ScrapeStore: A decentralized Paxos-based system for storing and analyzing web-scraped data

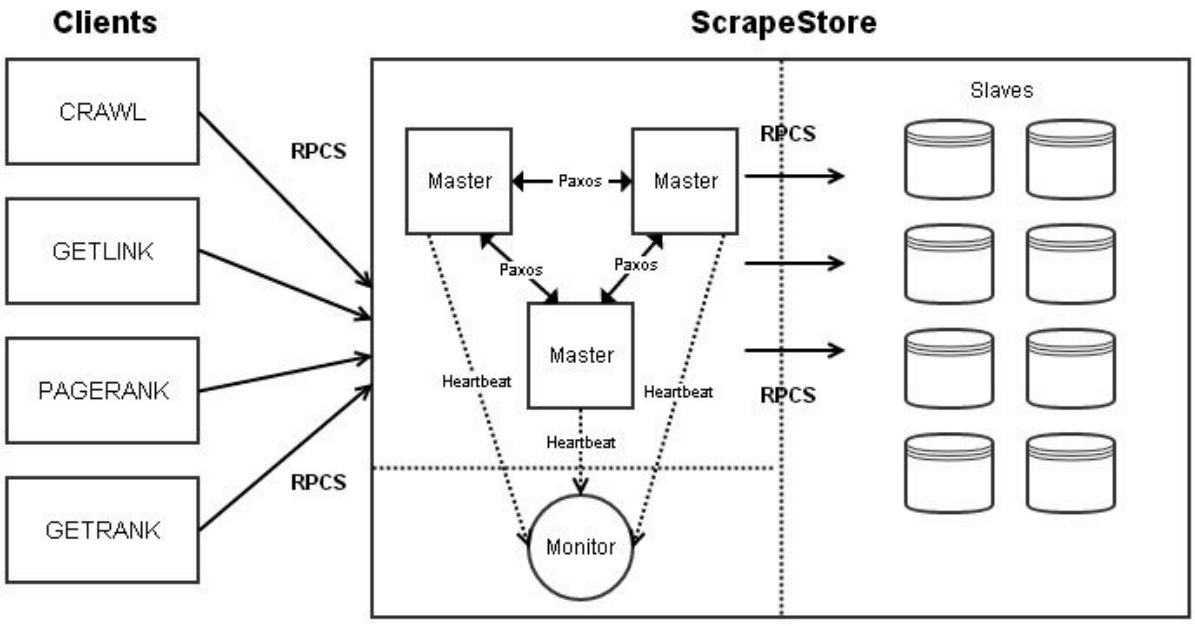
Introduction

ScrapeStore is a distributed storage system for storing web-scraped data, and analyzing the data for obtaining Page Ranks of URLs using the stored URL mappings. ScrapeStore offers high reliability and fault tolerance by using redundancy, much like the philosophy behind popular distributed storage systems like GFS and HDFS. At the heart of ScrapeStore is a collection of metadata servers, which, together, act as a replicated metadata store. The actual storage sites are managed in the form of many more slave nodes, usually much much higher in number than the master nodes. The master/paxos servers synchronize all their metadata using the Paxos algorithm. The metadata is mainly of two types: 1) mappings of which key's data is present on which collection of slaves, and 2) the amount of data stored on each slave node. The data stored on the slaves is also N-way replicated, where the N is tunable based on the requirements.

This system was built as part of the 15-640 Distributed Systems course at Carnegie Mellon University in Fall 2015 by Haibin Lin (haibinl) and Abhishek Joshi (abj1). This report delineates the design choices and implementation details of ScrapeStore.

ScrapeStore has four major components: Paxos-based Master Servers, dumb storage servers (or Slaves) which act as simple key-value stores, Clients that interact with ScrapeStore via the Master servers, and a Monitor node which monitors the health of the master servers. The next parts of the report illustrate the design of each of these 4 modules.

Architecture



1) Paxos-based Master servers

The design of the Google File System (GFS) has emerged to be a very popular design for distributed replicated storage. GFS has a GFS master node, which stores the metadata, and 1000s of Linux-based chunk servers which store the actual data. ScrapeStore is modeled on similar lines, where instead of having 1 master, we chose to have multiple masters to store the metadata. These master servers replicate the metadata amongst themselves using the Paxos based algorithm.

Metadata

The metadata is mainly of 2 types:

1. The first kind of metadata that the masters store is a mapping of URLs to a list of slave IDs which are the designated storage sites for the data associated with that URL. Once a client makes a request to a Master server to store the data for a given key, the Master server selects a list of N slaves to store that data on. The way in which this list is chosen will be discussed in the next part. Once the server selects this list, it initiates a round of Paxos to inform the other servers of the chosen mappings. Once a list of slaves is chosen, the Master proceeds to store the actual data on the slaves given the IDs in the list. Once a list of slaves is selected and agreed upon, the mappings do not change throughout the lifetime of the system.
2. The second kind of metadata that the masters store is a mapping of slave ID to the total amount of data stored in the slaves, measured in bytes. These mappings are initiated to be 0, and whenever a client makes a request to store the data, the master calculates the new sizes of data of all slaves, and initiates a fresh Paxos round to update these sizes. Here, it is crucial that the value proposed by the master is the one that is committed. This is a little unlike the pure Paxos algorithm that guarantees that *some* value is committed. Hence, our implementation has tweaked the algorithm a

little to also notify the proposer whether its proposed value was proposed, or some others value was committed. That allows the master to propose a fresh value in a new Paxos round.

Size-Aware data placement

The idea behind storing the amount of data placed on each slave redundantly across the master nodes is to allow size-aware data placement for new data. Thus, whenever data for a new key is to be stored, the master server sorts the list of slaves based on the amount of data stored by each of them, and selects **the N least-loaded slaves** to store the data on. This provides better results as compared to randomization, since it allows us to intelligently place data on slaves to provide load balancing and fairly uniform size of data on each slave node.

Append Operation

Much like GFS, ScrapeStore offers applications to store data in the form of Appends. The main idea behind providing Appends as opposed to Writes, or Puts, is to allow data versioning. Here, it is important to consider the use case for ScrapeStore, which is to store web-scraped data, which is subject to frequent changes. In such a scenario, it helps to have all previous data along with the latest data. Hence, we allow applications to *Append* their newest data for a given URL, while also keeping the older data in case it is required at a later point.

GetLinks

The Masters allow clients to obtain the scraped data for a given URL, by exporting a GetLinks RPC. The key is a URL, and the value returned is a collection of links. In conjunction with the mechanics of Append, GetLinks returns the last appended data for that URL.

GetAllLinks

The clients willing to run PageRank on entire data stored in ScrapeStore can use the GetAllLinks service provided by the Master servers. This returns a snapshot of the entire newest data stored in the slave nodes. This can be imagined as calling GetLinks in a loop for all the keys stored in the system.

PutRank and GetRank

Once a client computes the Ranks of URLs, it may choose to store the computed ranks to retrieve them at a latter point. Hence, we also provide two services: PutRank and GetRank, which store key-value mappings of the form URL->rank.

2) Slave servers

The Slave servers in ScrapeStore are synonymous to the dumb ChunkServers in GFS. Their job is simple: given a key, return the value associated with the key. It would be helpful to look at the data structures contained inside the Slave servers, after learning about the exported functions. The services provided by the slave servers are quite similar to the ones provided by the Master servers.

Append Operation

Whenever an application calls append on a master server, it translates into an Append call to the Slave node. How the slave appends this data will be explained in a short while.

Get

The clients GetLinks call gets translated into a Get call on the appropriate slave node for that key. The key is a URL, and the value returned is a collection of links. GetLinks returns the last appended data for that URL.

PutRank and GetRank

The client's PutRank and GetRank calls translate into similar calls on the appropriate slave nodes.

Data Structures

1. *valuesMap map[string] [ ] [ ] string*
   This is the main data store inside the slave servers. Here the key is a URL, and the value is a *slice of slice of strings*. For the append operation, the value is always a slice of strings. So, each slice of slice of strings in the map represents a *version* of the data. So, an append operation simply results in the slice of string being appended to the slice of slices, and a return involves returning the last *version* of the data.
2. *ranksMap map[string] float64*
   This is the data structure for storing the ranks of pages. The key is a URL, and the value is the rank of that URL as computed by the client and passed onto the slave.

3) Monitor node

We introduce the concept of a monitor node, to monitor the health of the Master servers, since they are the most important part in the whole system. Each master node knows the host and port details of the monitor node, and sends Heartbeats to the monitor after every 3 seconds. The monitor also has a list of all master nodes in the ring, and expects to hear from them after every 3 seconds. If the monitor doesn't hear from a master node in 3 epochs, it concludes that that master is dead or unreachable, and spawns a new master node with the same ID and hostport. That node is informed that it is a replacement server, and proceeds to update itself with the metadata from the other master nodes.

4) Client

In this project, we provide a simple client which interacts with ScrapeStore in two aspects: crawling web pages as well as running pagerank analytics on dataset. Each client is randomly connected with one of the master node, and multiple clients can be run in parallel.

Web Crawler

The crawler application client attempts to visit and crawl the entire internet from a given root url. A crawler client fetches the web page at root url, parses all the links contained in that web page, which form a *link relationship* (root_url, [followed_url1, followed_url2], ...). Such pair is stored in ScrapeStore for every url we crawl. For all urls appeared in the relationship, treat each one of them as the root url and

repeat the process. Based on local information, the crawler will not crawl the same url it has crawled before.

Considering the huge amount of existing and newly generated web pages, more than one clients are run together to get a more complete view of the internet. Although we tried to avoid crawling the same url repeatedly in each client, the contents which multiple clients crawl may overlap. Just like BigTable, link relationships of the same url is stored as *multiple versions* in ScrapeStore. When we retrieve the link relationships for analytical queries, the latest version is returned by default.

PageRank Analytic Engine

The client application also has the feature to run pagerank algorithm to measure the importance of the web pages we crawled. The basic idea is to rank the importance based on the link relationships on the internet, the larger number of web pages follows a certain web page, this web page is viewed as "more important".

The client runs the simplest form of pagerank algorithm. We use PR(pi) to denote the pagerank of a web page pi. M(pi) denotes the set of all web pages that points to web page pi. N is the total number of web pages in the dataset. d is the damping factor, which indicate how likely a user will follow the link contained in the website. L(pi) is the number of urls which pi follows.

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

In our implementation, we consider all links appeared in our dataset, including the links which other pages point to but not crawled yet. For these yet-to-crawl pages, our implementation distributes their rank evenly to all other pages, since the number of pages they follow is not available yet. The damping factor by default is set to 0.85, which is the recommended configuration. Since the algorithm is an iterative process, we'll keep running the algorithm until all the ranks converges with some negligible error bound.

Crawl Command

The client application can specify the root url to start crawling with and the number of pages to crawl. The crawl function uses a http client to retrieve web pages of target urls, parses and extracts all the urls contained in the web page to form a link relationship unit. The crawled result is then stored in ScrapeStore for further use.

GetLink Command

The client application provides a command to retrieve all the urls contained in a specified root url. It basically queries the server to get the latest version of link relationships of the root url(if the root url exists in ScrapeStore).

PageRank Command

The pagerank command will run the described pagerank algorithm on the entire dataset. This is a very data intensive operation, since the client has to gather all the link relationships stored in dumb slaves to

compute the pagerank of each page. The result rank of each page is then stored into ScrapeStore at the end of this operation.

GetRank Command

The client is able to retrieve the rank calculated previously by querying the ScrapeStore. It will either return the pagerank of the target url if the record is found, or throw an error that the rank is not calculated yet.

User Manual

Environment setup

1. Set environment variable `GOPATH`
2. use `go get golang.org/x/net/html` to install the html package provided by golang
3. Make sure the `cmu440-F15` folder is under `$GOPATH/github.com/cmu440-F15/`

Launch the application

1. Before starting the application, we need to start the master, slave and monitor node by `$GOPATH/github.com/cmu440-F15/paxosapp/scripts/start_server.sh`
   This starts a ScrapeStore cluster of 6 slave nodes, 3 master nodes and 1 monitor node. You need them running so that application client can connect them with.
2. Run the client application with `$GOPATH/github.com/cmu440-F15/paxosapp/scripts/run_client.sh`
   The script comes with detailed usage of the commands for crawler and pagerank. The client will exit as soon as it finishes it job.
3. Try to launch several client to do `crawl` command in parallel with different starting root urls. There should be no problem for them to store data to ScrapeStore concurrently.
4. Verify if the crawl data is successfully stored in ScrapeStore by `getlink` command.
5. Run `pagerank` command to calculate the rank over all crawl data collected
6. Verify the rank data by `getrank` command with a url you're interested in
7. You can also kill one of the master server by `kill 12345`(replace 12345 with any master's pid) and see if a new one is launched to replace this one later
8. Stop the ScrapeStore cluster with `$GOPATH/github.com/cmu440-F15/paxosapp/scripts/stop_server.sh`

Acknowledgement

1. pagerank - a pagerank algorithm skeleton implemented in go. Our client application implements the pagerank analytic feature based on this.
   https://github.com/dcadenas/pagerank
2. collectlink - a html parser which collects all the urls in a web page. This is quite helpful for our crawler implementation
   https://github.com/JackDanger/collectlinks