Distributed Systems

26. Distributed Caching & Some Peer-to-Peer Systems

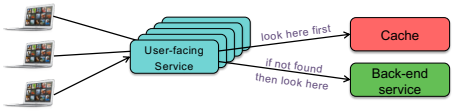
Paul Krzyzanowski
Rutgers University
Fall 2018

Caching

- **Purpose of a cache**
  - Temporary storage to increase data access speeds
  - Increase effective bandwidth by caching most frequently used data
- **Store raw data from slow devices**
  - Memory cache on CPUs
  - Buffer cache in operating system
  - Chubby file data and metadata
  - GFS master caches all metadata in memory
- **Store computed data**
  - Avoid the need to look the same thing up again
    - Results of database queries or file searches
    - Spark RDDs in memory

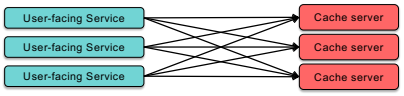
Distributed In-Memory Caching

- A network memory-based caching service
  - Shared by many – typically used by front-end services
- Stores frequently-used (key, value) data
  - Old data gets evicted
- General purpose
  - Not tied to a specific back-end service
- Not transparent (usually)
  - Because it's a general-purpose service, the programmer gets involved



Deployment Models

- Separate caching server
  - One or more computers whose sole purpose is to provide a caching service



- Or share cache memory among servers
  - Take advantage of free memory from lightly-loaded nodes

What would you use it for?

- Cache user session state on web application servers
  - No need to keep user coming back to the same computer
- Cache user preferences, shopping carts, etc.
  - Avoid repeated database lookups
- Cache rendered HTML pages
  - Avoid re-processing server-side includes, JSP/ASP/PHP code

Example: memcached

- Free & open source distributed memory caching
- Used by
  - Facebook, Wikipedia, Flickr, Twitter, YouTube, Digg, Bebo, WordPress, Craigslist, …
- Protocol
  - Binary & ASCII versions
- Client APIs for
  - command line, C/C++, C#, Go, PHP, Java, Python, Ruby, Perl, Erlang, Lua, LISP, Windows/.NET, mySQL, PostgreSQL, ColdFusion, …

memcached.org

Example: memcached

- Key-Value store
  - Cache is made up of { key, value, expiration time, flags }
  - All access is O(1)
- Client software
  - Provided with a list of *memcached* servers
  - Hashing algorithm: chooses a server based on the *key*
- Server software
  - Stores keys and values in an in-memory hash table
  - Throw out old data when necessary
    - LRU cache and time-based expiration
    - Objects expire after a minute to ensure stale data is not returned
  - Servers are unaware of each other

Memcached API

- Commands sent over TCP (UDP also available)
  - Connection may be kept open indefinitely.
- Commands
  - **Storage**
    - Storage commands take an expiration time in seconds from current time or 0 = forever (but may be deleted)
    - set – store data
    - add – store data only if the server *does not* have data for the key
    - replace – store data if the server *does* have data for the key
    - append – add data after existing data
    - prepend – add data before existing data
    - cas – check & set: *store data only if no one else updated it since I fetched it*
      (cas = unique, 64-bit value associated with the item)
  - **Retrieval**
    - get – retrieve one or more keys: returns *key, flags, bytes,* and *cas unique*

Memcached API

Commands

- **Deletion**
  - delete *key*
- **Increment/decrement**
  - Treat data as a 64-bit unsigned integer and add/subtract value
  - incr *key value* – increment *key* by *value*
  - decr *key value* – decrement *key* by *value*
- **Update expiration**
  - touch *key exptime* – Update the expiration time
- **Get Statistics**
  - stats – various options for reporting statistics
- **Flush**
  - flush_all – clear the cache

Another example: Redis

Memory cache + in-memory database + message broker

- Open source: see redis.io
- Text-based command interface
- Features
  - Key-value store
  - Transactions
  - Publish/subscribe messaging
  - Expiration of data
  - Built-in replication
  - Optional disk persistence
  - Lua scripting (via EVAL command)
  - Automatic partitioning with Redis Cluster
- Used by
  - Twitter, GitHub, Weibo, Pinterest, Snapchat, Craigslist, Digg, StackOverflow, Flickr, Shopify, Hulu, Trello, Uber. Coinbase, …

Redis Data Types

- Strings
  - Simplest type; only type supported in memcached)
- Lists
  - Collections of strings sorted by order of insertion
- Sets
  - Collections of unique, unsorted strings
- Sorted sets
  - Every element is associated with a **score** (floating point number)
  - Elements sorted by score
  - Operations to retrieve ranges (e.g., top 10, bottom 10)
- Hashes
  - Maps of fields associated with values (fields & values are strings)
- Bitmaps
  - Commands to treat strings as bits (set/clear bits)
- HyperLogLogs
  - Probabilistic data structure to estimate the cardinality of a set
    - Count # of unique items without storing the entire set of items
  - Use a fixed amount of memory

Redis as a memory cache

Timeouts & Evictions

- Set expiration for specific keys
  - Associate a timeout with a key
  - Key deleted after the timeout

```
SET mykey "hello"
EXPIRE mykey 10
```

expire key in 10 seconds

- Tell the cache to automatically evict (delete) old data
  - Methods of eviction
    - LRU (least recently used)
    - LRU only for keys that have an expiration time
    - Random
    - Random only for keys that have an expiration time

Redis as an in-memory database

- EXEC
  - Execute queued commands in a transaction
- MULTI
  - Mark the start of a transaction (operations queued until EXEC)
- DISCARD
  - Abort transaction & revert to previous values
- WATCH
  - Check-and-set behavior to ensure mutual exclusion
  - Monitor keys to detect changes
  - Abort if change takes place

Redis as a message broker

- **Publish/subscribe model**
  - Senders (publishers) do not send messages to specific receivers
  - Messages go to channels
  - Subscribers listen to one or more channels, receiving messages of interest
- **Allows for scalability and dynamic topology**
  - Publishers do not know subscribers
  - Subscribers do not know publishers
- **Support for pattern-based channels**
  - Subscribe to all channel names matching a pattern

Redis partitioning

Data can be partitioned across multiple computers

- Types
  - Range partitioning
    - Use table that maps ranges to instances
  - Hash partitioning
    - Based on hash(key): works with any key
- Who does the partitioning?
  - Client-side partitioning
  - Proxy-assisted partitioning
  - Query forwarding

Discussion
Some Peer-to-Peer Systems

Example: Gnutella

- Background
  - Created by Justin Frankel and Tom Pepper (authors of Winamp)
  - AOL acquired their company, Nullsoft in 1999
  - In 2000, accidentally released gnutella
  - AOL shut down the project but the code was released
- Big idea: create fully distributed file sharing
  - Unlike Napster, you cannot shut down gnutella

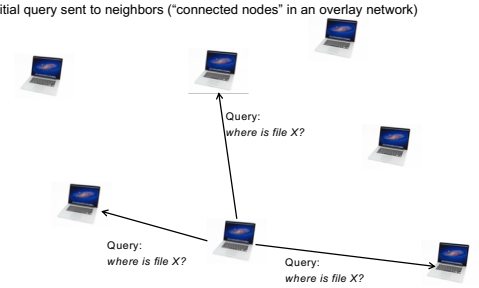
Gnutella: Overview

Gnutella is based on **query flooding**

- Join
  - On startup, a node (peer) contacts at least one node
    - Asks who its friends are
  - These become its "connected nodes"
- Publish
  - No need to publish
- Search
  - Ask connected nodes. If they don't know, they will ask their connected nodes, and so on…
  - Once/if the reply is found, it is returned to the sender
- Fetch
  - The reply identifies the peer; connect to the peer via HTTP & download

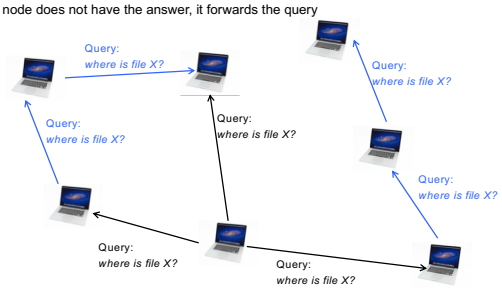
Gnutella: Search

Initial query sent to neighbors ("connected nodes" in an overlay network)



Gnutella: Search

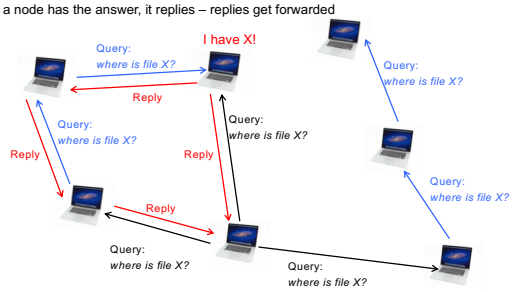
If a node does not have the answer, it forwards the query



Queries have a hop count (time to live) – so we avoid **forwarding loops**

Gnutella: Search

If a node has the answer, it replies – replies get forwarded



Gnutella: Search

Original protocol

- Anonymous: you didn't know if the request you're getting is from the originator or the forwarder
- Replies went through the same query path

Downloads

- Node connects to the server identified in the reply
- If a connection is not possible due to firewalls, the requesting node can send a *push request* for the remote client to send it the file

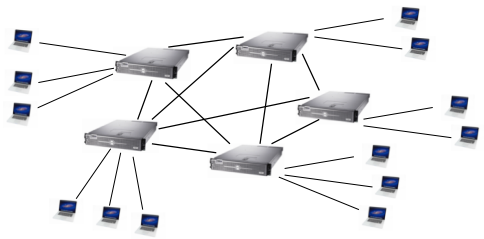
Gnutella: Summary

- Pros
  - Fully decentralized design
  - Searching is distributed
  - No control node – cannot be shut down
  - Open protocol
- Cons
  - Flooding is inefficient:
    - Searching may require contacting a lot of systems; limit hop count
  - Well-known nodes can become highly congested
  - If nodes leave the service, the system is crippled

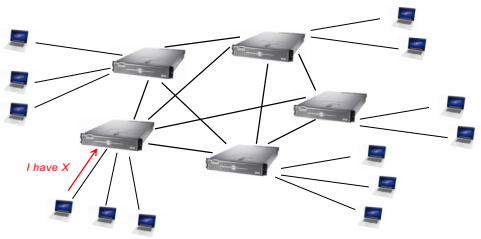
Example: FastTrack/Kazaa

- Background
  - Kazaa & FastTrack protocol created in 2001
  - Team of Estonian programmers – same team that will later create Skype
  - Post-Napster and a year after Gnutella was released
  - FastTrack: used by others (Grokster, iMesh, Morpheus)
    - Proprietary protocol; Several incompatible versions
- Big idea: Some nodes are better than others
  - A subset of client nodes have fast connectivity, lots of storage, and fast processors
  - These will be used as supernodes (similar to gnutella's ultrapeers)
  - Supernodes:
    - Serve as indexing servers for slower clients
    - Know other supernodes

Kazaa: Supernodes

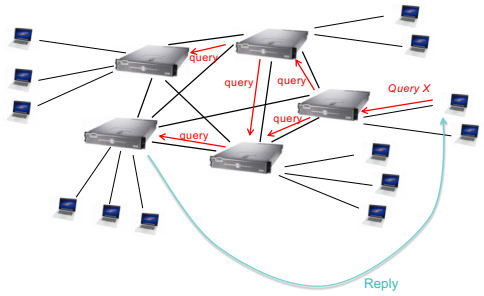


Kazaa: publish a file



Kazaa: search

Supernodes answer for all their peers (ordinary nodes)



Kazaa: Discussion

Selective flooding of queries

- Join
  - A peer contacts a supernode
- Publish
  - Peer sends a list of files to a supernode
- Search
  - Send a query to the supernode
  - Supernodes flood the query to other supernodes
- Fetch
  - Download the file from the peer with the content

Kazaa: Summary

- Pros
  - Efficient searching via supernodes
  - Flooding restricted to supernodes
- Cons
  - Can still miss files
  - Well-known supernodes provide opportunity to stop service
- Gnutella also optimized its architecture
  - Added ultranodes = supernodes

BitTorrent

- Background
  - Introduced in 2002 by Bram Cohen
  - Motivation
    - Popular content exhibits temporal locality: *flash crowds*
      - E.g., slashdot effect, CNN on 9/11, new movies, new OS releases
- Big idea: allow others to download from you while you are downloading
  - Efficient fetching, not searching
  - Single publisher, many downloaders

BitTorrent: Overview

Enable downloads from peers

- **Join**
  - No need to join
    (seed registers with tracker server; peers register when they download)
- **Publish**
  - Create a torrent file; give it to a *tracker server*
- **Search**
  - Outside the BitTorrent protocol
  - Find the tracker for the file you want, contact it to get a list of peers with files
- **Fetch**
  - Download pieces of the file from other peers
  - At the same time, other peers may request pieces from you

BitTorrent: Publishing & Fetching

To distribute a file

- Create a .torrent file
- Contains
  - name
  - Size
  - Hash of each piece
  - Address of a tracker server
- Start a seed node: initial copy of the full file
- Start the *tracker* for the file
  - Tracker manages uploading & downloading of the content

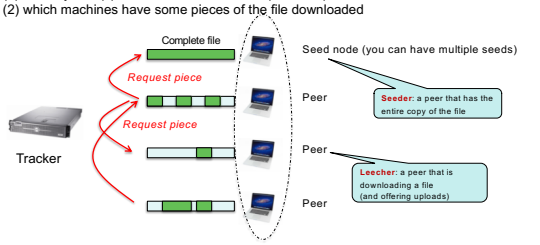
BitTorrent: Publishing & Fetching

To get a file

- Get a .torrent file
- Contact the *tracker* named in the file
  - Get the list of seeders and other nodes with portions of the file
  - Tracker will also announce you to others
- Contact a random node for a list of file piece numbers
- Request a random block of the file

BitTorrent: Downloading a file in chunks

Tracker identifies:
(1) initial system(s) that has 100% of the file (the seed)
(2) which machines have some pieces of the file downloaded



When a peer finished downloading a file, it may become a seed and remain online without downloading any content.

BitTorrent Summary

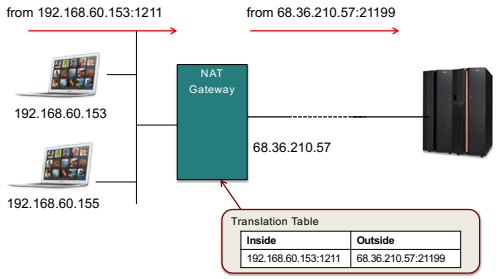
- Pros
  - Scales well; performs well when many participants
  - Gives peers an incentive to share
    - It is sometimes not possible to download without offering to upload
- Cons
  - Search is not a part of the protocol; relies on torrent index servers
  - Files need to be large for this to work well
  - Rare files do not offer distribution
  - A tracker needs to be running to bootstrap the downloads

Skype

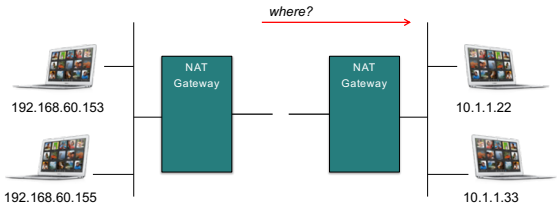
What's so hard about
User A communicating with *User B*?

Network Address Translation & Firewalls

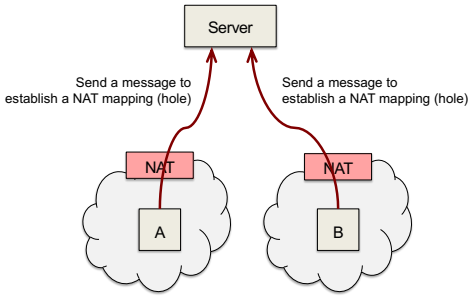
NAT: This is easy



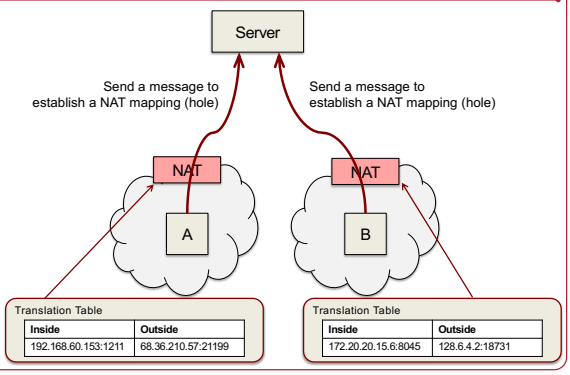
NAT: This is tricky



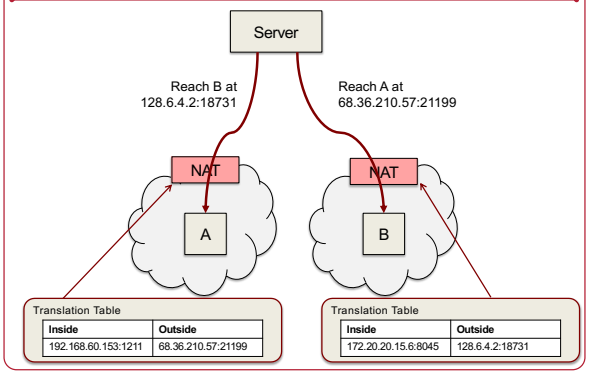
UDP hole punching



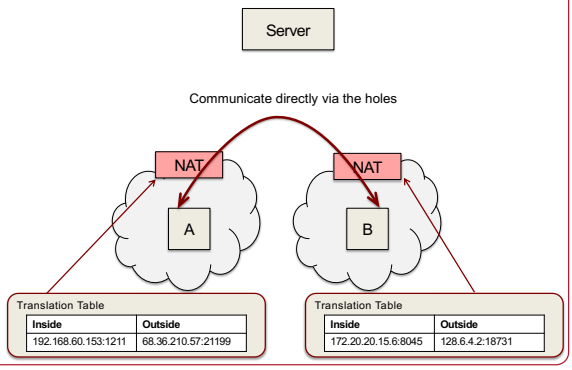
UDP hole punching



UDP hole punching



UDP hole punching



Skype

- First peer-to-peer IP-based phone - 2003
  - Developed by the people who created KaZaa
  - Niklas Zennström and Janus Frees
- Centralized component: login server
  - Manages usernames, grants access
- Otherwise fully decentralized: nodes & supernodes
  - Each client becomes an active part of the network
  - Helps locate and route traffic to other users
  - **Supernodes**: user nodes with highest bandwidth and best connectivity
    - No firewalling/NAT
    - Act as traffic hubs
    - UDP hole punching - solves NAT & firewalling problem
    - A Skype client cannot prevent itself from becoming a supernode

Skype Client

- Ports
  - Skype client opens a TCP & UDP listening port
  - Also opens TCP listening ports on ports 80 & 443
- Host cache
  - Each client builds and refreshes a table of reachable nodes
  - Contains IP address & port number of supernodes
- Buddy list
  - Stored locally – signed & encrypted – not on central server

Skype Startup

- Startup
  - Contact skype.com to see if there is a newer version
- Login
  - Authenticate user via login server (lots of them; pick one)
  - Advertises user's presence to other peers
  - Initialize client cache with info about supernodes – first use
  - Contact multiple supernodes to ensure they're alive
  - Check for presence of NAT/firewall
  - Checks for ability to communicate via UDP
    - Otherwise try direct TCP
    - Otherwise try TCP port 80 (HTTP) or port 4443 (HTTPS)
  - Login server creates session key – encrypted with server's private key

Skype User Search

- Contact supernode
  - Receives 4 nodes to query
  - If not found, then the supernode gives the client 8 nodes to query
  - Continue process until Skype gives up (unknown criteria)
- If behind a UDP-restricted firewall
  - Skype client sends a request to the supernode via TCP and the supernode does the entire search.
- Obtain user's public key signed by Skype
  - Now we can encrypt data for the other side

Skype Call Signaling & Messaging

Direct communication
If possible (no NAT/firewall problems)

Forward through a Skype node
Avoids NAT/firewall problems
Enables mixing for conferencing

By using P2P services, Skype did not have to set up & pay for directory servers and forwarding nodes

Skype Connection

- If both users on public IP addresses
  - Use a direct TCP connection
- If caller is on port-restricted NAT & callee on public address
  - Send signaling info via TCP to a Skype node, which forwards to callee
  - Node also routes UDP messages to callee and back
- If both users are on port-restricted NAT & UDP-restricted firewalls
  - Both exchange signaling info with another Skype node
  - Caller sends media over TCP to an online node, which forwards it to the callee over TCP
- Advantages of using a node as a relay
  - Allows users behind NAT & firewall to communicate
  - Users behind NAT or firewall can participate in

Special nodes

- **SkypeOut** servers
  - Skype to PSTN gateway
- **SkypeIn** servers
  - PSTN to Skype gateway

- Skype isn't really peer-to-peer anymore
  - By 2012, Skype operated ~10,000 supernodes
  - User devices would never be promoted to supernodes
  - With up to 50 million simultaneous users, a peer-to-peer environment was not efficient – there were outages
  - Mobile devices aren't suitable as P2P nodes – battery, uptime, and data volume ($) issues
  - All supernodes are now run from Microsoft data centers

The end