word? Regular expressions? File names? Attributes? Whole-text search?
    - E.g., p2p gnutella or p2p google?

What’s out there?

<table>
<thead>
<tr>
<th></th>
<th>Central</th>
<th>Flood</th>
<th>Supernode flood</th>
<th>Route</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whole File</td>
<td>Napster</td>
<td>Gnutella</td>
<td></td>
<td>Freenet</td>
</tr>
<tr>
<td>Chunk Based</td>
<td>BitTorrent</td>
<td>KaZaA (bytes, not chunks)</td>
<td>DHTs eDonkey 2000</td>
<td></td>
</tr>
</tbody>
</table>
The Solution Space

- **Centralized Database**
  - Napster
- **Query Flooding**
  - Gnutella
- **Intelligent Query Flooding**
  - KaZaA
- **Swarming**
  - BitTorrent
- **Structured Overlay Routing**
  - Distributed Hash Tables
- **Unstructured Overlay Routing**
  - Freenet

---

Napster: History

- 1999: Sean Fanning launches Napster
- Peaked at 1.5 million simultaneous users
- Jul 2001: Napster shuts down

---

Napster: Overview

- Based on Centralized Database:
  - **Join**: on startup, client contacts central server
  - **Publish**: reports list of files to central server
  - **Search**: query the server => return someone that stores the requested file
  - **Fetch**: get the file directly from peer

---

Napster: Publish

- I have X, Y, and Z!
- 123.2.21.23
- Insert(X, 123.2.21.23)
Napster: Search

Where is file A?

Query

Fetch

123.2.0.18

search(A) --> 123.2.0.18

Reply

Napster: Discussion

• Pros:
  • Simple
  • Search scope is O(1)
  • Controllable (pro or con?)
• Cons:
  • Server maintains O(N) State
  • Server does all processing
  • Single point of failure

Next Topic...

• Centralized Database
  • Napster
• Query Flooding
  • Gnutella
• Intelligent Query Flooding
  • KaZaA
• Swarming
  • BitTorrent
• Structured Overlay Routing
  • Distributed Hash Tables
• Unstructured Overlay Routing
  • Freenet

Gnutella: History

• In 2000, J. Frankel and T. Pepper from Nullsoft released Gnutella
• Soon many other clients: Bearshare, Morpheus, LimeWire, etc.
• In 2001, many protocol enhancements including “ultrapeers”
Gnutella: Overview

- Query Flooding:
  - **Join**: on startup, client contacts a few other nodes; these become its "neighbors"
  - **Publish**: no need
  - **Search**: ask neighbors, who ask their neighbors, and so on... when/if found, reply to sender.
    - TTL limits propagation!
  - **Fetch**: get the file directly from peer

Gnutella: Search

- I have file A.
- Where is file A?
- Query
- Reply
- I have file A.

Gnutella: Discussion

- Pros:
  - Fully de-centralized
  - Search cost distributed
  - Processing @ each node permits powerful search semantics
- Cons:
  - Search scope is O(N)
  - Search time is O(???)
  - Nodes leave often, network unstable
  - TTL-limited search works well for haystacks.
    - For scalability, does NOT search every node.
    - May have to re-issue query later

KaZaA: History

- In 2001, KaZaA created by Dutch company Kazaa BV
- Single network called FastTrack used by other clients as well: Morpheus, giFT, etc.
- Eventually protocol changed so other clients could no longer talk to it
- Very popular file sharing network today with >10 million users (number varies)
KaZaA: Overview

- “Smart” Query Flooding:
  - **Join**: on startup, client contacts a “supernode” ... may at some point become one itself
  - **Publish**: send list of files to supernode
  - **Search**: send query to supernode, supernodes flood query amongst themselves.
  - **Fetch**: get the file directly from peer(s); can fetch simultaneously from multiple peers

KaZaA: Network Design

‘Super Nodes’

KaZaA: File Insert

```
insert(X, 123.2.21.23)
```

I have X!

123.2.21.23

KaZaA: File Search

```
search(A) --> 123.2.22.50
```

Where is file A?

```
123.2.21.23
```

```
search(A) --> 123.2.0.18
```

```
123.2.0.18
```

```
123.2.22.50
```

```
Quer
```

```
Replies
```
KaZaA: Fetching

- More than one node may have requested file...
- How to tell?
  - Must be able to distinguish identical files
  - Not necessarily same filename
  - Same filename not necessarily same file...
- Use Hash of file
  - KaZaA uses UUHash: fast, but not secure
  - Alternatives: MD5, SHA-1
- How to fetch?
  - Get bytes [0...1000] from A, [1001...2000] from B
  - Alternative: Erasure Codes

KaZaA: Discussion

- Pros:
  - Tries to take into account node heterogeneity:
    - Bandwidth
    - Host Computational Resources
    - Host Availability (?)
  - Rumored to take into account network locality
- Cons:
  - Mechanisms easy to circumvent
  - Still no real guarantees on search scope or search time
  - Similar behavior to gnutella, but better.

Stability and Superpeers

- Why superpeers?
  - Query consolidation
    - Many connected nodes may have only a few files
    - Propagating a query to a sub-node would take more b/w than answering it yourself
  - Caching effect
    - Requires network stability
- Superpeer selection is time-based
  - How long you have been on is a good predictor of how long you will be around

BitTorrent: History

- In 2002, B. Cohen debuted BitTorrent
- Key Motivation:
  - Popularity exhibits temporal locality (Flash Crowds)
  - E.g., Slashdot effect, CNN on 9/11, new movie/game release
  - Focused on Efficient Fetching, not Searching:
    - Distribute the same file to all peers
    - Single publisher, multiple downloaders
  - Has some “real” publishers:
    - Blizzard Entertainment using it to distribute the beta of their new game
BitTorrent: Overview

- Swarming:
  - **Join**: contact centralized “tracker” server, get a list of peers.
  - **Publish**: Run a tracker server.
  - **Search**: Out-of-band. E.g., use Google to find a tracker for the file you want.
  - **Fetch**: Download chunks of the file from your peers.
    Upload chunks you have to them.
- Big differences from Napster:
  - Chunk based downloading (sound familiar? :)
  - “few large files” focus
  - Anti-freeloading mechanisms

BitTorrent: Publish/Join

BitTorrent: Fetch

BitTorrent: Sharing Strategy

- Employ “Tit-for-tat” sharing strategy
  - A is downloading from some other people
    - A will let the fastest N of those download from him
  - Be optimistic: occasionally let freeloaders download
    - Otherwise no one would ever start!
    - Also allows you to discover better peers to download from when they reciprocate
- Goal: Pareto Efficiency
  - Game Theory: “No change can make anyone better off without making others worse off”
  - Does it work? (don’t know!)
BitTorrent: Summary

- **Pros:**
  - Works reasonably well in practice
  - Gives peers incentive to share resources; avoids freeloaders
- **Cons:**
  - Pareto Efficiency relative weak condition
  - Central tracker server needed to bootstrap swarm
  - (Tracker is a design choice, not a requirement, as you know from your projects. Could easily combine with other approaches.)

Next Topic...

- **Centralized Database**
  - Napster
- **Query Flooding**
  - Gnutella
- **Intelligent Query Flooding**
  - KaZaA
- **Swarming**
  - BitTorrent
- **Structured Overlay Routing**
  - Distributed Hash Tables (DHT)
- **Unstructured Overlay Routing**
  - Freenet

Distributed Hash Tables

- Academic answer to p2p
- **Goals**
  - Guaranteed lookup success
  - Provable bounds on search time
  - Provable scalability
  - Makes some things harder
  - Fuzzy queries / full-text search / etc.
  - Read-write, not read-only
  - Hot Topic in networking since introduction in 2000-2005 time frame
- Distributed version of consistent hashing concept

DHT: Overview

- **Abstraction:** a distributed “hash-table” (DHT) data structure:
  - put(id, item);
  - item = get(id);
- **Implementation:** nodes in system form a distributed data structure
  - Can be Ring, Tree, Hypercube, Skip List, Butterfly Network, ...
**DHT Operations**

- **Join**: On startup, contact a “bootstrap” node and integrate yourself into the distributed data structure; get a node id
- **Publish**: Route publication for file id toward a close node id along the data structure
- **Search**: Route a query for file id toward a close node id. Data structure guarantees that query will meet the publication.
- **Fetch**: Two options:
  - Publication contains actual file => fetch from where query stops
  - Publication says “I have file X” => query tells you 128.2.1.3 has X, use IP routing to get X from 128.2.1.3

**DHT: Example - Chord**

- Associate to each node and file a unique id in an uni-dimensional space (a Ring)
  - E.g., pick from the range \([0...2^m]\)
  - Usually the hash of the file or IP address
- Properties:
  - Routing table size is \(O(\log N)\), where \(N\) is the total number of nodes
  - Guarantees that a file is found in \(O(\log N)\) hops

**DHT: Consistent Hashing**

- A key is stored at its successor: node with next higher ID

**DHT: Chord Basic Lookup**

- "Where is key 80?"
  - "N90 has K80"
- How do you speed up look up?
DHT: Chord “Finger Table”

- Entry $i$ in the finger table of node $n$ is the first node that succeeds or equals $n + 2^i$.
- In other words, the $i$th finger points $1/2^{n-i}$ way around the ring.

DHT: Chord Join

Assume an identifier space [0..8]

Node n1 joins

Node n2 joins

Nodes n0, n6 join
DHT: Chord Join

- Nodes: n1, n2, n0, n6
- Items: f7, f2

DHT: Chord Routing

- Upon receiving a query for item id, a node:
  - Checks whether stores the item locally
  - If not, forwards the query to the largest node in its successor table that does not exceed id

DHT: Chord Summary

- Routing table size?
  - Log N fingers
- Routing time?
  - Each hop expects to 1/2 the distance to the desired id => expect \(O(\log N)\) hops.

DHT: Discussion

- Pros:
  - Guaranteed Lookup
  - \(O(\log N)\) per node state and search scope
- Cons:
  - No one uses them? (only one file sharing app)
  - Supporting non-exact match search is hard
Freenet: History

- In 1999, I. Clarke started the Freenet project
- Basic Idea:
  - Employ Internet-like routing on the overlay network to publish and locate files
- Addition goals:
  - Provide anonymity and security
  - Make censorship difficult

Freenet: Overview

- Routed Queries:
  - **Join**: on startup, client contacts a few other nodes it knows about; gets a unique node id
  - **Publish**: route file contents toward the file id. File is stored at node with id closest to file id
  - **Search**: route query for file id toward the closest node id
  - **Fetch**: when query reaches a node containing file id, it returns the file to the sender

Freenet: Routing Tables

- **id** – file identifier (e.g., hash of file)
- **next_hop** – another node that stores the file id
- **file** – file identified by id being stored on the local node

<table>
<thead>
<tr>
<th>id</th>
<th>next_hop</th>
<th>file</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Forwarding of query for file id
  - If file id stored locally, then stop
    - Forward data back to upstream requestor
  - If not, search for the "closest" id in the table, and forward the message to the corresponding next_hop
  - If data is not found, failure is reported back
    - Requester then tries next closest match in routing table

Freenet: Routing
Freenet: Routing Properties

• “Close” file ids tend to be stored on the same node
  • Why? Publications of similar file ids route toward the same place
• Network tend to be a “small world”
  • Small number of nodes have large number of neighbors (i.e., ~ “six-degrees of separation”)
• Consequence:
  • Most queries only traverse a small number of hops to find the file

Freenet: Anonymity & Security

• Anonymity
  • Randomly modify source of packet as it traverses the network
  • Can use “mix-nets” or onion-routing
• Security & Censorship resistance
  • Data is encrypted using a key that is derived from file
  • Two types of keys:
    • Content Hash Key: Hash of the (encrypted file) – easy to check correctness of a returned file (by router or client)
    • Signed Subspace Key: documents are signed by inserter (based on public key cryptography)
      • Document can be modified by the inserter
      • Can create multiple ids

Freenet: Discussion

• Pros:
  • Intelligent routing makes queries relatively short
  • Search scope small (only nodes along search path involved); no flooding
  • Anonymity properties may give you “plausible deniability”
• Cons:
  • Still no provable guarantees!
  • Anonymity features make it hard to measure, debug

When are p2p / DHTs useful?

• Caching and “soft-state” data
  • Works well! BitTorrent, KaZaA, etc., all use peers as caches for hot data
• Finding read-only data
  • Limited flooding finds hay
  • DHTs find needles
• BUT
A Peer-to-peer Google?

• Complex intersection queries (“the” + “who”)
  • Billions of hits for each term alone
• Sophisticated ranking
  • Must compare many results before returning a subset to user
• Very, very hard for a DHT / p2p system
  • Need high inter-node bandwidth
  • (This is exactly what Google does - massive clusters)

Writable, Persistent p2p

• Do you trust your data to 100,000 monkeys?
  • May be ok for “free” song, but not for information critical to a company
    • E.g., how about DNS based on a DHT?
• Node availability hurts
  • Ex: Store 5 copies of data on different nodes
  • When someone goes away, you must replicate the data they held
  • Hard drives are “huge”, but cable modem upload bandwidth is tiny - perhaps 10 Gbytes/day
  • Takes many days to upload contents of 200GB hard drive. Very expensive leave/replication situation!