Peer-to-peer web objects cache

Tomasz Drwięga

20.03.2013 / Seminary

Outline

Problem Statement

Content provider

A lot of requests can cause server to become “flooded” (“swamped”)

Network admins

A lot of outgoing traffic for the same resources results in lower QoS.

Users

Requesting large files from remote servers can cause significant delays.

Problem Statement

Content provider

A lot of requests can cause server to become “flooded” (“swamped”)

Network admins

A lot of outgoing traffic for the same resources results in lower QoS.

Users

Requesting large files from remote servers can cause significant delays.

Problem Statement

Content provider

A lot of requests can cause server to become “flooded” (“swamped”)

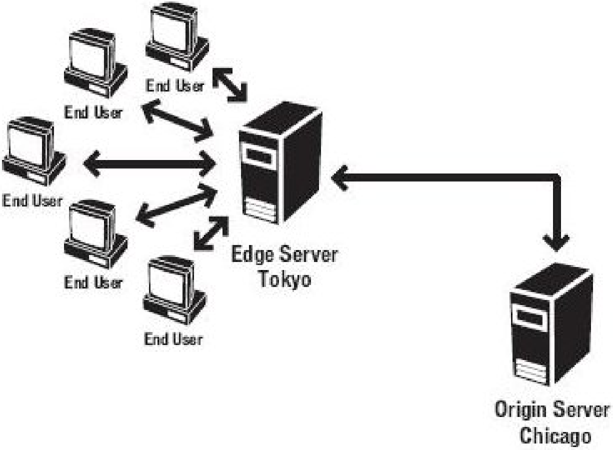
Network admins

A lot of outgoing traffic for the same resources results in lower QoS.

Users

Requesting large files from remote servers can cause significant delays.

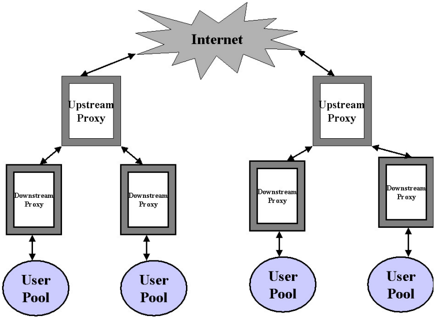
Squid object cache



A simple solution: introduce servers that will replicate original content.

Hierarchical cache

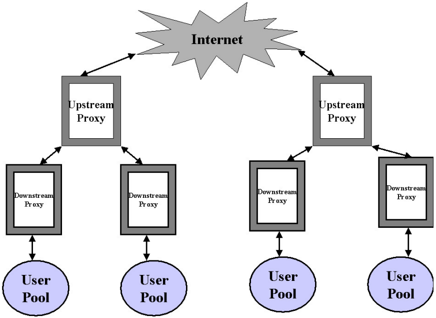
Multiple cache servers can be organized into hierarchy [2].



Leaf servers can transfer resources among themselves (cooperative caching). However it leads to excessive communication [8] [9].

Hierarchical cache

Multiple cache servers can be organized into hierarchy [2].



Leaf servers can transfer resources among themselves (cooperative caching). However it leads to excessive communication [8] [9].

Towards Consistent Hashing [4]

The main problem with multiple caching servers was to determine which server might contain the resource.

Naive Distribution

Let's assume we are searching for resource R in server S:

$$S \equiv hash(R) \bmod n$$

Serious Flaw of Naive Distribution

When new servers are added or removed the whole content has to be remapped to new targets.

Towards Consistent Hashing [4]

The main problem with multiple caching servers was to determine which server might contain the resource.

Naive Distribution

Let's assume we are searching for resource R in server S:

$$S \equiv hash(R) \bmod n$$

Serious Flaw of Naive Distribution

When new servers are added or removed the whole content has to be remapped to new targets.

Towards Consistent Hashing [4]

The main problem with multiple caching servers was to determine which server might contain the resource.

Naive Distribution

Let's assume we are searching for resource R in server S:

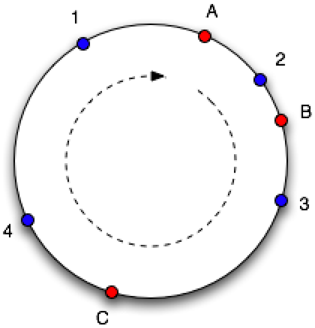
$$S \equiv hash(R) \bmod n$$

Serious Flaw of Naive Distribution

When new servers are added or removed the whole content has to be remapped to new targets.

Consistent hashing

We would like to optimize the process of adding/removing nodes so that new nodes takes their fair share of objects from others.



Consistent hashing

Each node (as well as each resource) is mapped to a point on unit circle. Node is responsible for keys after its point and its successor's [5].

Distributed Hash Table

Distributed Hash Table

Decentralized (autonomous), self-organized peer-to-peer system that provides service similar to a hash table. DHTs should also be fault tolerant and scalable.

DHTs research was originally motivated by existing systems:

- **Napster** P2P with a central index handling searches
- **Gnutella** P2P with flooding query model
- **Freenet** distributed, but no guarantee that data will be found

Four main DHTs (2001)

CAN, Chord, Pastry, Tapestry

Distributed Hash Table

Distributed Hash Table

Decentralized (autonomous), self-organized peer-to-peer system that provides service similar to a hash table. DHTs should also be fault tolerant and scalable.

DHTs research was originally motivated by existing systems:

- **Napster** P2P with a central index handling searches
- **Gnutella** P2P with flooding query model
- **Freenet** distributed, but no guarantee that data will be found

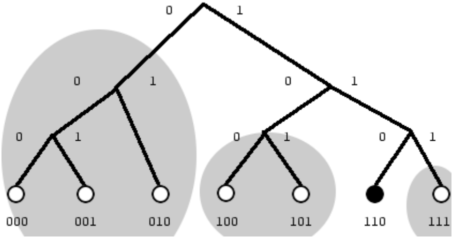
Four main DHTs (2001)

CAN, Chord, Pastry, Tapestry

Kademlia [6] (2002)

Like other DHTs, Kademlia contacts only $O(\log n)$ nodes.

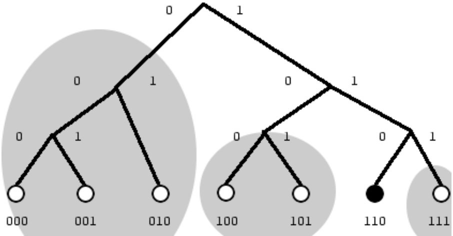
Each node has a 160 bit key. A node has a routing table that stores lists (k-buckets) of nodes whose keys share a prefix of length i with its key, but differ from it on bit $i+1$. Those k-buckets exists for each $i \in [0, 160)$



Kademlia [6] (2002)

Like other DHTs, Kademlia contacts only $O(\log n)$ nodes.

Each node has a 160 bit key. A node has a routing table that stores lists (k-buckets) of nodes whose keys share a prefix of length i with its key, but differ from it on bit $i+1$. Those k-buckets exists for each $i \in [0, 160)$



Kademlia - XOR metric

Kademlia uses XOR metric to define distance, which means that keys with long common prefix are close. It simplifies formal analysis, correctness proof and implementation.

Protocol messages:

- **PING** verifies that a node is still connected,
- **STORE** asks a node to store (key, value) pair
- **FIND_NODE** given node ID returns *k* closest nodes
- **FIND_VALUE** given object ID returns *k* closest nodes or corresponding value

Kademlia - XOR metric

Kademlia uses XOR metric to define distance, which means that keys with long common prefix are close. It simplifies formal analysis, correctness proof and implementation.

Protocol messages:

- **PING** verifies that a node is still connected,
- **STORE** asks a node to store (key, value) pair
- **FIND_NODE** given node ID returns *k* closest nodes
- **FIND_VALUE** given object ID returns *k* closest nodes or corresponding value

Kademlia routing comparison

XOR metric simplifies routing algorithm - unlike Pastry or Tapestry same algorithm is used during the whole process.

Because XOR is unidirectional (for any x and Δ there is only one y such that $d(x, y) = \Delta$) lookups for the same key converge along the same path.
So it is possible (like in design of Tapestry) to cache (key, value) pairs along the path.

This caching speeds-up retrieving popular content from the network

Kademlia routing comparison

XOR metric simplifies routing algorithm - unlike Pastry or Tapestry same algorithm is used during the whole process.

Because XOR is unidirectional (for any x and Δ there is only one y such that $d(x, y) = \Delta$) lookups for the same key converge along the same path.
So it is possible (like in design of Tapestry) to cache (key, value) pairs along the path.

This caching speeds-up retrieving popular content from the network

Kademlia routing comparison

XOR metric simplifies routing algorithm - unlike Pastry or Tapestry same algorithm is used during the whole process.

Because XOR is unidirectional (for any x and Δ there is only one y such that $d(x, y) = \Delta$) lookups for the same key converge along the same path.
So it is possible (like in design of Tapestry) to cache (key, value) pairs along the path.

This caching speeds-up retrieving popular content from the network

P2P Caching

Instead of downloading resource from original server we perform lookup in Kademlia DHT.

Potential benefits

- Large resources can be obtained faster (nodes are in the same LAN)
- WAN network bandwidth is saved

P2P Caching - Implementation

First attempt: Javascript browser plugin

An easy-to-install plugin that uses HTML5 APIs.
Why not? Requests have to be processed synchronously.

Native Client plugin for Chrome

Easy-to-install, good performance, but limited only to Chrome.
Why not? Lack of documentation, unsufficient APIs.

Fallback: Proxy [3]

Proxy server written in Python using Twisted framework and Entangled library implementing Kademlia.
Drawback: Requires additional configuration

P2P Caching - Implementation

First attempt: Javascript browser plugin

An easy-to-install plugin that uses HTML5 APIs.

Why not? Requests have to be processed synchronously.

Native Client plugin for Chrome

Easy-to-install, good performance, but limited only to Chrome.

Why not? Lack of documentation, unsufficient APIs.

Fallback: Proxy [3]

Proxy server written in Python using Twisted framework and Entangled library implementing Kademlia.

Drawback: Requires additional configuration

P2P Caching - Implementation

First attempt: Javascript browser plugin

An easy-to-install plugin that uses HTML5 APIs.
Why not? Requests have to be processed synchronously.

Native Client plugin for Chrome

Easy-to-install, good performance, but limited only to Chrome.
Why not? Lack of documentation, unsufficient APIs.

Fallback: Proxy [3]

Proxy server written in Python using Twisted framework and Entangled library implementing Kademlia.
Drawback: Requires additional configuration

Challenges

Caching logic

- To cache or not to cache? That is the question!
- Removal of old items

Requests balancing

Requests for items from the same source can be routed to completely different parts of network (random hashing keys).

Caching and Streaming of partial content

- Cache parts of content
- Stream data instead of sending whole files at once

Challenges

Caching logic

- To cache or not to cache? That is the question!
- Removal of old items

Requests balancing

Requests for items from the same source can be routed to completely different parts of network (random hashing keys).

Caching and Streaming of partial content

- Cache parts of content
- Stream data instead of sending whole files at once

Challenges

Caching logic

- To cache or not to cache? That is the question!
- Removal of old items

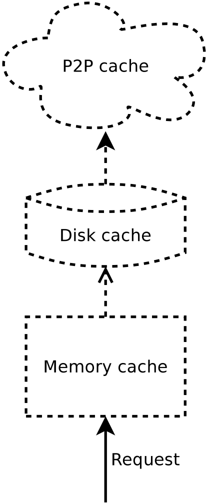
Requests balancing

Requests for items from the same source can be routed to completely different parts of network (random hashing keys).

Caching and Streaming of partial content

- Cache parts of content
- Stream data instead of sending whole files at once

Caching logic [7] - multilevel cache



Retrieving an item from disk or P2P cache increases latency.

We don't know if the item even exists in P2P cache.

Requests balancing

We could use skip graphs [1] to ask for keys in some range (for instance resources from same domain).

Multiple files from same domain could be also stored as one resource in P2P network - we could retrieve whole bunch when any resource is requested (keywords based searching).

Streaming partial content

Increasing demand on multimedia content (e.g. Youtube)

Should parts of a file be stored separately?

We need to retrieve parts in order to allow streaming of content.

References I

J. Aspnes and G. Shah.
Skip graphs.
ACM Transactions on Algorithms (TALG), 3(4):37, 2007.

A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell.
A hierarchical internet object cache.
Technical report, DTIC Document, 1995.

R. Guha and J. Wang.
Improving web access efficiency using p2p proxies.
Distributed Computing, pages 24–34, 2002.

References II

D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin.
Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web.
In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM, 1997.

D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi.
Web caching with consistent hashing.
Computer Networks, 31(11):1203–1213, 1999.

References III

P. Maymounkov and D. Mazieres.
Kademlia: A peer-to-peer information system based on the xor metric.
Peer-to-Peer Systems, pages 53–65, 2002.

R. Motwani and P. Raghavan.
Randomized algorithms.
Cambridge university press, 1995.

D. Povey, J. Harrison, et al.
A distributed internet cache.
Australian Computer Science Communications, 19:175–184, 1997.

References IV

A. Wolman, M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy.
On the scale and performance of cooperative web proxy caching.
ACM SIGOPS Operating Systems Review, 33(5):16–31, 1999.