UNIVERSITY OF VICTORIA

Department of Engineering & Computer Science
CSC 462 – Distributed Systems

Distributed Web Scraping

Aug 15, 2020

Blake A Smith - V00850827 - blakeas@uvic.ca
Quinn Gieseke - V00884671 - qgieseke@uvic.ca
Michail Roesli - V00853253 - mroesli@uvic.ca

https://github.com/BlakeASmith/DistributedWebScraping

Table of Contents

List of Figures

Glossary

Assigned job — A **Job** is considered **assigned** once it has been sent to the "*jobs*" Kafka topic to be processed

Client — A process which receives jobs from the *"jobs"* Kafka topic and sends the retrieved data to the *output topics*

Completed job — A **Job** is considered **completed** once it has been sent to the "*completed*" Kafka topic and all results have been sent to the *output topic* (a topic with the same name as the **service**)

Job — A **Job** is a set of URLs to be processed, along with a list of which plugins to apply and the name of the associated **service**

```
data class Job(
     val Id: Int,
     val Urls: List<String>,
     // names of plugins to use for this job
     val Plugins: List<String>,
     // name of the service which this job
     // is a part of
     val Service: String
)
```

Plugin — A **Plugin** defines what operation to perform on each discovered webpage. Users implement the *Plugin* interface and produce a JAR file containing their implementation and any required dependencies

```
interface Plugin {
   fun scrape(doc: Document): String
}
```

Those JAR files are stored in a Kafka topic as a *ByteArray* and loaded dynamically at runtime

Producer A process which discovers new URLs, groups them into jobs, and sends them to the "*jobs*" Kafka topic

Service A **Service** is a **set of URLs** to be crawled along with **rules** for crawling those URLs

```
data class Service(
      val name: String,
      val rootDomains: List<String>,
      val filters: List<String>,
      val plugins: List<String>
)
```

Each **Service** has an associated topic of the same name which the results (as **JSON** strings) can be retrieved. These topics will be referred to as **Output Topics**

User An individual or organization which has produced a **Plugin** and a **Service** definition using our API

Abstract

We set out to create a framework providing distributed web scraping as a service, able to harness the latent network and computation ability of idle computers. A user should only have to, at minimum, define the data to be created, in the form of a plugin, and the websites they wish to scrape. Since the limiting factor in most web scraping tasks is bandwidth, we wished to create a service that would be highly scalable across many separate low-powered devices, allowing any of our users access to enterprise grade bandwidth.

In order to meet our goals, we used a combination of Kafka and etcd to provide reliable and distributed data and key-value storage respectively. Since each of our web scrapers and crawlers communicate both upstream and downstream to these industry standard tools, we can guarantee that the number of nodes can be highly scalable in order to match demand, while allowing any device with an internet connection to participate in the cluster.

Upon testing our initial implementation, we have shown that our framework has succeeded in becoming highly scalable across multiple tested machines, including Google Cloud and each of our group members personal computers. Further work revolves mostly around user experience, quality of life, and optimization changes.

Introduction

Data science is important for both small and large companies who seek growth. This is especially true now that more companies are having to move their infrastructure from in person to online services to online with the effects of the Covid-19 pandemic. For companies to grow, they need to make use of data analysis techniques which extract correct and actionable data. Unfortunately, some companies do not have the funds necessary to buy or gather enough data for their businesses. In exploring this, we identified that many organizations have computing resources which go unused for much of the day, especially overnight.

For our project we decided to create a solution which could utilize phones, tablets, laptops, and other hardware, during those times it goes unused, to gather and scrape data from websites. The idea is to enable organizations to better utilize their existing hardware, and to enable individuals to assist organizations they support by volunteering unused hardware resources. Thus providing a greater capacity to gather data to businesses, non-profits, and research organizations without large up-front hardware costs.

Project Goals

The purpose of this project is to provide a platform for distributed web scraping as a service which is reliable and highly available, both for organizations to deploy on their own hardware, and as a proof of concept for a standalone cloud hosted web scraping service. An important goal is for the number of nodes operating in the cluster to scale without any external intervention. Additionally, both Client nodes and Producer nodes should be able to exit and enter the cluster at any time, possibly only contributing for a few minutes. The Client program will also be available as an Android app, which means that network connections will be unreliable for many clients. The system must be adaptable and ensure **at least once** semantics, regardless of frequent network failures. On the same token, nodes should be able to recover and rejoin the cluster once they are re-connected. Thus we aim to provide a fault-tolerant system that is flexible enough for custom data gathering tasks.

In this project we set several technical goals and requirements:

- Ensure At-least-once semantics for scraping
- Run multiple independent scraping services
- Start new services and add new functionality at runtime
- Resilient to node failures
- Near-linear scaling of pages processed when running clients on separate networks
- Allow for Android devices to run clients

To meet these goals we require a better understanding of the related work, available tools and technology available to accomplish this project.

Literature Review

In this section we'll discuss related work, followed by some important background knowledge researched for making design decisions in the project including inter-process communication, and key-value storage.

Related Work

The initial idea for the project came from reading the papers on design and Implementation of a High-Performance Distributed Web Crawler [1,2]. The paper describes the problem a web crawler may encounter when interacting with millions of hosts which poses issues of robustness, flexibility, and manageability. This gave us the idea of providing a solution similar to that of Scrapinghub but to make our idea unique we looked for some inspiration from the Folding@home project [3]. Their purpose is to provide people with ways to contribute towards disease research by providing their devices for solving problems which require many computer calculations. This allows users to provide an alternative to monetary funding for cures. Overall, it is a distributed computing project for simulating protein dynamics which helps scientists to better understand biology and provide new opportunities for developing therapeutics. For our project, we thought that this was a very interesting idea and that it could be useful in other areas including tech and the ever growing industry of data science. It has long been regarded that accessibility to information drives business in the modern age [4], and so the opportunity seems ripe for web-scraping as a service, allowing for smaller businesses to access the vast amount of information available on the internet.

Inter-Process Communication

Throughout the development process we gained new insights into the problem space and discovered that our initial design did not best enable us to meet our goals. As such, our design has evolved and much has changed throughout the process. Here we will briefly discuss the initial design, as well as the technical reasons it was necessary to change it.

The initial design for this project used **Cassandra** as a storage solution, with the intent that users could direct the system to a Cassandra cluster running in the cloud and their results would be written to that cluster. We chose Cassandra because of it's excellent write scalability and flexibility as a noSQL database [5]. However we soon realized that requiring a Cassandra cluster to be deployed in order for a user to access their results was too large a barrier of entry. Additionally it did not allow for an alternative database solution, or the ability to perform any

further operations on the collected data before storing it. For these reasons we decided to use **Kafka** to store the user's data temporarily and allow them to consume and store the data as they please [6].

The switch to Kafka as a means for users to access their data quickly led to Kafka being the central point of communication for all components of the system. We initially used **gRPC** to communicate between components, which required a routing service (implemented in python using **Flask**) to be available at a static IP address. Due to the dynamic sizing of the cluster and nature of the hardware we are designing for, the IP addresses of the nodes in the cluster are likely to change frequently as mobile devices transition between access points. Thus there must be some component of the system which can perform address resolution as new nodes enter the system. We quickly saw that the Kafka cluster could facilitate communication between the components of our system and eliminate the need for components to communicate with each other directly. Having clients and producers communicate via Kafka topics greatly simplified the process of including additional clients and producers dynamically.

Overall we found that the web scraping tasks lend themselves well to stream processing which is the main design philosophy of Kafka. We also chose to use Kafka because it provided more flexibility for the cluster owner to be able to choose where they would like to set up their cluster alongside their preferred database.

Key-value Storage

Initially we explored golang packages which performed distributed hash tables (DHT) for storing the urls that we have traversed. DHTs are distributed systems which provide the capability to perform lookups like a hashtable; the user can store or retrieve a value based on a key within a distributed environment. We found packages such as Stovepipe's DHT, [7] and niktuku's [8] DHT implementations which provided this functionality. Although the functions were well documented, the implementation was not as easy without existing examples to use. We then explored other frameworks which are often used to communicate information between clusters such as Zookeeper, Consul and etcd.

Out of the three the oldest is **Zookeeper** which was "originally created to coordinate configuration data and metadata across Apache Hadoop clusters" [9]. Like **etcd** and **Consul** it requires a quorum of nodes to operate which are "strongly consistent" [10]. They also provide client libraries to build more complex distributed systems. Unlike Zookeeper, etcd and Consul use the raft implementation for maintaining consistency and come with a built in key-value store that "supports HTTP/JSON application programming interfaces (APIs)" [9]. Consul builds on top of this and provides "native support for multiple datacenters as well as a more feature-rich gossip system that links server nodes and clients" [10].

Unfortunately due to the way that Zookeeper manages nodes with key-value entries, it provides inherent scalability issues and adds client-side complexity. This led us to choose

between Consul and etcd and out of the two etcd provided us with all the necessary functionality of being fully replicated, highly available, reliable, consistent, secure, and simple.

Implementation

Distributed web scraping as a problem can be broken down into a few distinct steps;

1. Discover pages which have not yet been seen (crawling/job production)
2. Distribute those pages amongst a set of processes (job distribution)
3. Perform some operation on those pages (scraping/job processing)

An overview of the system is shown in figure 1 below.

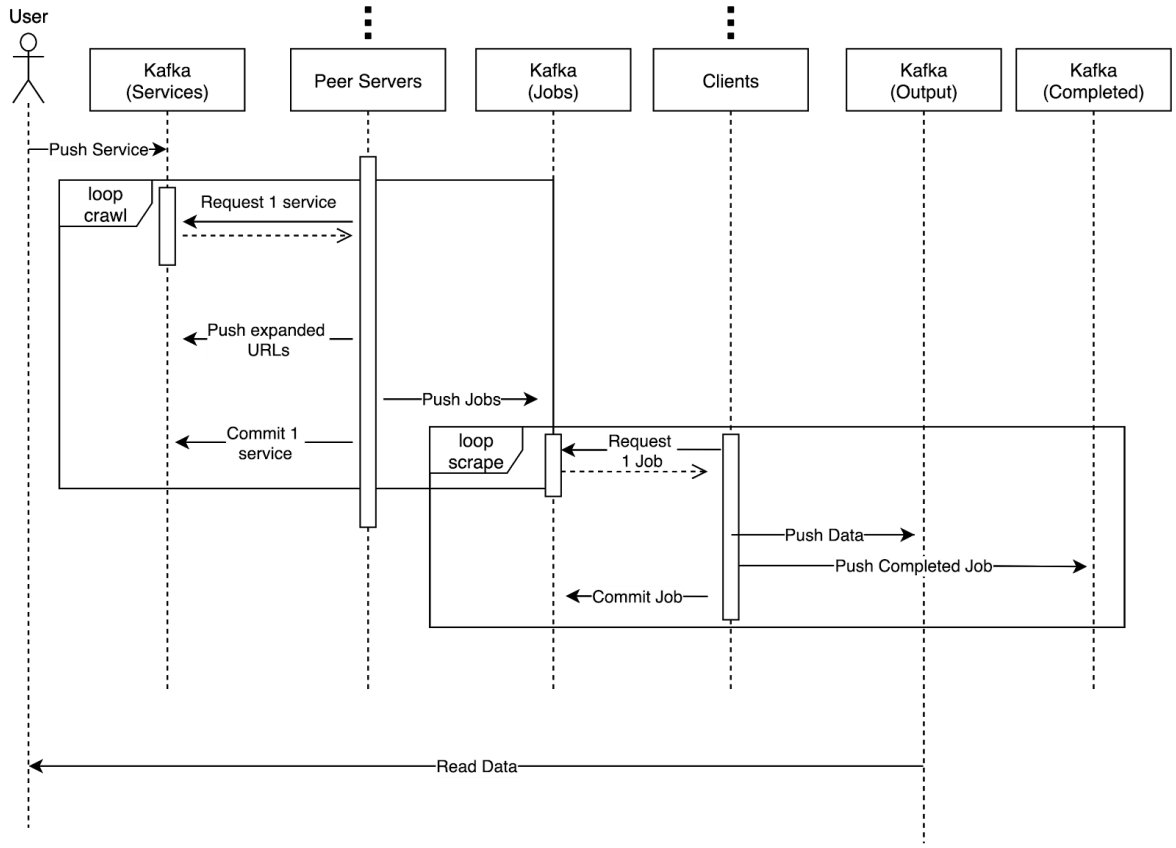

Figure 1. System Communication Sequence Diagram

Crawling/Job Production

The crawling process starts when the user pushes a single service to the *services topic* in Kafka. When a service is pulled from Kafka by a peer server, it crawls across the pages given in the root URL list, travelling over every found URL in parallel. When each URL is found, it is checked against the key-value store to check for **per-service** uniqueness, the provided list of

paths to ignore, and the **robots.txt** blacklist. If the URL is found in any of these, it is ignored, otherwise it is grouped into a Job and pushed to the *jobs topic*. Once all URL paths no longer yield unique URLs, or a size criterion is met, the process commits the offsets to the *services topic* as seen in figure 2-3.

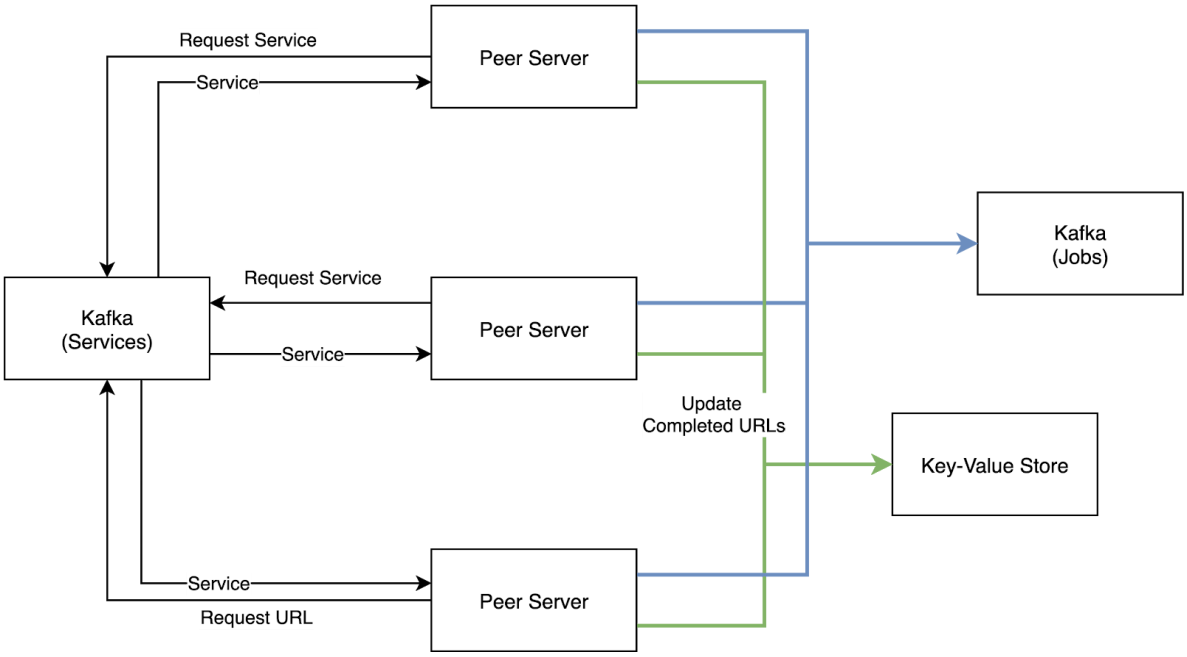

Figure 2. Data flow diagram for creating jobs, and storing data to key-value store

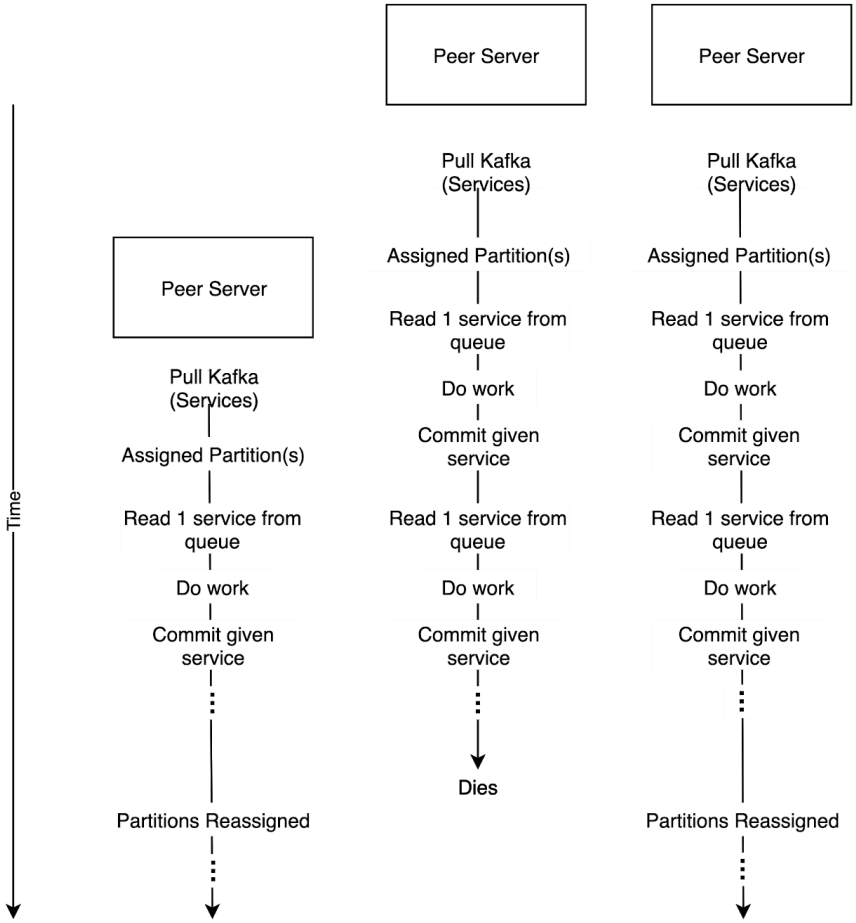

Figure 3. Job distribution system data flow and sequence diagram

Here the main change in our design philosophy becomes evident, initially we decided on having a leader server manage all communication from the *services topic* in order to manage job distribution and redistribution to ensure no lost or duplicated work. This would be done via the **RAFT** algorithm. However, as we realised the duplicated work does not need to be as strictly guarded against in this use case, we transitioned to a leaderless paradigm, where each Peer Server self manages its work, increasing scalability and robustness as was seen in figure 2 above.

There are plenty of places in which the Peer Servers can fail, and since there is no leader server to manage the work, it could become duplicated. However, since we only wish to guarantee at-least-once semantics, duplicated work is allowed. Since each Peer Server completes all work before sending commit messages to Kafka, we can guarantee that every Service gets processed at least once. Additionally, the parallel nature of reading and writing to the Key-Value store means that its likely certain URLs could mistakenly be added to the *jobs topic* multiple times, as there is a delay between a write and parallel reads, but again this process does guarantee each URL gets processed at-least-once.

One last complication in the crawling process is that most websites are too large for a single scraping node to comfortably process efficiently. In order to combat this, we utilize a process we call Recursive URL Expansion as seen in figure 4 below. We have modeled a website crawl as a tree, with each unique successive link followed representing a new level in the tree. We set a tunable parameter for the max number of levels a single Peer Server is allowed to crawl down. Once this limit is met, each set of "leaf node" URLs found by that crawl is bundled into its own *service definition* and pushed back upstream to Kafka, where it can then be consumed by other Peer Servers. By keeping all fields except the root URL list the same, we ensure that child services will be handled properly by the downstream *jobs topic*, while allowing us to balance the crawling of large websites over multiple Peer Servers.

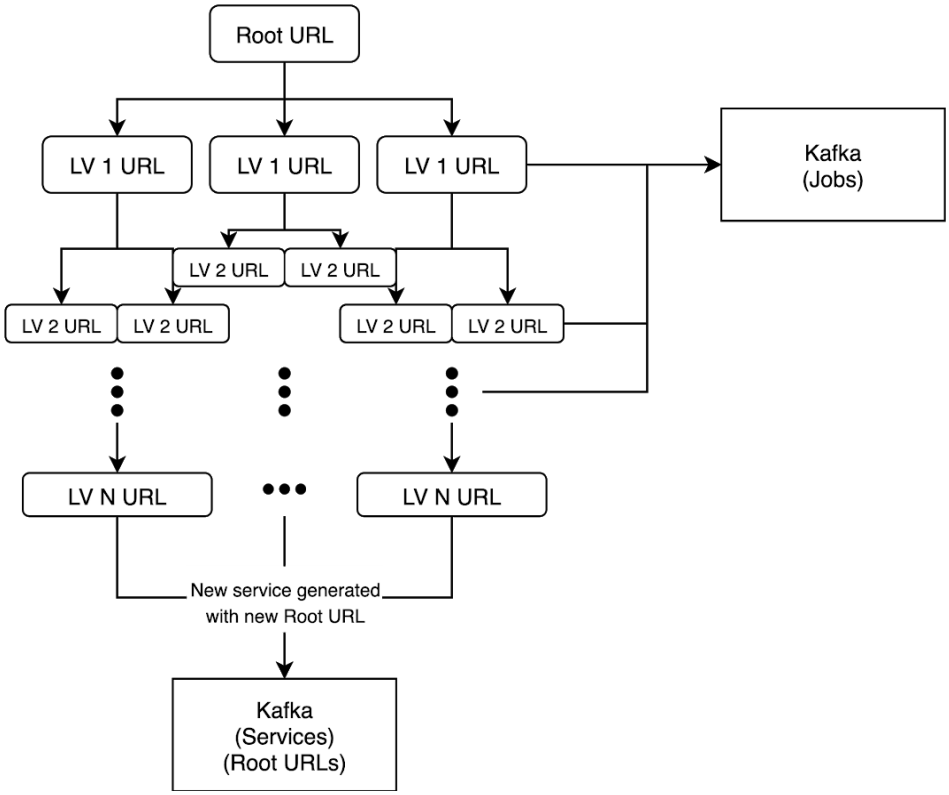

Figure 4. Recursive URL Expansion

Job Distribution

Distribution of jobs is handled mostly by Kafka, however there are some rules that the clients and producers must follow in order to ensure **at least once semantics**.

The Jobs Topic

All jobs are distributed via a Kafka topic called *jobs*. The topic is configured to have many *partitions*, at least one for each client (or proxy server) process. All client processes are part of the same **consumer group**, so Kafka will distribute the partitions amongst the available clients. By default Kafka will assign new messages to partitions based on the JAVA hash of their **key**. In our case we send the jobs to Kafka **without providing a key**. This causes Kafka to use a *round-robin* algorithm to assign partitions, ensuring an even distribution of jobs. This configuration ensures that clients receive an even workload, but has the drawback that there is no way to receive jobs from Kafka in any particular order. For our use case of distributed web scraping, the order in which the jobs are processed is much less important than the fact that they are **eventually** processed.

To ensure each Job is processed at least once, client processes must not commit any offsets until they have;

1. Processed the job at that offset
2. Sent the results to the *output topic* and received confirmation from Kafka
3. Send the Job to the *completed jobs topic* and received confirmation from Kafka

If a client process were to fail at any point between receiving the job and committing it's offset, their assigned partition(s) would (eventually) be given to another client. Since they would not have committed (unless the job was complete) the next client with access to the partition will process that job.

Though this approach ensures each job is processed **at least once**, It does not guarantee that a job will be processed **only once**. If a client does fail between processing the job and sending the commit message to Kafka, the next client with access to that partition will process the job a second time.

Scraping/Job Processing

The Client/Consumer process is responsible for processing jobs and pushing the resulting JSON data to an *output topic* where a user will be able to access it. It operates as follows;

1. Poll Kafka for job
2. For each URL in the job
    a. Get the HTML located at the URL
    b. Parse the HTML into a DOM representation (Document)
    c. For each Plugin defined for the job
        i. Ensure the plugin is loaded
            1. If it is not, wait for to be loaded from Kafka
        ii. Perform `plugin.scrape(Document)`

3. Send the results to the *output topic,* which is the topic with the same name as the *Service* defined within the job
4. Send the job to the *completed jobs* topic
5. Back to **1.**

An overview of such a system is shown in figure 5 below.

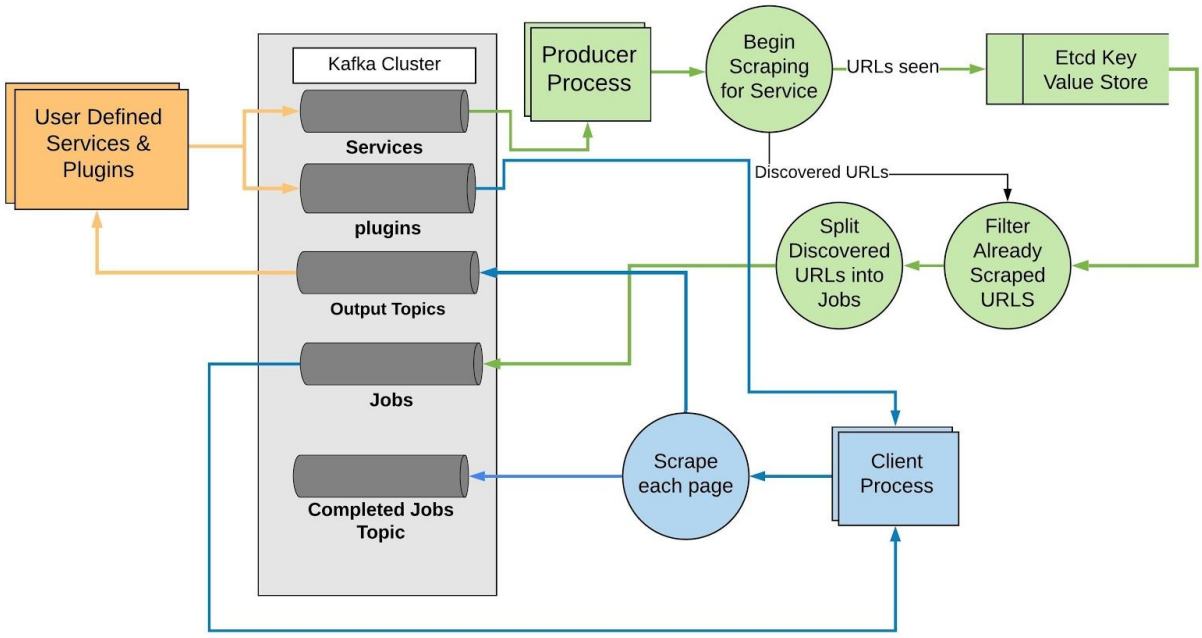

Figure 5. Kafka cluster communication

In addition to the standard client, there is also a **Proxy Server** which exposes jobs via an HTTP API. The purpose of the Proxy Server is twofold. Firstly it enables Android clients to access jobs, plugins, and to operate within the system. This is required as the Kafka consumer API is not available on Android. Secondly, it allows for increased parallelism beyond the number of **partitions** configured within Kafka, and reduces the number of concurrent connections the Kafka cluster has to deal with. Clients to the Proxy Server (referred to as **Proxy Clients**) operate similarly to the standard client processes, except that they acquire jobs from (and send results to) the Proxy Server instead of Kafka. This system is shown in figure 6 below.

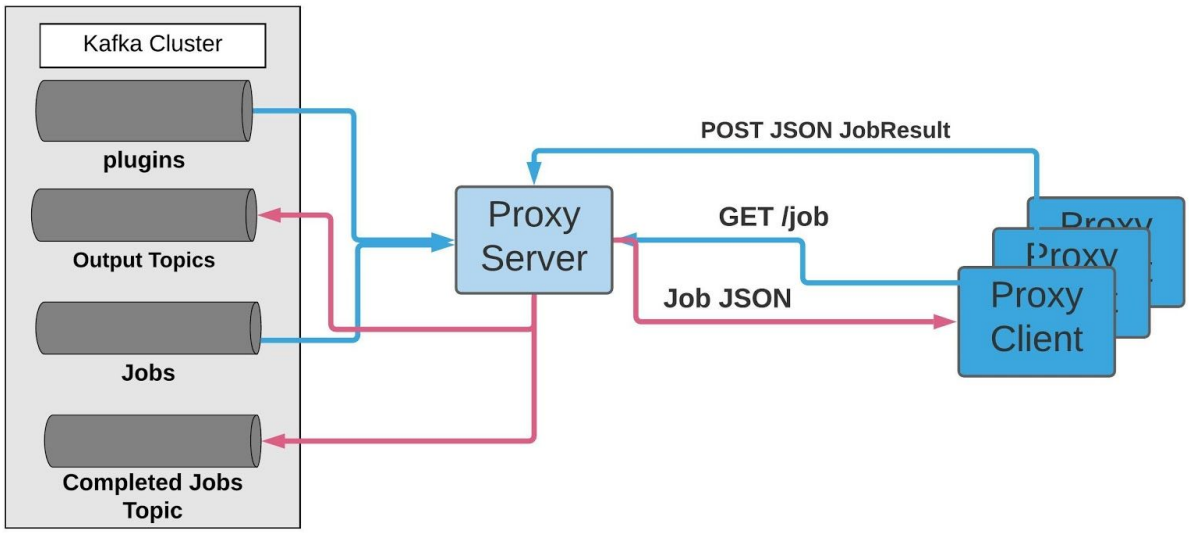

Figure 6. Proxy server and client communication

Plugins

The Plugin system allows users to expand the functionality of the system at runtime by defining new ways of processing web pages. The user implements a single functional interface;

```
interface Plugin {
   fun scrape(doc: Document): String
}
```

Their implementation is then packaged in a JAR file, containing any dependencies it requires. The user then pushes the packaged JAR to the *plugins topic* as a `ByteArray` using the provided *client library* (which also contains the `Plugin` interface). An example showing how to use the client library is shown in Appendix A

The plugins are stored in Kafka for the client processes to load upon startup.

Experiments

In this section we'll discuss what experiments we ran to analyze our solution, what the results were and some analysis on whether the results are satisfactory.

Methodology

To examine our implemented prototype solution we'll test how our solution performs in terms of its reads and writes per second to the key-value store and the performance of our solution with a specified number machines, producers and clients. The full system will be tested using 1 machine running on 1 producer and 10 clients and on 3 machines, with 4 producers and 50 clients. In particular the number of pages processed and the average processing times is examined alongside their respective distribution tables.

Results

When running the etcdctl command line, we are able to determine the performance of our key-value table which we found to be able to handle ~151 writes per second and ~20k reads per second. Of these operations, the slowest request took 0.029955s with a standard deviation of 0.002580s.

The number of pages processed, the average processing time, and a distribution table for 1 machine, 1 producer and 10 clients is shown in figures 7-9 respectively below. In the distribution table we observed a mode of 400ms.

Figure 7. Number of pages processed over time running on 1 machine, 1 producer and 10 clients

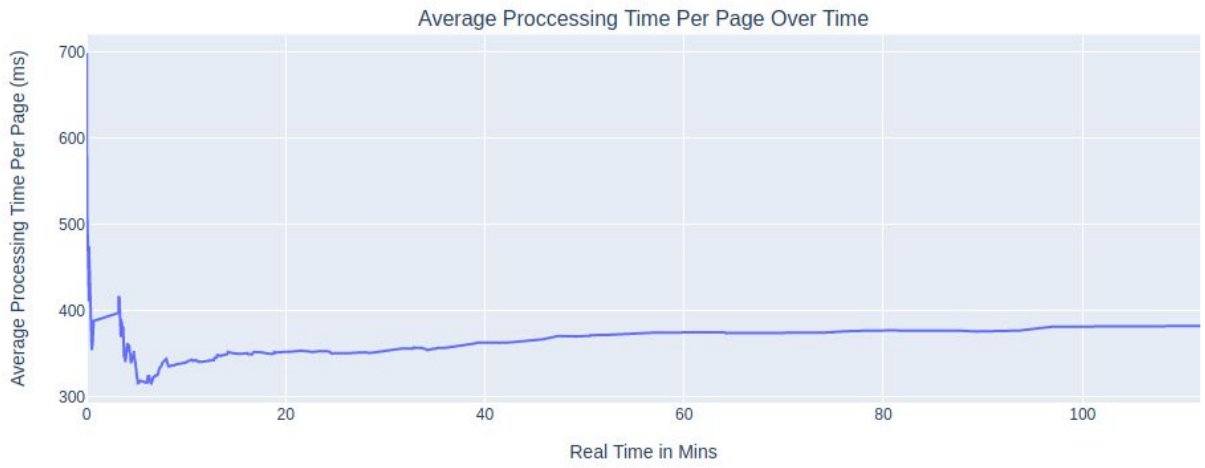

Figure 8. Average processing time per page over time running on 1 machine, 1 producer and 10 clients

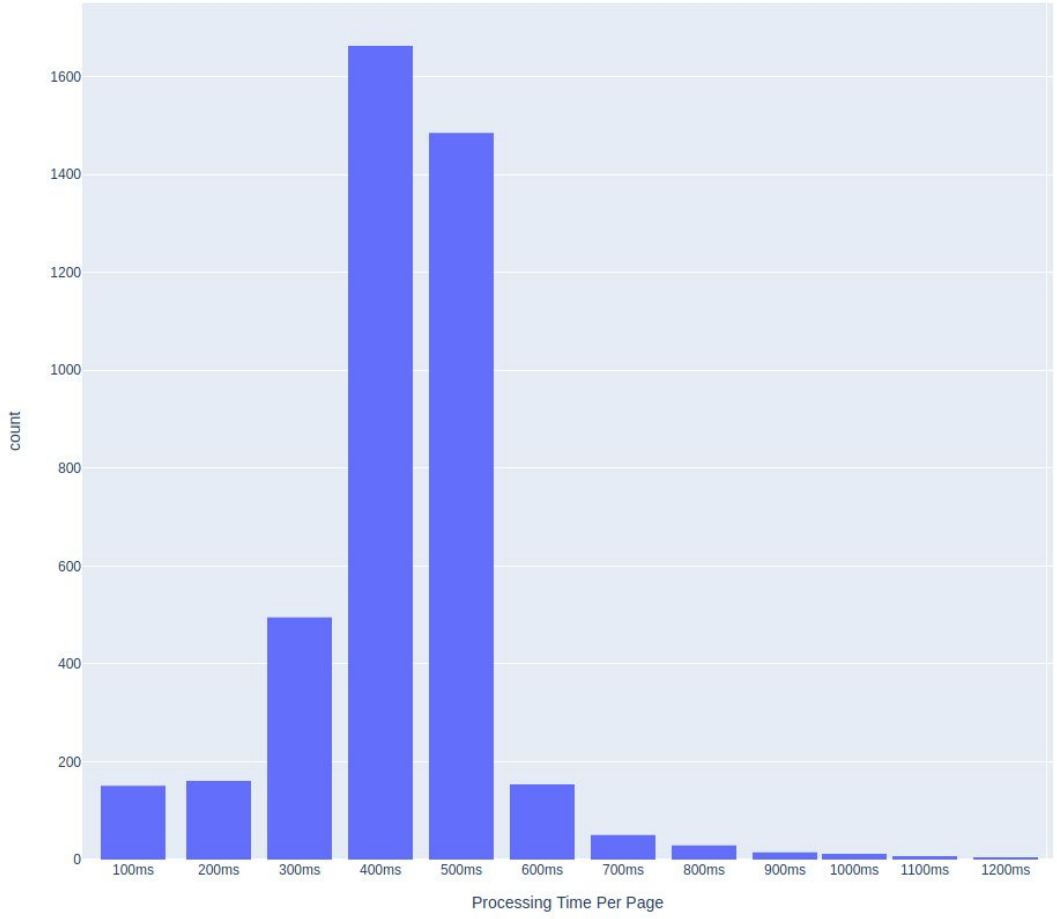

Figure 9. Distribution of processing time per page running on 1 machine, 1 producer and 10 clients

The number of pages processed, the average processing time, and a distribution table for 3 machines, 4 producers and 50 clients is shown in figures 10-12 respectively below. In the distribution table we observed that the mode of the processing time is 900ms.

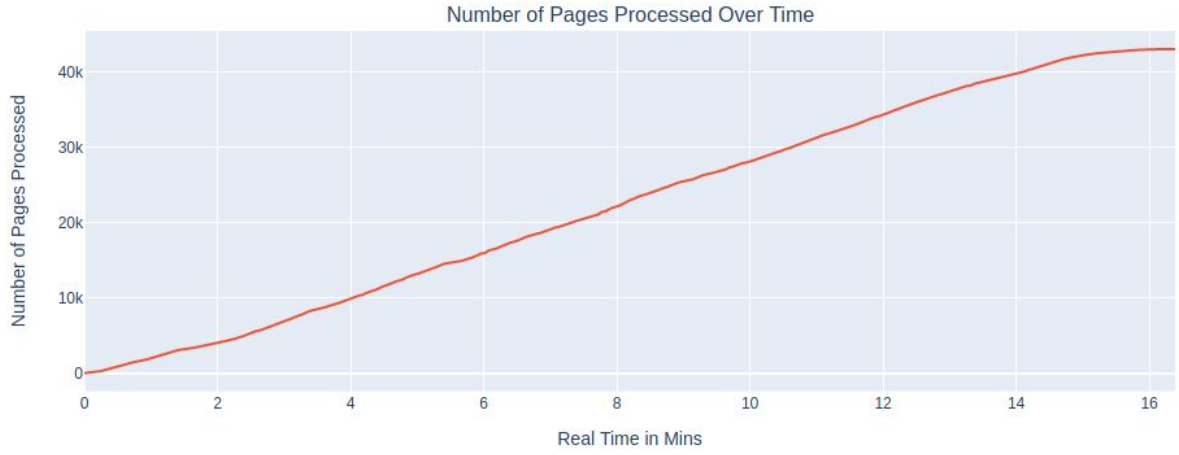

Figure 10. Number of pages processed over time for system running on 3 machine with 4 producers and 50 clients

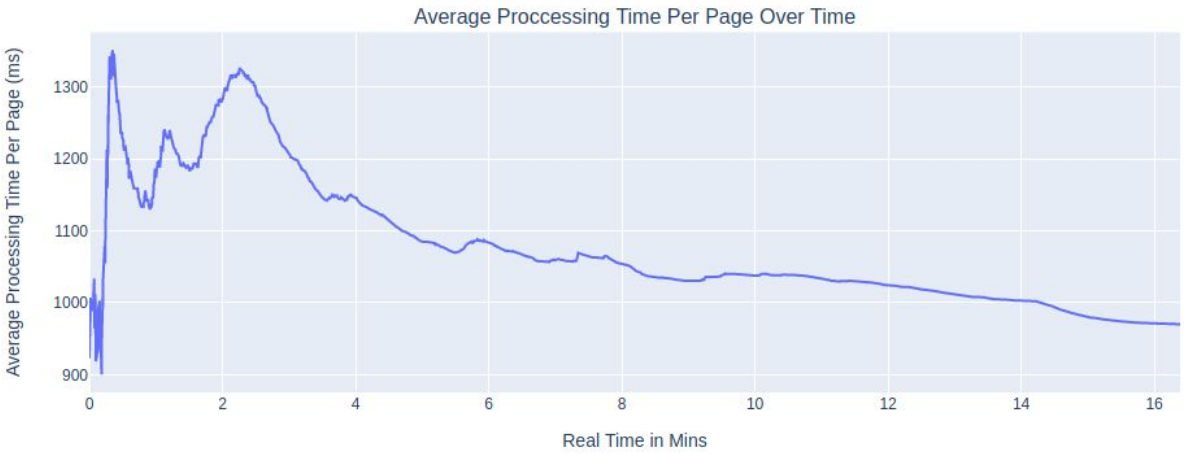

Figure 11. Average processing time per page over time for system running on 3 machine with 4 producers and 50 clients

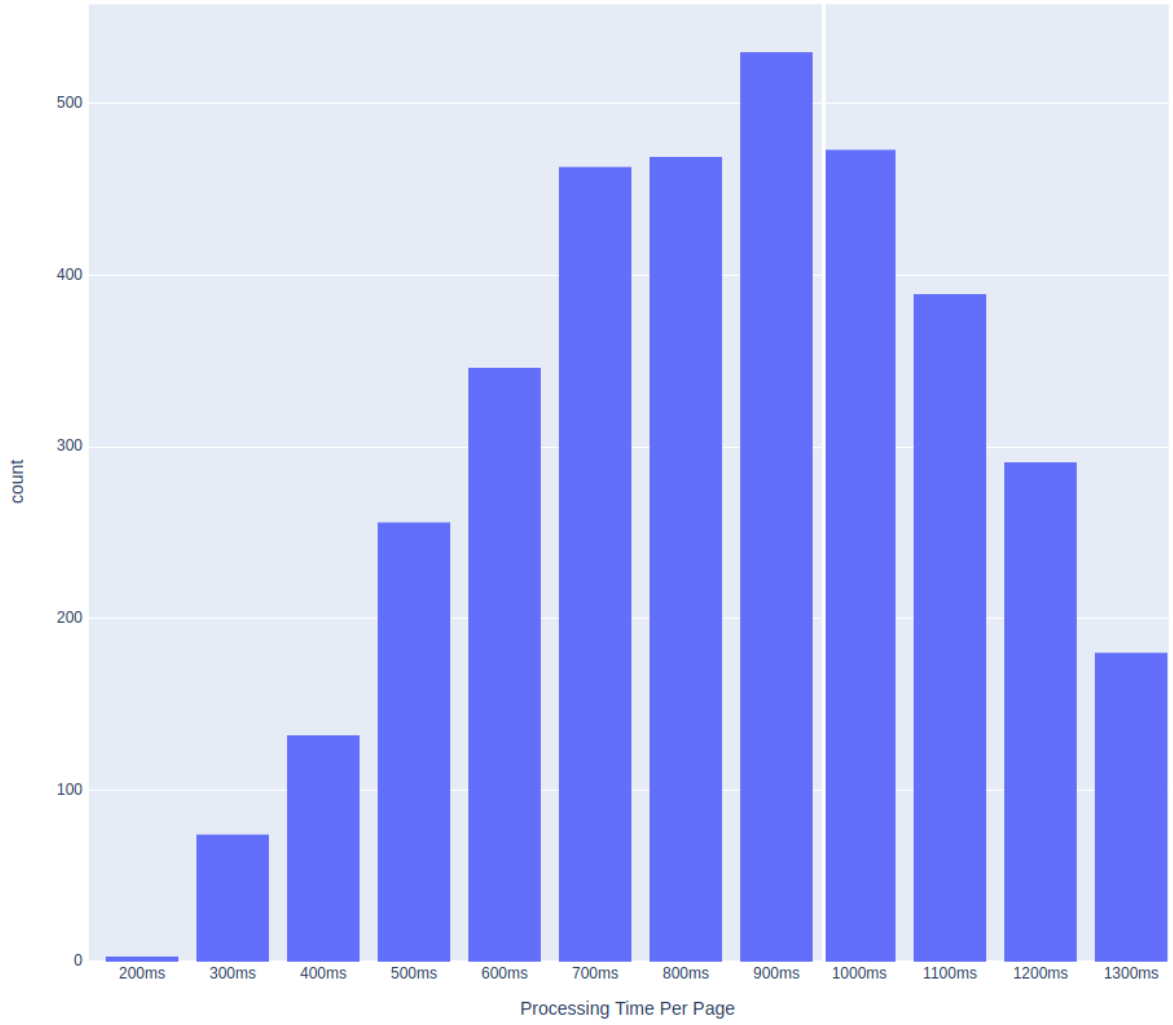

Figure 12. Distribution of processing time per page running on 3 machines, 4 producers and 50 clients

Analysis

Overall, our results were promising. Our testing has concluded that our implementation not only works over multiple producers and clients, but also that speed scales roughly linearly with the number of nodes. This is because we found on our personal computers that the limiting factor tended to be network bandwidth, even with upwards of 25 clients running on one machine. One cap that we did experience was the key-value store only being able to handle approximately 151 writes per second, however since the clients have a much more resource intensive task, 50 clients required 3 times more processing time to handle 1 producer. Additionally no work duplication, as mentioned earlier, was never a goal of this framework, so slow key-value writes do not impact the rest of the system.

Another irregularity in our data is the prominent spike in average processing time at the beginning of each test. We believe that this is due to the concurrent starting of both our runtime codebase, as well as long-term processes like etcd and Kafka starting at the same time. In a real distribution, these processes would already be running in the background when a job starts, and so these latency spikes will likely not be present.

Additionally, the overall lower response times in the 10 client test is due to all communication happening over localhost, causing massively decreased latency. This is not an indication that fewer nodes perform better, as scaling the nodes did increase the amount of work completed. The second test also reflects the latency present in a much more distributed setup.

Future Work

In completing our initial prototype we identified several areas in which we could add additional functionality or improve the usability. If we wish to monetize this solution, we would have to host our solution as a service. With this comes the requirement to provide a **REST API** to our service so that interfacing with the cluster can be done more easily. Currently there are also no stats to view as the program is running so a web based dashboard for stats could be helpful.

In our current implementation we guarantee at-least once semantics for processing our URLs, but to optimize our performance we can examine how to go about providing exactly-once semantics so that a url is never executed more than once. Further exploration is also needed in implementing modular plugin functionality instead of hardcoding it on phones, since in our current solution we only have desktop clients connecting to our cluster using modular plugins. In particular, we have to create a JAR file with all the dependencies in order to run the functions on phones since Android uses a different java virtual machine.

Some more trivial quality of life improvements include:

- More control for the user
    - Start and stop on demand
    - Specify how many urls in a job
    - Service struct for defining depth to traverse before calling recursive crawl
    - Rate limiting defined by service definition
- Optimize plugin creation
- Error and logging viewing
- Optimize speeds on various functions
- Parse robots.txt and to check interval and rate limiting
- Streamline moving processes to cloud computing resources
- Allowing for more fine grained distribution of resources among tasks

Conclusion

In this project we aimed to provide a distributed platform solution for web scraping as a service. To accomplish this we examined some inter process communication tools such as Cassandra and Kafka and settled on using Kafka in favour of its flexibility and scalability. In terms of storage we examined some DHT solutions, and other frameworks with coordinate information between clusters including Zookeeper, Consul and etcd. We selected etcd for its simplicity to implement and the ability to accomplish all our necessary functionality such as being highly available and reliable. We then examined several implementations of job distribution and processing and decided on a leaderless implementation. In our experimentation we were able to confirm that our system is able to perform distributed web crawling across several clients with near linear scaling in the number of pages processed over time. Going forward we will examine hosting and modular android plugin features alongside various quality of life improvements for user control.

References

[1] V. Shkapenyuk and T. Suel, *Design and Implementation of a High-Performance Distributed Web Crawler*. Brooklyn.

[2] P. Boldi, B. Codenotti, M. Santini and S. Vigna, "UbiCrawler:A ScalableFullyDistributedWebCrawler", *Citeseerx.ist.psu.edu*. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.80.8704&rep=rep1&type=pdf . [Accessed: 14- Aug- 2020].

[3] "Front Page - Folding@home", *Folding@home*, 2020. [Online]. Available: https://foldingathome.org/home/. [Accessed: 14- Aug- 2020].

[4] Elisabetta Raguseo, *Big data technologies: An empirical investigation on their adoption, benefits and risks for companies,* [Online]. Available: https://doi.org/10.1016/j.ijinfomgt.2017.07.008 . [Accessed: 14- Aug- 2020]

[5] "Benchmarking Cassandra Scalability on AWS", *Medium*, 2011. [Online]. Available: https://netflixtechblog.com/benchmarking-cassandra-scalability-on-aws-over-a-million-writes-per-second-39f45f066c9e. [Accessed: 14- Aug- 2020].

[6] "Apache Kafka", *Apache Kafka*. [Online]. Available: https://Kafka.apache.org/uses. [Accessed: 14- Aug- 2020].

[7] "stvp/dht", *GitHub*, 2015. [Online]. Available: https://github.com/stvp/dht. [Accessed: 14- Aug- 2020].

[8] "nictuku/dht", *GitHub*, 2020. [Online]. Available: https://github.com/nictuku/dht. [Accessed: 14- Aug- 2020].

[9] I. Education, "What is etcd?", *Ibm.com*, 2019. [Online]. Available: https://www.ibm.com/cloud/learn/etcd. [Accessed: 14- Aug- 2020].

[10] "Consul by HashiCorp", *Consul by HashiCorp*. [Online]. Available: https://www.consul.io/intro/vs/zookeeper.html. [Accessed: 14- Aug- 2020].

Appendix A: Client Library

The library is written in **Kotlin**, but should be accessible from other JVM languages. However the API may be less friendly to use from java or other JVM languages. Particularly consuming and producing to topics relies heavily on Kotlin coroutines & Flows.

As of now the library needs to be included manually using the output jar file in . In future we would like to have the library available via *Maven*. In intellij you can add it by selecting

`File > Project Structure > Modules > Dependencies` and clicking the + icon.

Or add this to the build.gradle for your project

```
repositories {
    ...
    flatDir {
        dirs "path/to/dir/containing/library"
    }
    ...
}

dependencies {
    ...
    compile(name:"clientlib.definitions", ext:"jar")
    compile(name:"clientlib", ext:"jar")
    Implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.3"
    implementation 'org.apache.Kafka:Kafka-clients:2.0.0'
    implementation group:'org.jsoup',name: 'jsoup',version:'1.11.3'
    implementation 'com.google.code.gson:gson:2.8.6'
    compile("org.jetbrains.kotlinx:kotlinx-serialization-runtime:0.20.0")
    ...
}
```

To create a plugin, simply implement the Plugin interface provided in the *clientlib.definitions* library. The following is the example Word Count plugin we used in our testing.

```
import com.google.gson.GsonBuilder
import csc.distributed.webscraper.definitions.plugins.Plugin
import org.jsoup.nodes.Document

data class WCResult(val word: String, val occ: Int)

class WCPlugin: Plugin {
   private val gson = GsonBuilder().setPrettyPrinting().create()
   override fun scrape(doc: Document): String =
       doc.allElements.map { it.text() }
           .flatMap { it.split(" ") }
           .filter { it.find { !it.isLetter() } == null }
           .groupBy { it }
           .mapValues { it.value.size }
           .map { WCResult(it.key, it.value) }
           .let { gson.toJson(it) }
}
```

The scrape function receives a Jsoup Document as a parameter which provides a JQuery-like interface to the DOM. The function returns the JSON data which should be collected from each webpage.

Once your Plugin implementation has been packaged in a jar file with it's dependencies, you can write a program to send it, and a Service definition, to Kafka.

```
...
import csc.distributed.webscraper.Scraper
import ca.blakeasmith.kKafka.jvm.*

@Serializable
data class MyPluginResult(...)

fun main() {
    val myService = Service(
      name = "example",
      rootDomains = listOf("https://www.scrapethissite.com"),
      filters = listOf("#", "/dontlook", "/garbage path"),
      plugins = listOf("myPlugin")
    )

    val myPlugin = File("path/to/myPlugin.jar")

    val config = KafkaConfig(listof("<ip>:<addr>"))
    Scraper.Services(config)
             .write(myService.name, myService)
             .close()

    Scraper.Plugins(config)
             .write("myPlugin", myPlugin.readBytes())
             .close()

    Scraper.Output(myService.name, MyPluginResult.serializer().list)
           .read("my_Kafka_group_id") // returns a Flow
           .forEach { (url, pluginName, pluginResult)
               println(pluginResult)
           }.collect()
}
```