Distributed Web Scraping - Phase 1

Group: Michail Roesli (V00853253), Quinn Gieseke (V00884671), Blake Smith (V00850827)

Introduction

In this report, we will discuss how our distributed web scraping project implementation is progressing and provide some explanation towards why we made certain decisions in our project. In the first section of this report, we'll discuss the design of the project and explore Raft, the Linux virtual machines, the android environment, the web crawling functions that we're exploring, and how we connect these aspects together with remote procedure calls (RPCs). Next, we'll discuss our implementation for each respective section, and showcase some features of the project. Lastly, a small discussion on our future plan will then be discussed. A link to the project code is provided at the end of this report.

Design

In this section we'll discuss why we chose certain tools, libraries and languages for each aspect of the project. Diagrams will be included in this section as well.

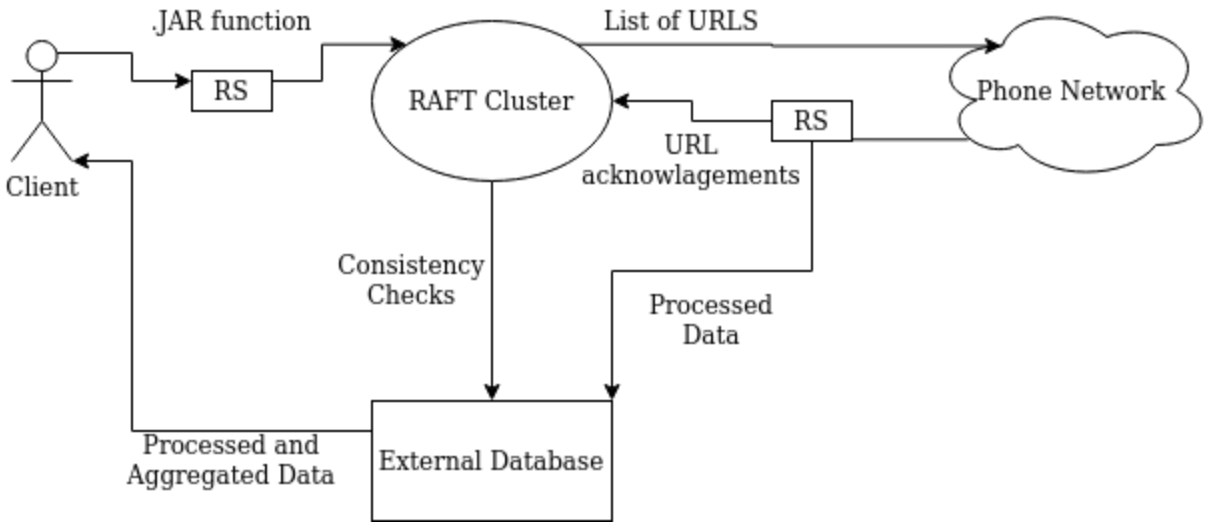
Data Flow

The project breaks down into 3 main components

1. A reliable cluster of dedicated machines running a distributed consensus protocol
2. Mobile phones which will connect and disconnect to the system at will
3. A routing service which allows new mobile clients to locate the machines which are in the reliable cluster

The work breaks into the following

1. Crawl the web from a start url and collect links on each page. Produce a stream of urls to be processed
2. Distribute the urls from that stream between different nodes, ensuring a low rate of replication (processing the same url more than once).
3. Map the HTML of the webpage at each given url to JSON data (via a user provided function)
4. Store the JSON data in some well-indexed distributed database



This diagram shows the network in its fully distributed state. For the initial tests, the RAFT cluster is replaced with a single locally running process, and the phone network is replaced with a single instance of an Android VM running within Android Studio. The database is replaced with a local file for ease of testing, and the routing service is reduced to a simple response to localhost.

Raft

We are using the Raft algorithm to ensure that a Master node is reliably available. The system is based on having a reliable network for mobile clients to join and exit as they please. The Raft Leader will be responsible for distributing work among the follower nodes, as well as the mobile clients.

We are using Golang for the Raft implementation, as well as the distribution of work due to it's great support for concurrency. It also fits best for the project as we can apply our experience in the Labs implementing Raft.

Linux VM

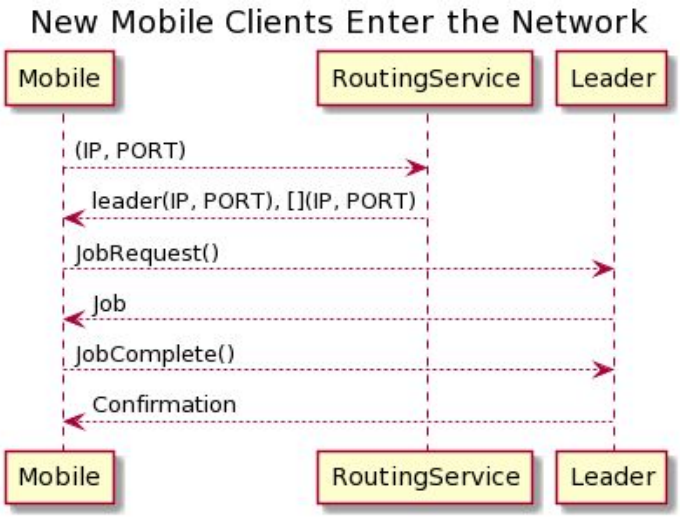
The next component is a Linux (command line) program which will emulate the mobile clients. Unfortunately it is difficult to simulate many Android devices as an Android emulator uses almost 2GB of RAM. To mitigate this, and allow us to test many devices entering and leaving the system at will, we created a linux client which emulates an Android client.

The Linux client is written in Kotlin as most of its code is shared with the Android project. All of the important logic is shared between the two projects and only specific platform code is written differently. For instance Android uses a different library for making HTTP requests, which has more complexity and only works on the Android operating system. In contrast the Linux client uses the standard Java API's for this purpose.

We plan to test the scalability of the system by running the linux client in many docker containers and having the clients purposefully drop out of the network and come back into it in random intervals.

We will also test the Android app on our own devices and several VM's to verify that it functions properly. Since all of the important code is shared it should not be necessary to test on many android devices simultaneously, as that will already be considered in testing with many linux clients

Android



The above is a sequence diagram describing how the mobile clients will communicate with the Raft cluster. When a user chooses to start a web scraping job the android/linux client app will make a request to the (Python & Flask) routing service. It will then communicate with the Raft leader directly via gRPC calls.

Jobs are proved to the mobile client on a first-come first serve basis. Once a job is complete the android/linux client will inform the Raft Leader via a call to the JobComplete() RPC. Once the client is done working (user indicated a desire to stop scraping) it will simply not make another call to the JobReqest() RPC.

The Android/Linux client app is set up such that the User of the service (eg. a company with a large web scraping workflow) only needs to implement a single scraping function (provided in a Jar library with it's dependencies). The function will take a Jsoup Document object as input and produce a JSON string. The system will handle calling the function with appropriate URLS and communicating the JSON data to the distributed database.

RPC

The main infrastructure we use for passing RPC calls is with gRPC. This tool is widely used since it provides a modern open source and high performance RPC framework that can be run in any environment. Another reason why we chose to use gRPC is because of its ability to work with Go which was our choice of language for the Raft cluster. With gRPC we are able to automatically generate some Client, and job handlers.

Functions

Our first web crawling function we decided to implement is the word count feature. This is because it is one of the most simple. However, for future implementations, this function will be provided by the client, which will be compatible with the supplied API, and will be able to execute arbitrary code on the data gathered, and be able to return its arbitrary results to the client.

Implementation

In this section we'll discuss how each part of the project is being implemented, deployed, tested, any problems we've had, and changes that are being.

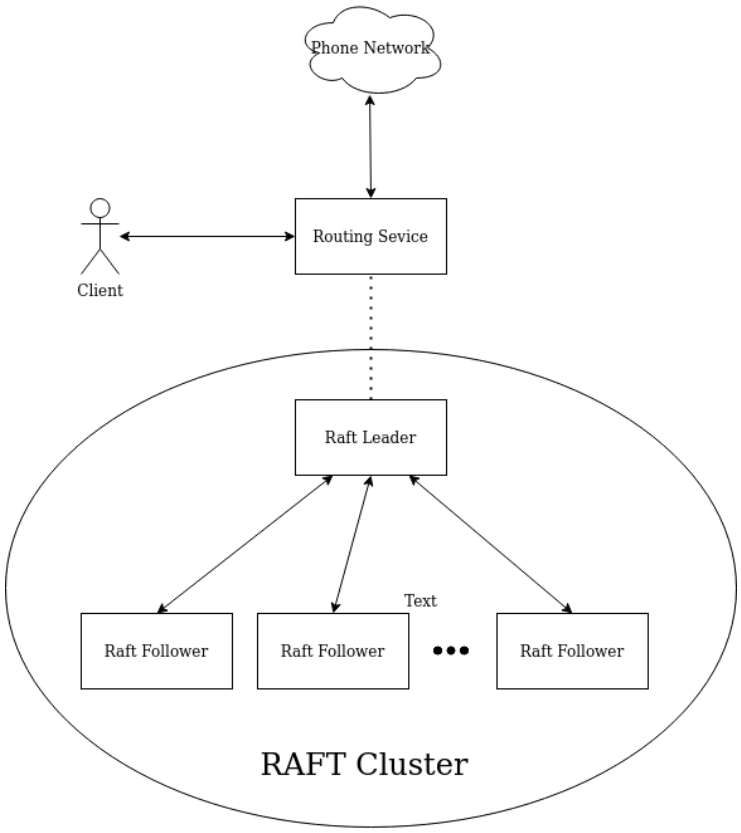
Docker

To be able to test our implementation consistently and on any system we are making use of Docker containers. This is an open source containerizer that allows us to package our program into standardized units for development, and eventual deployment. This also reduces the amount of work needed to set up our python, go and kotlin environments and start using our program from anywhere. Eventually, we plan on adding this to some continuous integration platform such as CircleCI, TravisCI or Jenkins. This allows us to reduce the effects of software evolution and find problems in our code earlier.

Raft

Our RAFT cluster has a few key differences from the lean paper description provided in the course. The paper simply assumes that some communication channel exists that all nodes can access, and that outside actors (in our case, the client and the phone network) can freely communicate with the leader. In our implementation, since we are assuming that RAFT nodes might be distributed among different computers, this implementation is a little more difficult than simply sharing a channel in GO. Since we have to deal with the larger WAN network, we run

into issues with IP addresses being dynamically allocated when nodes might be reset, NATs that might obfuscate



RAFT nodes as well as phones and a whole host of other real world issues. As shown in the diagram, we have avoided a majority of these problems by adding one layer of abstraction between the RAFT cluster and all external communication, namely an extra Routing Service. This is a service that can be started by any RAFT leader that sits at a single, static IP that is open to the larger network, meaning that all connection requests can access it, and then pass through to the RAFT network as a whole, with updated local IP values changed as needed. Since all external services, including the phone network and the client (or clients), now only have to remember this one, static IP, all issues regarding RAFT node unreliability from a network perspective, have been fixed. Additionally, for all network needs, we are utilizing flask to setup a simple routing service to direct android clients to the current cluster leader.

Functions

One function that we have implemented is a word count. We were able to successfully run this and produce the result as seen in Appendix A. This was done by parsing the HTML tags using Jsoup. Other functions we think would be useful to implement include word relation maps, unique sentences, tweet maps, geo-tagged data for clustering and various other machine learning applications to gather pictures or other data types.

Future Work

In the next phase we plan on completing our raft cluster communication, and completion of the client provided arbitrary functions. This will coincide with our completion of the raft in labs which will help us with working towards having our raft cluster to withstand failures with android devices coming online and offline at will. Lastly we'll explore plugins for functions so that we can run and send results to the distributed database dynamically.

Project Code Link

https://github.com/BlakeASmith/DistributedWebScraping

Appendix A: Word Count example

{"reddit:":140,"the":20059,"front":163,"page":140,"of":10774,"internet":163,"Press":3388,"J":1694,"to":17241,"jump":1694,"feed.":1694,"question":1718,"mark":1694,"learn":2167,"rest":1718,"keyboard":1694,"shortcuts":1694,"Log":2444,"insign":2064,"up":4055,"User":2918,"account":2449,"menu":2310,"Create":560,"a":11856,"postDrafts0":525,"Post":1596,"Image":630,"\u0026":1260,"Video":630,"Link":1807,"Poll":630,"0/300":665,"Bold":805,"Italics":805,"Strikethrough":805,"Inline":805,"Code":1610,"Superscript":805,"Spoiler":1505,"Heading":805,"Bulleted":805,"List":1610,"Numbered":805,"Quote":805,"Block":1610,"Table":805,"Add":1610,"an":1053,"image":878,"video":805,"Switch":805,"markdownMarkdown":735,"mode":770,"This":2306,"community":2383,"does":1941,"not":2925,"allow":758,"original":1330,"content":1890,"tag":735,"OC":700,"Mark":1470,"as":2707,"spoiler":735,"Not":782,"Safe":735,"For":1413,"Work":735,"NSFW":701,"Select":735,"subreddit":1722,"enable":735,"flair":1107,"Flair":735,"Cancel":746,"Send":770,"me":2461,"post":1461,"reply":770,"notifications":770,"Posting":560,"Reddit":7459,"Remember":1118,"human":664,"Behave":595,"like":2089,"you":3501,"would":1348,"in":8063,"real":687,"life":618,"Look":619,"for":4213,"source":595,"Search":595,"duplicates":595,"before":641,"posting":595,"Read":595,"community's":595,"rules":645,"Please":1145,"be":5888,"mindful":525,"reddit\u0027s":525,"policy":3675,"and":8620,"practice":525,"good":617,"reddiquette.":525,"helpReddit":2653,"AppReddit":2653,"coinsReddit":2653,"premiumReddit":2653,"gifts":2807,"aboutcareerspressadvertiseblogTermsContent":2653,"policyPrivacy":2653,"policyMod":2653,"Inc":2499,"©":2499,"2020.":2499,"All":2522,"rights":2871,"reserved":2499,"":42268,"sign":822,"Drafts0":35,"0":2308,"markdown":70,"Markdown":35,"reddiquette":35,"help":178,"App":154,"coins":154,"premium":154,"about":1452,"careers":154,"press":154,"advertise":177,"blog":154,"Terms":154,"Content":154,"Privacy":154,"Mod":154}