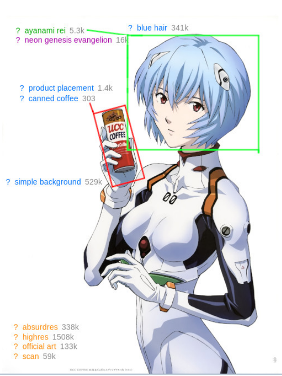
HooYa! Network: Proposal for a Distributed Booru-like P2P Image-addressable Network

Wesley Coakley
w@wesleycoakley.com

Linux User Group at N.C. State

Feb. 27 2020



Outline

Overview
- What is a Booru?
- Goals of Boorus
- Problems

HooYa! Introduction
- What is HooYa?
- Query Routing
- Tag-root (R_0) Lookups

Kademlia
- Overview
- HooYa! Extensions to Vanilla Kademlia

HooYa! Typical Operation
- Routing Example
- Joining the Network

Extra Functionality

Overview

Overview
- What is a Booru?
- Goals of Boorus
- Problems

HooYa! Introduction
- What is HooYa?
- Query Routing
- Tag-root (R_0) Lookups

Kademlia
- Overview
- HooYa! Extensions to Vanilla Kademlia

HooYa! Typical Operation
- Routing Example
- Joining the Network

Extra Functionality

What is a Booru?

Boorus are imageboard communities; they revolve around the collection, organization, and indexing of images / drawings (お絵描き (o-ekaki)) which are centrally shared and pruned. Many such booru communities exist, such as:

- Danbooru (NSFW)
  - > 3.69m+ images
  - > 108m+ tags
  - Around since May 25 2005
- Gelbooru (NSFW)
  - > 4.8m+ images
  - Around since 2007
- Konachan (NSFW)
- Derpibooru
- Safebooru

Booru Community Features

Boorus are centralized communities (i.e. running on a dedicated server); upon creating an account users may:

- Post in a community forum
- Upload images to the site
- Add tags to existing images (help organize the site)
- Flag inappropriate content
- Create and edit pages on the community wiki

Accounts on one booru (e.g. Danbooru) **do not** translate into accounts on other boorus they are separate instances (with lots of the same images)

Boorus organize Image Information

Boorus associate a set of **images** with a number of **tags** (many to many relationship). The following are examples of tags:

- `character:ayanami_rei`
- `canned_coffee`
- `simple_background`

Tags describe what is present (visually) in an image.

Example of a Booru Image

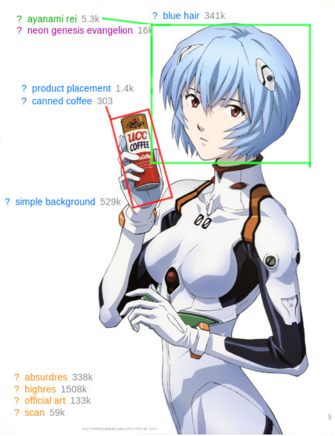


Figure: An example of Image Tagging

What are tags?

A tag t is a unique, descriptive string in the booru vocabulary V of the form:

$$t = \underbrace{\text{Character}}_{\text{Namespace}} : \underbrace{\text{Ayanami Rei}}_{\text{Attributre}}$$

Notice that $Namespace(t) = \emptyset$ is perfectly valid as in the case of $t =$ `canned_coffee` and $t =$ `simple_background`. In many booru softwares, ":" is used as the delimeter between namespace and attribute as above.

Boorus organize Image Information

The association of metadata (a mapping onto some $T \in V$) allows images to be easily searchable and discoverable.
Much like how Google indexes webpages with metadata (allowing them to be discovered with a simple query), boorus allow similar queries on image metadata.

$$\exists D : i \to V, \; i \in I$$
$$\exists Q : t \to I, \; t \in V$$

D is used to **describe** an image i using the booru vocabulary V; Q is used to find (**query**) all images i described by tag t.

Example of a booru tag

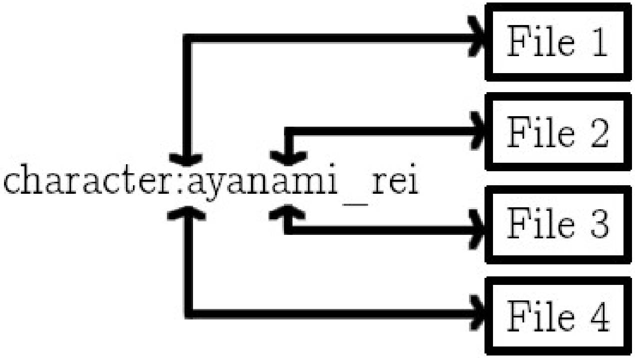


Figure: Individual tags are one-to-many relationships

Goals of Boorus

I assume that **the primary goal of boorus is to organize and index information about images** via tags; supporting this, there are a number of other goals of any Booru software:

Association
: Boorus allow users to define a set of tags associated with an image s.t. the set describes and classifies an image

Accuracy
: The set of tags must be an accurate reflection of the image in question

Consistency
: The method used to determine a tag's applicability to any given image must be the same for every other image

Community
: Allow users to collaboratively decide a tag's applicability to an image

Problems

There are a number of problems with the way things are:

- Redundant images across Gelbooru, Danbooru, etc. etc.
- Duplication of work constructing tag-sets for images
- Single point-of-failure (think DDoS, unpatched software)
- Censorship (as in the case of Danbooru)
- Limited bandwidth

Of course there are archives (e.g. Danbooru2019) of the data . . . but can we build it better?

Overview

Overview
- What is a Booru?
- Goals of Boorus
- Problems

HooYa! Introduction
- What is HooYa?
- Query Routing
- Tag-root (R_0) Lookups

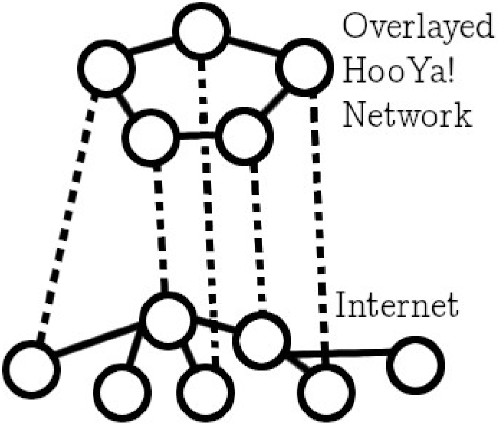
Kademlia
- Overview
- HooYa! Extensions to Vanilla Kademlia

HooYa! Typical Operation
- Routing Example
- Joining the Network

Extra Functionality

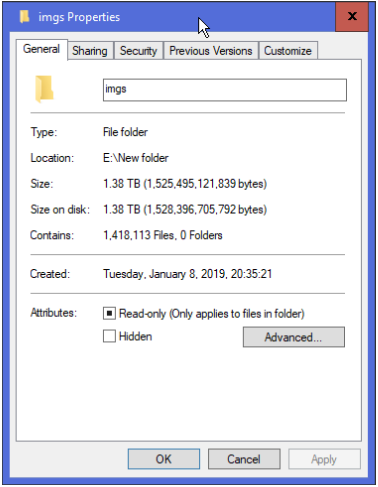
What is HooYa?

HooYa! is a protocol describing a **distributed, fault-tolerant booru** operating on top of the Internet.



What is HooYa?

Most booru users tend to save lots of those images to their (local) hard-drive; HooYa! exploits this trend.



What is HooYa?

In HooYa!, images are distributed across clients on the network; every client is a server (i.e. the HooYa!Net is P2P) which exposes its indexed files to the rest of the network.

$$I : \text{set of all images on HooYa!}$$
$$V : \text{set of all tags known to HooYa! (network vocabulary)}$$
$$C : \text{set of all clients connected to HooYa!}$$

We must modify a core function $Q : t \to I$ (Tag Query, $t \in V$) to better fit this network, however, because there is no longer a central server! The original $D : i \to V$ (Description of an image) will stay the same.

Query Routing

$$Q : t \to C$$

Queries resolve to a subset of **clients**!
Not just a subset of images matching the query.

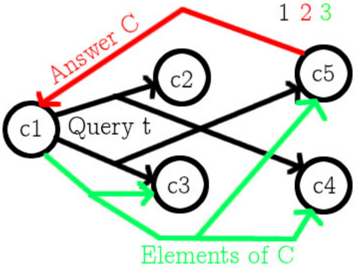


Figure: New Tag Query Function Q

Furthermore, we must have a client which knows who owns files tagged with t!

Query Routing

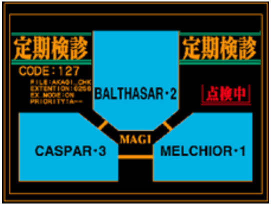
Define three roles of any node c during typical operation:

- **Root (R_0)** Knows a particular value of $Q(c)$ or $D(i)$ for some c, i
- **Server (R_1)** Exposes some files with many different $t \in V$
- **Client (R_2)** Creates requests across the network

Any client wishing to discover files (as R_2) must perform queries to nodes of the other two capacities in-order: $R_0 \rightarrow R_1$.

- Increased fault-tolerance
- Opportunity for caching along the query path!

Every client acts as R_0, R_1, R_2 at different moments.



Anatomy of a Query

But how does this work in practice? Consider the below example of a file query originating from client c_1 on tag $t =$ `Character:Akagi Ritsuko`

1. Client c_1 contacts a client c_2 who is the root for `Character:Akagi Ritsuko`; c_2 gives c_1 a list of clients c_3, c_4, c_5 who posess such files
2. c_1 queries c_3, c_4, and c_5 directly for all files and additional metadata matching t.

Tag-root (R_0) Lookups

Once the corresponding R_0 node is contacted, transactions with nodes acting in R_1 follow iteratively and without any other information needed

...but how do we know where R_0 nodes are? Especially when **all nodes are** R_0?

There are several existing methods:

- Query flooding (ask everyone!)
- Central database (what's the point?)
- Superpeers (it's a possibility, but ...)
- **Structured Search**

HooYa! uses the last one, as it reduces network traffic while keeping lookup times relatively fast.

Tag-root (R_0) Lookups using DHT

Distributed Hash Table (DHT) lookup is a structured lookup algorithm for finding values from a network-wide hash table

- BitTorrent (magnet links)
- Freenet
- IPFS
- Perfect Dark

Each node has an ID (randomly generated); nodes send queries to their neighbor(s) and their neighbor(s) forward the request until the target (in this case an R_0 node is reached. There are many implementations of DHT: Kademlia, Pastry, CAN, Tapestry (all born in 2001)!

Overview

Overview
- What is a Booru?
- Goals of Boorus
- Problems

HooYa! Introduction
- What is HooYa?
- Query Routing
- Tag-root (R_0) Lookups

Kademlia
- Overview
- HooYa! Extensions to Vanilla Kademlia

HooYa! Typical Operation
- Routing Example
- Joining the Network

Extra Functionality

Using Kademlia DHT for R_0 lookup

Kademlia is the DHT which defines the HooYa! network; it has the following characteristics:

- The distance between two nodes is the `XOR` ($\oplus$) of the node IDs of both
- Nodes have a list of contacts (other nodes it knows)
- This list (called a series of *buckets*) is used to route requests to the correct nodes
- Routing is connection-less (i.e. `UDP`)

In our case, the "keys" in this DHT are tags t and the "values" are the R_1 nodes indexed by the R_0 node.

More on Kademlia k-buckets

Any node c in a Kademlia network organizes contacts into a series of *k-buckets*, where k is a system-wide parameter.
Any contact on a Kademlia network may be stored into a given client's bucket j matching:

$$2^j \leq \mathrm{distance}(c,\, c_2) < 2^{j+1}, 0 \leq j < k$$

- Contacts in a given *k-bucket* are sorted by last-seen time
- Buckets are updated as new contacts are discovered and old ones are pruned

Stored information is **replicated** across k nodes by iteratively publishing information to the k clients nearest the key

More on Kademlia k-buckets

Typical Kademlia parameters:

1. $k = 20$ (bucket size)
2. $B = 160$ (Client ID size)
3. $\alpha = 3$ (parallelism parameter)

Primitive (non-iterative) Remote Procedure Calls (RPC):

PING Still-alive poll

STORE Store a block of data with the associated key locally

FIND_NODE Returns k nodes closest to a given ID

FIND_VALUE Returns stored data if applicable, otherwise returns a list of k nodes closest to the key

HooYa! Extensions to Vanilla Kademlia

DHT is ideal for storing one-to-one data. . .

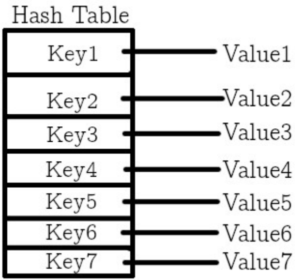


Figure: (Distributed) Hash Table lookup

But many different people can have the same file!

HooYa! Extensions to Vanilla Kademlia

Solution: l-*buckets*!

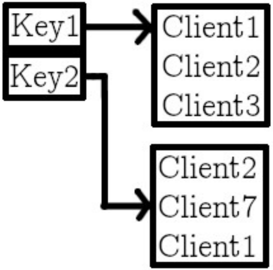


Figure: A one-to-many table

Clients are ordered by last-seen time (as in *k-buckets*); there is a maximum number of indexed clients, l, in any one l-*bucket*

Overview

Overview

What is a Booru?

Goals of Boorus

Problems

HooYa! Introduction

What is HooYa?

Query Routing

Tag-root (R_0) Lookups

Kademlia

Overview

HooYa! Extensions to Vanilla Kademlia

HooYa! Typical Operation

Routing Example

Joining the Network

Extra Functionality

Query-routing example

Assume an established network:

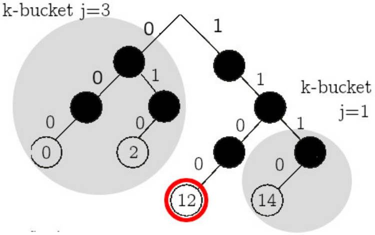
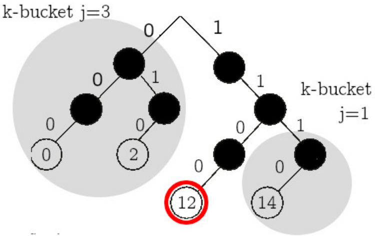


Figure: $B = 4$ Kademlia Network

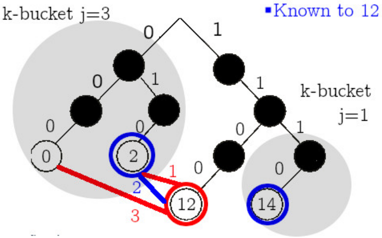
Consider client 12 ($4'b1100$); the distance to client 0 is $4'b1100 \oplus 4'b0000 = 12$.
Since $2^3 \leq 12 < 2^{3+1}$, this node may be in k-bucket $j = 3$.
Similarly, the distance to client 14 ($4'b1110$) is $4'b1100 \oplus 4'b1110 = 2$, so it may be in *k-bucket* $j = 1$ ($2^1 \leq 2 < 2^{1+1}$).

Query-routing example



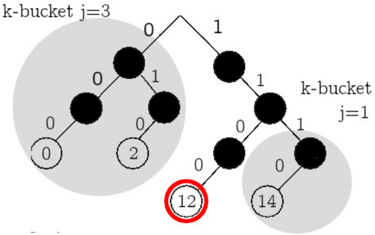
In big networks, nodes are not always aware of other nodes. Suppose we need to send a request to node 0 but we do not know its IP!

Query-routing example



Solution: issue a *FIND_NODE* request to a close neighbor (1); parse responses (2) and iterate until the node is found (3).

Query-routing example



Value lookups are conducted in a similar way; first we should hash the query and trim it to fit our B-size keyspace.

1. SHA1(`Character:Langley Asuka`) = `0xD5798B2F`...
2. For 8-bit keyspace (as in the example) we should look at nodes close to `0xD` $= 14$ for entries regarding `Character:Langley Asuka`.
3. Route the query similar to node searches (previous example)
4. If that node didn't exist, we would iteratively query nodes in that $j = 1$ *k-bucket* until we find a value (data is replicated).

Query-routing example

Node 14 is then contacted and the l-`bucket` is retrieved.

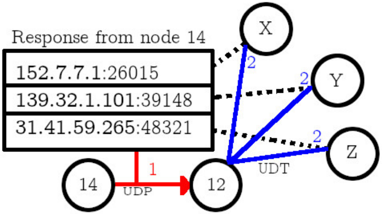


Figure: An R_0 response (1) and subsequent R_1 requests (2)

Upon receiving an l-`bucket` from node 14 corresponding to the query, node 12 is free to initiate a connection to retrieve the files and metadata from X, Y, Z; results may be cached locally to avoid excessive querying.

Notes on Data Duplication

When data is pushed to the DHT, it is **duplicated** to the k nearest nodes on the network. This data has a TTL (time-to-live), normally 24 hours, after which it must be republished by the owner.

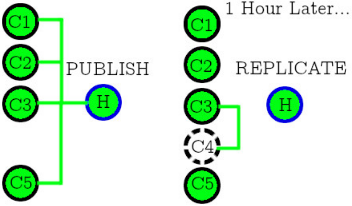


Figure: `PUBLISH-REPLICATE` Cycle

Additionally, every stored value is "replicated" periodically (traditionally every hour) to the nearest k nodes to its key.

Query-routing Protocols

What protocols do HooYa! nodes use?

UDP Connectionless, User Datagram Protocol

- R_0-level queries
- Iterative searches

UDT UDP-based data transfer

- R_1-level file transfers

Why UDP?

- UDP Hole-punching (no port-forwarding!)
- Reduce unnecessary handshaking (connectionless)
- DHT is message-oriented, so is UDP
- Sending packets? Fire and forget!

Joining the Network

Clients seeking to join the network must know a client already on the network. This is known as “bootstrapping” to the network and is accomplished by using:

1. Previously seen clients (from previous sessions)
2. DNS TXT lookups on `strap.hooya.org`
3. .txt of “preferred” boostrap nodes (last resort)
4. ~~IRC~~ (not even a last resort)

…in order of preference.

Joining the Network

Once a client node is known (using one of the methods above), insert the bootstrapping node(s) into the appropriate *k-bucket* and send a request to find its nearest neighboring nodes.

Figure: A node (black) bootstraps using a known node (pink)

Bootstrapped nodes can begin storing / querying information **immediately**.

Leaving the Network

Clients may exit the network by:

- Timeout (`PING` but no `PONG` response)
- Advertising a departure to its contacts

Figure: A node (pink) advertises a departure to another node (black)

Because **information is duplicated** across `k-buckets`, information is not lost!

Overview

Overview
- What is a Booru?
- Goals of Boorus
- Problems

HooYa! Introduction
- What is HooYa?
- Query Routing
- Tag-root (R_0) Lookups

Kademlia
- Overview
- HooYa! Extensions to Vanilla Kademlia

HooYa! Typical Operation
- Routing Example
- Joining the Network

Extra Functionality

Extra Functionality

- Embedded chatroom / community imageboard
  - PubSub? Gossip protocols?
- Synonymous tags
  - How do we decide? Active vote? Passive majority-rule?
  - And what about tag implications (`Ayanami Rei` $\Rightarrow$ `Neon Genesis Evangelion`)?
- Duplicate detection / SHA1 mismatch
  - Thumbnailing? Feature Detection?
- Tag Prediction via Cooperative, Convolutional Neural Networking