WHIRLPOOL: Data Acquisition using N-node Distributed Web Crawler

Report Compiled by: Rihan Stephen Pereira
Advisor: Dr. Michael Soltys
MSCS Graduate 2018-2019

October 10, 2019

Contents

List of Figures

1 | Introduction

A *web crawler* is a long running program that collects pages from the web and stores them to the disk either in raw format or after extracting its contents. Sometimes they are also referred as *random walker*, *worm*, *spiders*, *robot*. Crawlers serve variety of purposes. Web search engines are very well-known systems powered internally through crawlers. Other uses are web data mining for performing analytics, archiving portions of the web - a relevant active project is the Wayback Machine from The Internet Archive[22] operating since 2000. Most recently, there is an ongoing attempt from Software Heritage[18] that continuously crawl & archive publicly available source code repositories on the web.

This thesis is an attempt to implement a small-scale, special purpose crawler for collecting data about various job postings from a chosen set of job portals active on the web. As far as the scope of this thesis is concerned, the work done here does not dive into cleaning & analyzing the data set but instead focuses on its system design, implementation, and deployment. The crawler architecture builds on a open blueprint design of *Mercator*[8] which describes the design in enough detail but leaves out implementation specifics to developers.

1.1 Contributions

- **Use of Message Broker:** In one case, the papers doesn't provides information on how the subsystems that make up a crawler are connected together. This thesis makes use of message broker to facilitate collaboration between components thus providing asynchronous, event-driven style of communication. It also highlights its strengths & weakness.
- **Crawl Ordering Problem:** A more crucial issue is without having a target to crawl, the problem of crawling becomes endless, infinite crawling even for specialized crawler such as this. This thesis discusses the crawl ordering strategy to shape its crawling only to specific content.
- **Distributing the crawl:** Furthermore, this thesis shows the shortcomings of linear hashing used for partitioning hosts to distribute the crawl and instead employs consistent hashing[25] algorithm and discuss its merits over it.
- **Development strategy:** Given this complex program at hand, it becomes obvious to separate the concerns by developing each component in isolated pieces. This thesis leverages docker containers to build the project in stages. It allows to split the development and production environment by building images for the target platform to run the component.
- **AWS Services:** For running the crawler activity, this thesis makes the most of AWS 12-month free tier access by combining together suite of services and sketches an infrastructure to setup, deploy and demonstrate the program.
- **Microservices:** Lastly, this thesis explains the philosophy, principles behind microservices [17] by relating it to work done in this thesis.

1.2 Motivation(the big ideas)

A long term goal of this project is to built the infrastructure to run basic data science algorithms on the collected data and to tune the program characteristics to balance coverage & freshness(refer section 2.3). A web crawler has its place at the very bottom of the pyramid(figure 1.1 doing instrumentation and logging.

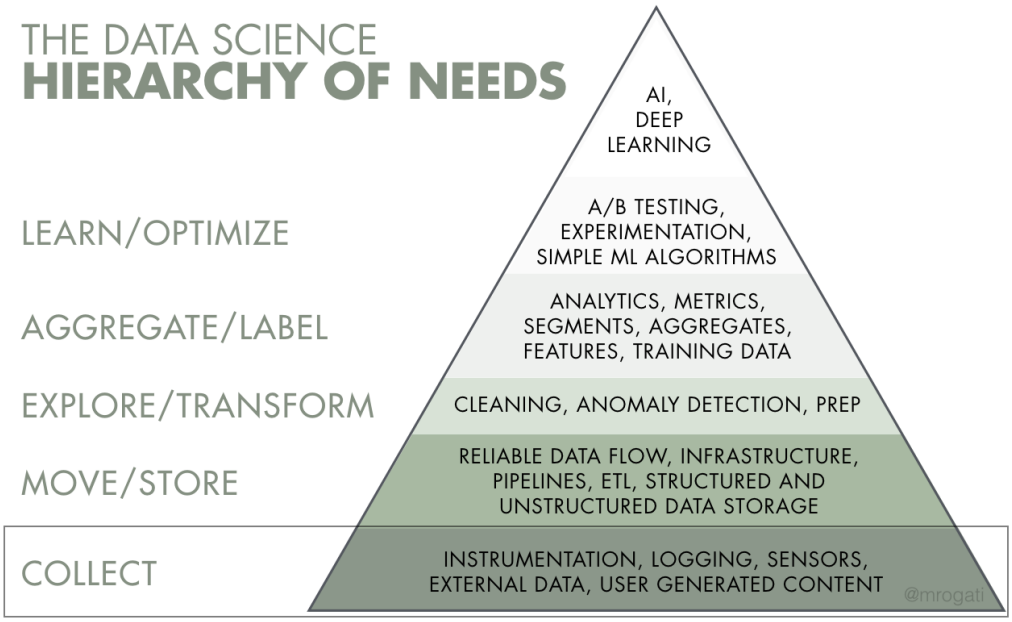

Figure 1.1: Source: Monica Rogati's "The AI hierarchy of Needs"[23]

Data science is used to build data products, optimize business activity, and also to gain insights into consumer behavior. Lot of companies are eager to adopt it but all these things can be achieved correctly by building the layers of foundational work first.

According to Monica Rogati,

"Think of Artificial Intelligence as the top of a pyramid of needs. Yes, self-actualization (AI) is great, but you first need food, water, and shelter (data literacy, collection, and infrastructure)."

The skill and experience for building such data-intensive applications fall under the discipline of data engineering[24]. It is a superset of *data warehousing*, specializing around the operation of distributed systems, stream processing, and handling computation at scale.

A web crawler can be seen as a fancy project and its complexity can easily be overlooked. Given the apparent simplicity of basic crawling algorithm, it posses numerous challenges. First of all, crawling the whole web is simply unrealistic even for the world's most advanced crawlers. Considering the current scale of web and while its still evolving, it is important to think thoroughly about characteristics of a crawler, because this brings control in the hands of developers, it enriches the value of problem they are trying to solve, it gives them a choice to either seek broad coverage over fresh content or maintain a balance between the two. Even a small scale version of a crawler can be built to be sophisticated and scalable. Moreover, building a crawler sets a playground to address problems such as concurrency, throughput, and load balancing. It is a program quite different from traditional, client-server paradigm that can fanout from single source file to modular, independent services that interact with each other through endpoints.

When discussing a system like crawler, people often critic saying that

"Stop worrying about scale. You're not Google/Amazon. Just use a relational database."

Well, that is true; building for scale without enough evidence from the metrics such as logging, locks you into an inflexible design. It is also a form of premature optimization. Along the same lines, there is also an urge felt to overengineer and loose focus on simplicity. Engineers loves to solve puzzles and challenge of building complex software. Overengineering is basically building a solution that is much more complex than is really necessary. This occurs when developer at a job tries to predict every possible use case and every edge case, eventually losing focus on the most common use cases.

However, the author of this thesis believes that it important to survey different technologies as each have their own strengths and weaknesses and choose the right tool for the job. The inclusion of message broker, docker, and AWS services to build the crawler are introduced with a mindset of promoting good design which will allow one to add more details and features later on. The design begins with a reasonable level of abstraction and with multiple iterations will gives better results.

2 | Background

This chapter primarily attempts to familiarize the reader with enough groundwork related to web crawler. It also elaborates on the miscellaneous resources that are integrated to support the objective of this thesis. It begins by explaining the basic steps taken by a crawler, followed by describing key features that underpin its system design. Properties of a crawler are discussed in the following section. A chronological walk through of highly established crawlers and key differences are noted later in the section. A section is dedicated to explaining the role of Message Queues. The remainder of this chapter covers different AWS services and docker used to interweave an environment for executing a complex program.

2.1 Basic Crawling Algorithm

The basic operation of any HTTP based crawler goes like this:

1: Let I $\leftarrow$ {1,2,3,4,5} such that seed set S = $\{U_i |$ i $\in$ I$\}$
2: $U_f \leftarrow$ S; where U_f is a Frontier queue
3: **procedure** SPIDER(U_f)
4: **while** $U_f \neq \emptyset$ **do**
5: $u \leftarrow$ Pop(U_f)
6: $p \leftarrow$ Fetch(u)
7: $T \leftarrow \exists p$ [{Extract(p, t) | t is a text }]
8: $L \leftarrow \exists p$ [{Extract(p, l) | l is a link }]
9: $U_f \leftarrow U_f \cup L$
10: $\exists u$ [{Delete(U_f, u) | u is a already fetched URL}]
11: **end**

The first run of a typical crawler program visits the links U_i from a finite seed set S, shown above. The set S is copied to frontier queue U_f which handles the logic to output the most appropriate url L to be visited. In a simple setting, the frontier queue U_f can be imagined as a classical FIFO queue, however, it is much complex assembly of queues in real-world crawlers. While queue U_f is not empty, url u is pulled out from the frontier. The $fetch$ function yields HTML page p after visiting u. Next step, page p is scrapped to obtain textual data T and list of new links L. Finally, list of links are merged into U_f. care is taken to disregard l that is already queued in U_f.

2.2 Features of a Crawler

The web is not a centrally hosted data warehouse but instead is comprised of billions of independent web hosts, operating within its limits. Taking this into consideration, a web crawler exhibits most of the following traits:

- **Being polite to web hosts:** A crawler shouldn't induce too much burden on target host at the time when they crawl. By not regulating this behavior, it is blocking the web site from pursuing its purpose or business and eventually there is a higher possibility of have ramifications on the operator running the crawler.
- **Resilent to spider traps a.k.a infinite loops:** Some hosts mislead crawlers into fetching page after page that never reach an end. Thus, a crawler should be robust to such fallacies.
- **Distributing load:** A crawler can make itself capable by distributing the load across multiple machines. Distributing the load across multiple machines is also a *shared nothing architecture*. It can make the crawler download web pages based on geographical proximity reducing request-response latency.
- **Scaling:** Scaling and Distributed may sound similar but subtle difference is a scalable crawler's sole purpose is to cop up with load as it comes, in this case, pacing the crawl rate as well as enhancing the crawl rate. With scalability in mind, the architecture needs to be rethought on every order of magnitude of load increase.
- **Page quality:** A crawler should seek broad coverage by being an explorer but at the same time it should be biased towards first selecting websites it is intended to crawl.
- **Extensible:** Like any complex program, a crawler system design should be flexible and extensible on top of its core modules.

2.3 Crawl Ordering Problem

The purpose of a web crawler can be viewed as traversing the web graph. The crawling order appears naturally to be breath-first problem (download pages following the links as they appear). But since the web is so massive and still expanding such a crawling activity is considered infinite and the crawler itself can be said to be *random walker* with no purpose. Most of the commercial or even open source crawlers have some sort of crawling order policy built into their system. It is a must for small-scale, special purpose crawler to order the extracted URLs because the act of acquiring content is restricted by various factors.

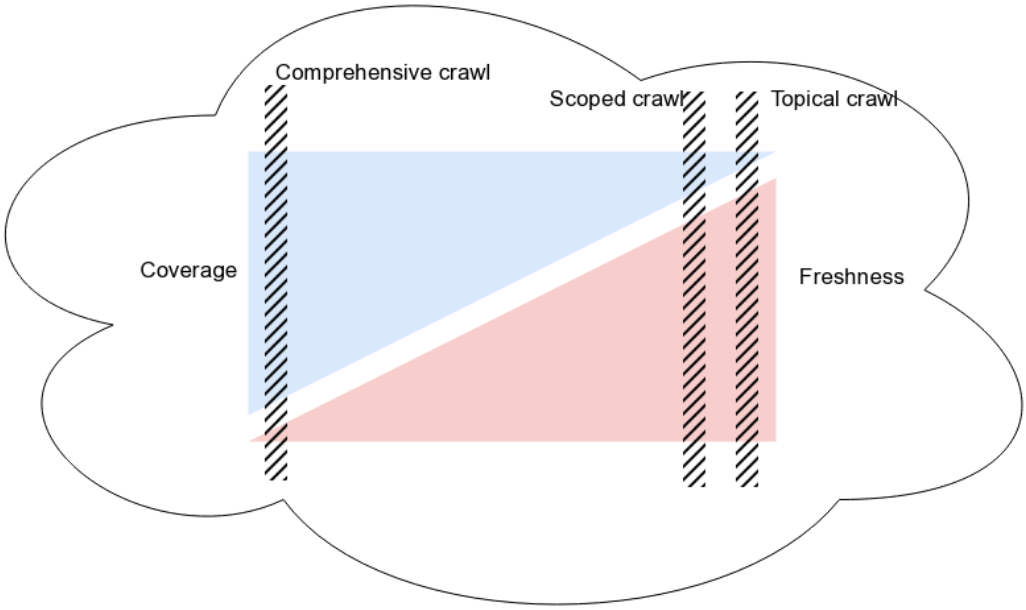

Figure 2.1: Crawl ordering based on Coverage vs. Freshness

Coverage: Focus is on fetching pages that crawler deems required.
Freshness: Here the focus is on revisting, redownloading already visited pages. Since web 2.0, pages are capable of dynamically changing content and popularity of Single Page Applications(SPA) has enabled rendering data through view logic controlled by javascript.

As shown in the figure 2.1, *Comprehensive Crawling* focuses on maximizing coverage by reaching out to content of all types. A particular page p is weighted important by counting external links outside current site's perimeter. Crawl order policies like Breath-first Search, PageRank, and other variations are applied. Such activity is carried out by Commericial Search Engines.

On the other hand, *Scoped Crawling* narrows its crawling activity to certain portion of the web i.e only crawl a specific category (e.g everything about Stock Market). The crawling order is determined based on a degree to which a page p falls under its umbrella. In constrast to Comprehensive, Scoped crawler is suitable for data-mining task

by performing data collection on topic of interest.*Topical Crawling* is a form of Scoped Crawling in which page relevance is determined by - given a page p links to another page p' and is in-scope(within bounds) either directly or through a series of subsequent links. There are 3 techniques to order URLs to crawl.

1. Fish Search (Binary classificaiton). Here, the relevance is measured by treating links within visited page either relevant/irrelevant upto a certain height.
2. The relevance is estimated by doing textual analysis on text surrounding a link in a visited page p that points to yet-to-be visited page q.
3. Train and query a Machine Learning classifier to get the score of set of topics of interest to crawl.

This crawler has to balance its greediness level by considering page relevance. It is possible that a already relevant page p may not always yield a new relevant page but it might would after going through several links of irrelevant pages. If the crawler is designed to be too greedy then it will have zero coverage which in turn means it won't discover new pages in its journey. A decay factor can be used to allow irrelevant fetches with an ultimate intention of finding relevant page of interest.

Finally, freshness is the attribute that makes the design of crawler being either continuous or non-continuous. The question of balancing content coverage and freshness in a crawler is solely a decision of entity behind it. A *batch crawling* design requires the restarted of a crawler at periodic interval to download revised content of previously crawled pages. A *continuous* or *incremental crawling* design would be technically a program that never terminates to revisit previously crawled pages.

2.4 Related Works

Web crawling is well studied topic, several crawlers emerged and vanished, some designs weren't open while some were not documented. This section briefly talks about history of web crawlers that have existed and have made notable design decisions each passing year.

Wanderer was the first crawler that was written in Perl in 1993. It ran on a single machine. It was used for collecting statistics and later used to power first search engine.

MOMSpider came into existence in 1994. It rate-limited requests to the web servers by introducing *Robot Exclusion Policy*. It allowed crawler operator to exclude sites. No focus on scalability. It used DBMS to store URL's and state of crawls.

Internet Archive [22] crawler addresses the challenges of growing web. It was designed to crawl 100 million URL's. Used disk-based queue to store URL's to-be crawled. Made use of bloom-filter to determine whether a page was previously crawled and if so, that page was not visited again. It also addressed politeness concerns and optimized *robots.txt* lookups. Apache Hendrix is a open-source out-of-box crawler used by the Internet Archive project entirely written in Java.

Mercator [8] came out in 1999. The paper discusses the blueprint design of its crawler. Initially it was non-distributed. It addressed social aspects of crawling through its frontier scheme. The announcement of 2nd paper incorporated host splitting component that made the crawler execution model distributed. It was extensively used in web mining projects. It ran on 4 machines for 17-days crawling over 891 billion pages. The program was designed with extensibility in mind and implemented fully in Java. The blueprint design is discussed in much detail in section 2.5 as it forms the basis of crawler designed for this thesis.

The **Polygot** web crawler also had distributed design. It had crawler manager handling multiple downloading process. It ran on 4 machines for 18 days and crawled over 120 million pages.

IBM Webfountain crawler was distributed, featured multi-threaded crawling process called 'ants'. It was polite and can be configured to change politeness policies on fly. Re-downloading of pages a.k.a freshness was based on historical change rate of pages. Webfountain was written in C++ & used Messaging Passing Interface(MPI) to allow collaboration between processes. It was deployed on 48 nodes.

Ubicrawler [9], 2004, was yet another scalable distributed web crawler. It employed consistent hashing algorithm to cope up with addition/removal of nodes designated for carrying out crawling task. The consistent hashing is explained in detailed in chapter 3 section 3.3. By using this technique it demonstrated graceful performance degradation in the event of failure. It was deployed on 5 nodes and dowloaded 10 million pages a day. This program was written entirely in Java.

Multicrawler [11], 2006, focused less on performance and crawl rate but more on detecting, transforming multiple formats of semantic web data.

IRLbot [10], 2009, is the most recent comprehensive crawler studied and implemented for over 3 years. It discusses its own implementation to address the increasing complexity of verifying unique URLs accumulated over time. Most notably it provides a solution to mitigate spider traps created by web sites. IRLbot was able to successfully crawled 6.3 valid HTML pages in a single day.

Apart from this, there exist open source web crawlers written in various languages notable ones are Apache Nutch(Java), Scrappy(Python).

2.5 Mercator Architecture

This section proceeds into detailed description of Mercator[8] web crawler surveyed in section 2.4. Figure 2.2 is a high-level composition of the same. As seen, Mercator specializes different steps defined in basic crawling algorithm covered in section 2.1 and adds several other steps to address the social and scalability challenges of web crawler. Its blueprint design offers horizontal scalability and therefore if implemented can run on more than one node. Each part that contributes to mercator crawler can also be referred to as a component, module, subsystem, etc. They can be used interchangeably from this point onward.

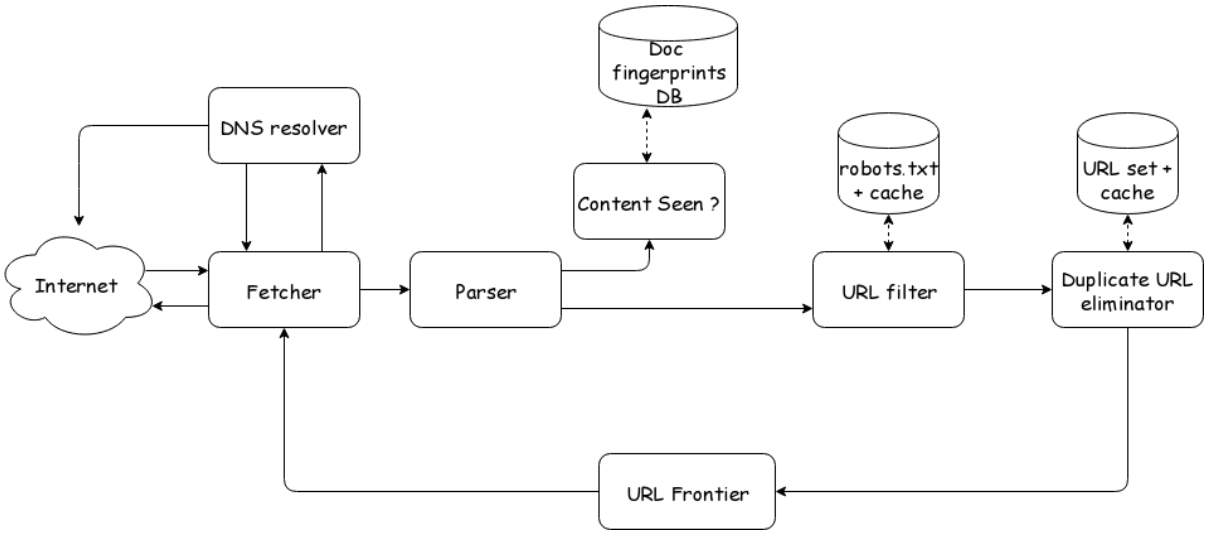

Figure 2.2: High-level organization of Mercator

Each work cycle begins and ends at the URL Frontier. Depending on what needs to be achieved, the entire crawler can run as a single process or can be partitioned into multiple processes - treating each subsystem as a process. Given a single URL, it goes through the cycle of being fetched followed by passing through various checks, filters and eliminations, then finally returned to the Frontier (for incremental crawling).

At the beginning of each logical loop cycle, a worker thread pops a URL from the Frontier data structure adhering to the priority and politeness policy. The output URL is then fetched by HTTP fetcher subsystem - which like any other web client contacts DNS module to get IP address of the corresponding web server. The web page is then retrieved and stored at a temporary location which is then picked up by parser subsystem which forwards the path to the page and extracted links to content seen and URL filter, respectively. The batch of URLs undergo a fixed set of pre-processing steps at the URL filter subsystem before being passed over to Duplicate URL eliminator (DUE). The DUE module tests each URL to determine whether the link should be added to the URL frontier.

2.5.1 Fetcher, Parser, and DNS

It occurred to designers of Mercator that the task of fetching each URL from seed set or newly discovered links is complex. First, the request made to the web server at a given URL endpoint will have several outcomes as a response. Therefore, it becomes a bottleneck to implemented fetcher as only single threaded, blocking synchronous module. Instead it should be written as a multi-threaded, synchronous I/O or non-block, asychronous I/O to speedup the operation. Moreover, it is also inefficient to have DNS resolve a given URL's IP address each time a fetch request is made to the same web server. This can be fixed by having a cache of lookup pair for each web host in the frontier. According to Mercator designers, another difficulty is the lookup implementation itself of DNS entries is blocking; meaning that at a time only one request is considered and completed and all other requests queue up. The solution for this is yet again a multi-threaded approach or a asynchronous I/O wrapper.

2.5.2 Handling De-duplication

Any professional or sophisticated crawler has the capability to minimize De-duplication. De-duplication simply tests whether a web page with the same content has already been seen at another URL. As per figure 2.2, the content seen subsystem takes care of the same. There exist three established methods to solve this problem.

Document Fingerprinting(checksums)
A digital fingerprint uniquely identifies a file by mapping large data inside a file to a shorter string. This method is straight forward way to control data duplication. Widely deployed fingerprinting functions are one-way cryptographic hash functions such as *MD5*, *SHA*, and *bcrypt*.However, this simplistic approach fails to detect near duplicates which is much needed for tools like crawlers

Bloom Filter
This is a data structure which gives a probability of same document being already present in the data store. One downside is sometimes it also gives false positives, meaning, the filter outputs saying 'the document already exist but in fact it doesn't'. This can be used for avoiding near duplicates but at the cost of missing unseen files due to false positives. This can be tackled by some application strategy.

Shingling
In many cases, the contents of one web page is similar to other differing only by few keywords, for example - the published date and time. This technique does textual analytics to detect near duplicates. Shingles[20] improve over bag of words technique because it ignores the context of words. Shingling overlaps phrases of neighboring few words at a time much like shingles on a roof. Minhash and Simhashing are best known non-trival implementations for near-duplicate detection.

2.5.3 URL Filtering

The URL Filter component takes input a batch of extracted URLs, applies a series of tests, modifiers on each URL, composes a newer batch of URL which fall in its criteria and forwards it to the next subsystem. For instance, assume a set of links $\{U_1, U_2, U_3, U_4, ..., U_i\}$ extracted from page p, there is a possibility that some U's link are relative to the page p, in such cases, this component normalizes such U's turning them into absolute URLs. Also, the crawler can enforce a rule to exclude out of bound URls - those that don't fall within a given list of domains.

```
User-agent: *
Disallow: /admin
Disallow: /jobman
Disallow: /reports
Disallow: /talentmatch
Disallow: /profman
Disallow: /regman
Disallow: /ows
Disallow: /config
Disallow: /m2
Disallow: /jobsearch/
Disallow: /job
Disallow: /feed/
Disallow: /resumepost
Disallow: /profile/
Disallow: /rss/
Disallow: /salary-calculator?title*
Disallow: /jobs?q*
Disallow: /jobs/?q*
Disallow: /canyouhackit/
Disallow: /career-paths?title
Disallow: /salary-calculator-for-tech-hiring
Disallow: /daf/servlet/
Disallow: /jobs/dc-*
Disallow: /jobs/djt-*
Disallow: /products/?
Disallow: *?CMPID*
Disallow: */jobs.html
Allow: /jobs
Allow: /register
Allow: /mobile

User-agent: Googlebot-News
Disallow: /
```

Figure 2.3: Robots Exclusion Policy set by dice.com

Apart from these, many hosts regulate whats allowed/not-allowed for downloading by placing a *robots.txt* file under root directory of a hosted web site. The standard is known as Robot Exclusion Protocol. Figure 2.3 shows a *robots.txt* of dice.com. Its interpretation beginning at line one goes like no crawler should visit */admin*, */jobman* pages, but it can visit */jobs*, */register*, and so on. A user-agent containing string Googlebot-news is not legally allowed to crawl any of its pages. For each domain, the crawler fetches robots.txt to test whether the URL under consideration passes the robot restrictions and only then it is added to its Frontier data structure. Given a high locality that many of the extracted links fall within the domain, it is efficient to cache robots.txt rule into an in-memory data store. The cache expires and redownloads robots.txt for each domain several times a day to stay in sync with the rule changes.

It should be noted that it is always safer and courteous to include a note in request header of the crawler indicating intentions to download the data, and also provide

your email address where the webmaster can contact you in case of any issues. Also, it is a good practice to enforce strict compliance with domain's robots.txt rules.

2.5.4 Duplicate URL Eliminator (DUE)

A batch of URLs qualified from URL filtering subsystem arrives at DUE. Sometimes it s also referred to URL-seen test. DUE guards the URL Frontier by eliminating multiple instances of the same URL from adding to the Frontier. It also keeps history of set of URLs that were previously added to the URL Frontier and those that are currently in it. The fire-and-forget, one-time crawl URLs are only crawled once. In continuous crawling Frontier scheme, some URLs are revisited periodically for new information, in such cases, the DUE has to delete its traces from its state.

Understanding the behavior of DUE explained in above paragraph, the size of URL set stored in relational database on disk will grow linearly as the size of web corpus grows irrespective of whether the crawler is continuous/non-continuous. Since the DUE has to make sure that it isn't adding duplicate URL to the Frontier, it will check against each entry in the URL set table. This will hamper DUE throughput and increase disk seek latency over time i.e the time taken to check and respond for existing entry increases. Each insertion in the URL set table is a fixed size checksum of textual URL along with the mapping to URL itself. The checksum algorithm should be such that it has exponentially small probabilistic bounds on the likelihood of collision in the URL set.

To reduce the number of operations required to DUE each URL in a batch, several optimizations can be combined. First, keeping an in-memory cache objects of popular URLs. The intuition for this is continuous crawling URLs endpoint will be revisited. But again with this, the cache hit rate is low as ratio of continuous to one-time crawls can be 20:1. Also as the computed checksums of the URLs are highly unique, it has lower locality, so the checksums that miss the cache will require disk access and disk seek. Secondly, taking advantage of high locality of checksum hostname(domain name) concatenating with checksum of absolute URL for that domain name results in same host names being numerically closer to each other. This in turn translates to access locality on the URL set.

2.5.5 URL Frontier

The URL Frontier subsystem receives a batch of URLs from its counterpart(it is either DUE or hostsplitter in case of Distributed crawling). It maintains those URLs in the Frontier and propagates them in some order whenever the fetch module seeks a URL from it.

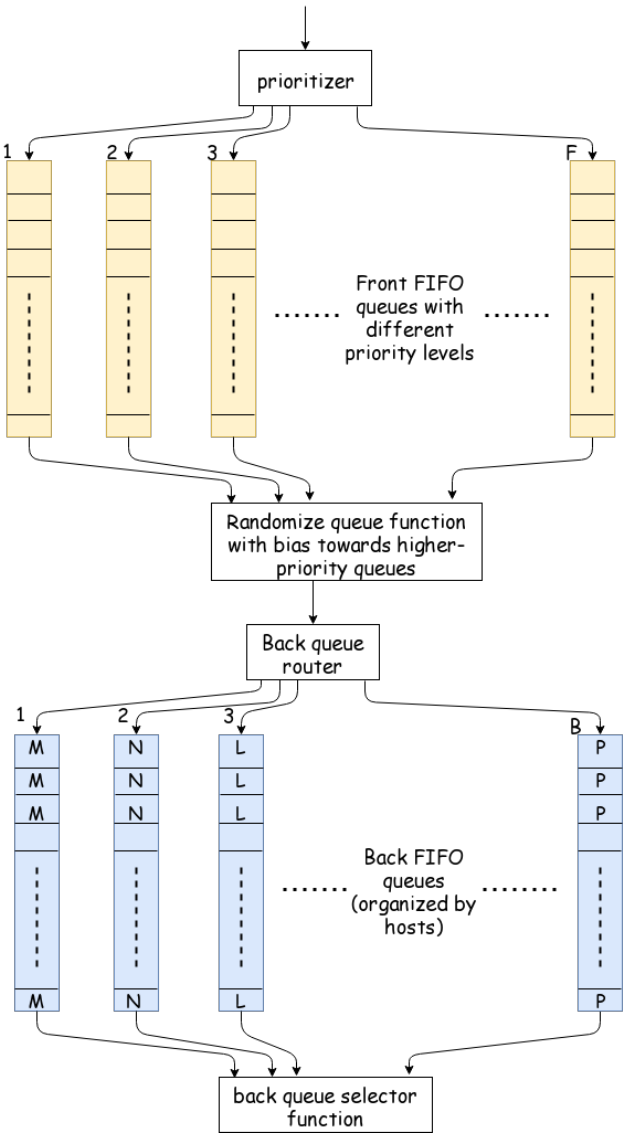

Figure 2.4: URL Frontier Scheme(based on Mercator)

Figure 2.4 is not just a simple queue but a complicated data structure. It includes multiple pages from the same host. It avoids trying to fetch all those pages from the same host at the same time because may be the crawler can overload the server which is not a good policy. Also, the fetch module should not wait doing nothing, instead keep the module busy. Thus, it balances the tradeoff between not bombarding the server and also keeping the fetch module busy so that if fetch is not crawling that website at least it can go to some other server and fetch some pages from there.

Following are important considerations that dictate the order in which the URLs are returned or pulled from the frontier. As mentioned earlier, simple priority queue fails as there is high locality of pages that go to its own site, creating a burst of accesses to that site.

- **Explicit politeness:** obeys any specifications from webmaster on what portions of the site site can be crawled, see section 2.5.3 for more details.
- **Implicit politeness:** Avoids hitting any site too often even though no specification exist on the site.
- **Freshness:** crawl some pages more often that others.

A common heuristic to handle politeness is to insert a time t gap between successive requests r such that,

$$t(r_i) > t(r_{i-1})$$
$$t(r_i) = (K.l(r_{i-1})) + TS \qquad \text{...}K \text{ is a variable delay factor}$$

meaning the time taken by successive request $t(r_i)$ is order of magnitude larger than time taken by most recently fetched $t(r_{i-1}$ request. K is the Crawl-Delay attribute found in web hosts robot-exclusion protocol. It varies from host to host and its presence is basically to inform target crawler agent that it may revisit the host in integer N, where N is usually seconds. On web servers where this attribute is not specified, general rule of thumb is to set K to 10 sec. $l(r_{i-1})$ is the latency of the request and TS is the timestamp. K, l, and TS is the time buffer which is specific to this thesis crawler implementation and is the earliest time $t(r_i)$ at which the host can be hit again.

It is important to visualize URL Frontier diagram figure 2.4 before diving into its pieces. A URL follows the path into the Frontier and immediately encounters set of front queues F. It enters into one of these F queues. There are also another set of queues called B back queues. At some stage the URL will be pulled out of the front queue and then sent into one these back queues. Later, at some stage, it will be pulled out of that back queue and accessed by fetcher thread which then goes and actually fetches the document at that URL endpoint.

Each front queue F_m in range $\{1, 2, 3...m\}$ and back queue B_n in range $\{1, 2, 3, ...n\}$ is literally a $FIFO$ queue. The incoming URL gets classified into one of the $1, 2, 3, ..F$ front queues

- **Front queues** F_m manage prioritization. They influence the rate at which you ping the same web server again and again.
- **Back queues** B_n enforces politeness.

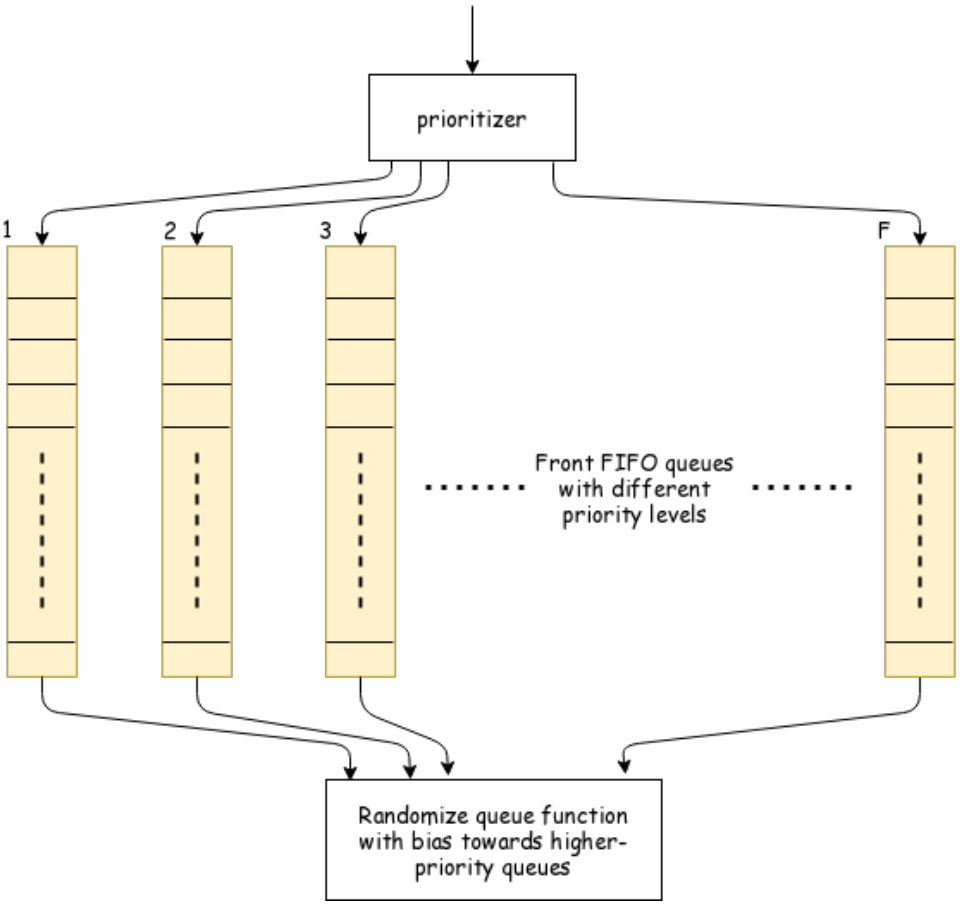

Figure 2.5: Frontier Front Queue

Figure 2.5 gives a close inspection on Front Queues F_m. Incoming URL is assigned a integer priority between 1 & m. Based on the priority assigned it is going to enter that particular queue. priority F_1 holds URLs for web servers that need to be crawled very very frequently. For instance, let L_s be links, $\forall L_s \in$ *dice.com* $\longmapsto F_1$ has highest crawling requirement whereas $\forall L_s \in$ *jobs.com* $\longmapsto F_m$ contains URLs which has least crawling requirement.

Heuristics for assigning priority,

- keeping track of how frequently the web pages are changing on a particular web server and how authoritative the pages are, if it occurs a case where a refresh rate is pretty high for any web server and pages happen to be important enough then those web servers are going to be crawled more often
- Application specific.

The prioritizer function for whirlpool is explained in chapter 3, subsection **??**.Also randomize queue strategy on how the items get pulled out of this F Front queues is explained in chapter 3, subsection **??**. This sums the top half of the picture.

The back queue B exist to achieve politeness because the crawler shouldn't hit the server too frequently. Again, there are B back queues from $\{1, 2, 3, ...n\}$, so when the URLs are pulled out in biased order from F Front queues, they are going to enter into one or more of these back queues.

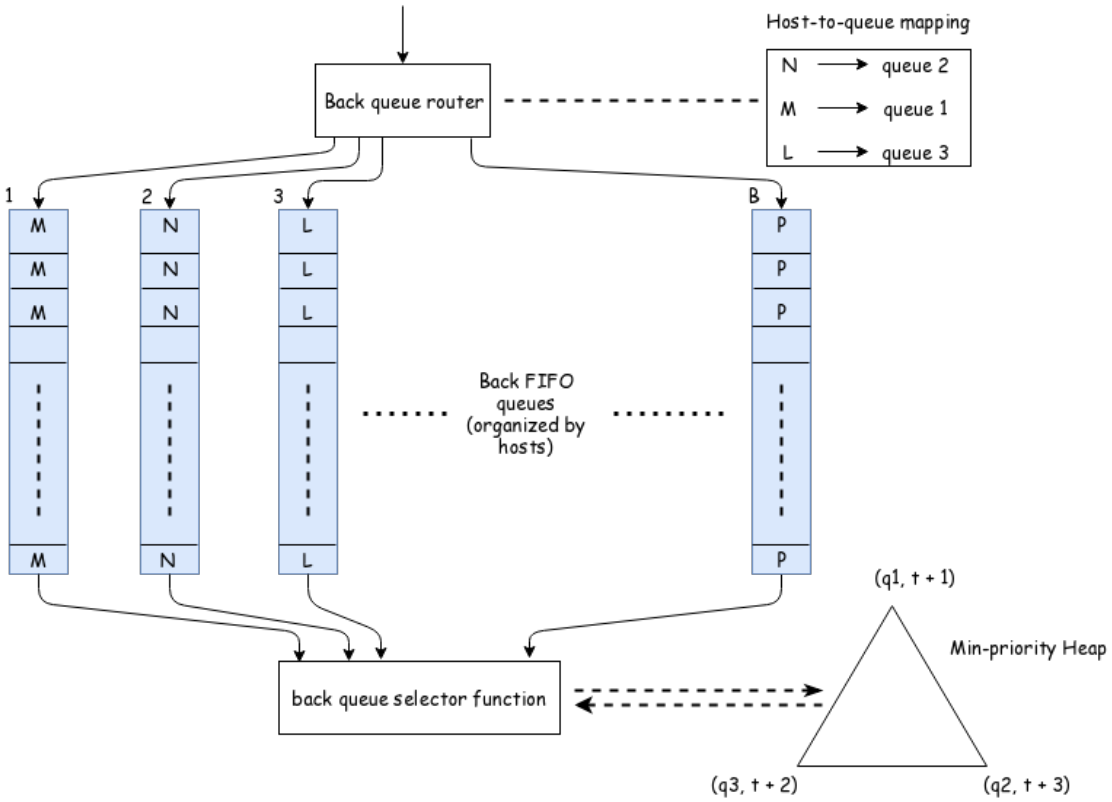

Figure 2.6: Frontier Back Queue

Recall the heuristic to handle politeness explained earlier, each back queue B_n corresponds to a particular domain. For e.g back queue B_1 is for server *monster.com*, B_2 could be the server *indeed.com*, and so on. So all the URLs which have base name *dice.com*, when they emerge from the front queue enter queue B_1. Furthermore, Back queues maintains a minimum priority heap data structure in which the keys are such that the parent key k is less than the key of both of its children i.e $k < 2k$ and $2k + 1$ and so the smallest item in the heap is at the root of the heap.

The size and vertex $V(q, t)$ of the heap is B. the key q from vertex pair (q, t) in it maps to one of the servers/back queues B_n. The value t is going to be threshold timestamp th_{ts} which is the value of time before which I am not allowed to hit the same server B_n again. The very fact that back queue structure maintains a minimum heap is that its particular root represents the server which has the least value of the timestamp. Its the very server that is allowed to query next before it queries any of those other servers. A URL u is pulled out from the head of server queue q and fetched by fetch module. This is handled by back queue selector function

While this cycle continues, the back queue router ensures the server queues B_n are non-empty. It is busy with pulling a URL u' from the head of front queue picked up by some randomize front queue function. Following this, it checks whether server queue

q exists for u‘, if so, then it gets added to that queue. It goes again to pull another URL w, this time it sees that server queue q is now empty and there insert w in it and renames the q to q‘. During the same iteration, the process sinks a new vertex V‘$(q$‘$, t$‘$)$ on to the min-heap based on the meta-information of w such as time t‘.

2.5.6 Distributing Web Crawl

A crawler program as a whole can be distributed for various reasons. A distributed crawler can crank up crawling rate. This is achieved by making a identical clone of single node crawler and replicating it onto another machine. Another advantage of distributing the crawl is that a subset of websites are crawled by a machine which is geographically closer to its data center/region. The immediate question is how does mercator extend its architecture and also how do these nodes communicated and share URLs is shown in figure 2.7.

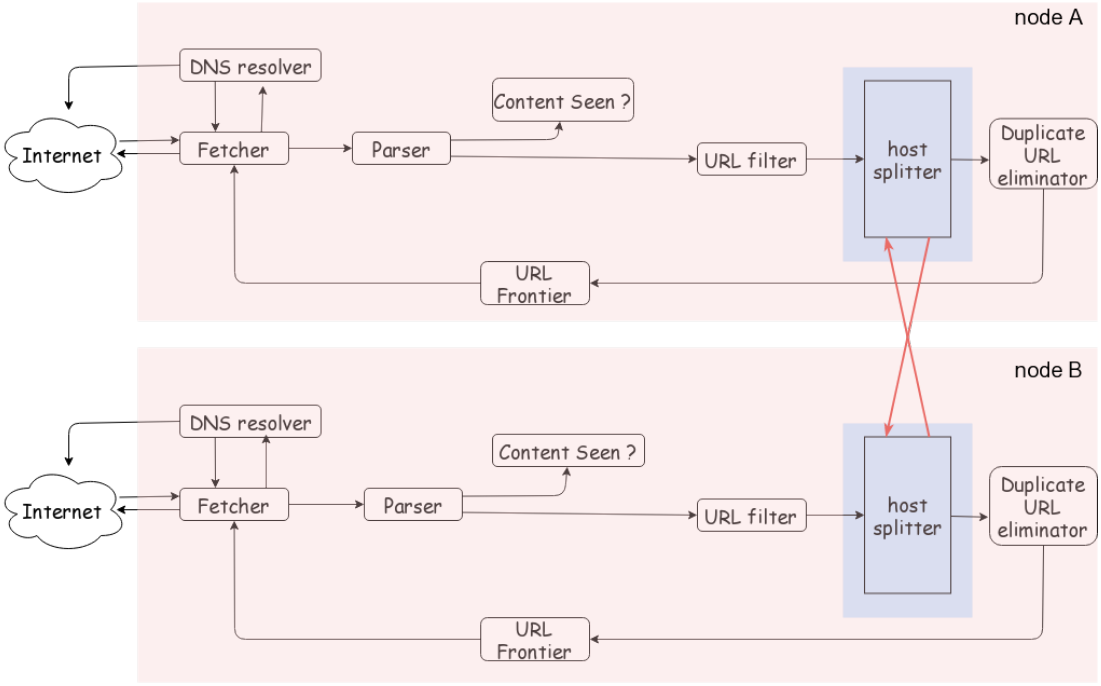

Figure 2.7: Host Partitioning

Lets say there are 3 different nodes designated to crawl the web M_1, M_2, M_3. All the hosts that are being crawled are partitioned into 3 sets. Imagine taking every single web server on the web or in the seed set, taking the URls for those servers, and hashing those URLs in the integer range $i \in \{1, 2, 3\}$. For instance, web server address *usjobs.com* could get hashed to an integer 2. So all those URLs that are being stored by that web server will be parsed by node M_2 while the other two nodes do not fetch documents located on *usjobs.com*.

The key to the implementation is the host splitting component shown in figure 2.7. The component at machine M_1 computes which machine is meant to crawl the URLs of *usjobs.com*. It takes the URL, looks at its server's address and hash it to the integer 2 upon which it knows M_2 is supposed to crawl URLs located on this server and therefore sends it to node M_2. Looking at the snapshot of node M_2, it has passed content seen, URL filter tests which confirms the incoming URL from different node is valid and need not go through the tests again. The only thing node M_2 has to check is whether that URL already exists in its own local URL frontier version or its own DUE. If the URL is new, the node will add it to its own URL frontier. At the same time node M_1 simultaneously receives URLs that it is supposed to crawl from those other nodes $\{M_2, M_3\}$.

2.6 Software Design Principles

This section mentions few core design principles necessary to maintain healthy codebase and knowledge to build scalable software. The design patterns are abstract and universally applied in different programming paradigms.

2.6.1 Open-Close Principle

When a certain code is designed with an intent to extend it but does not need any modification whenever requirements change or when new use-case is requested, such code is said obeying open-close principle. In other words, it is open for extension and closed modificaiton. This design process gives more flexibility to the program and make future changes cheaper.

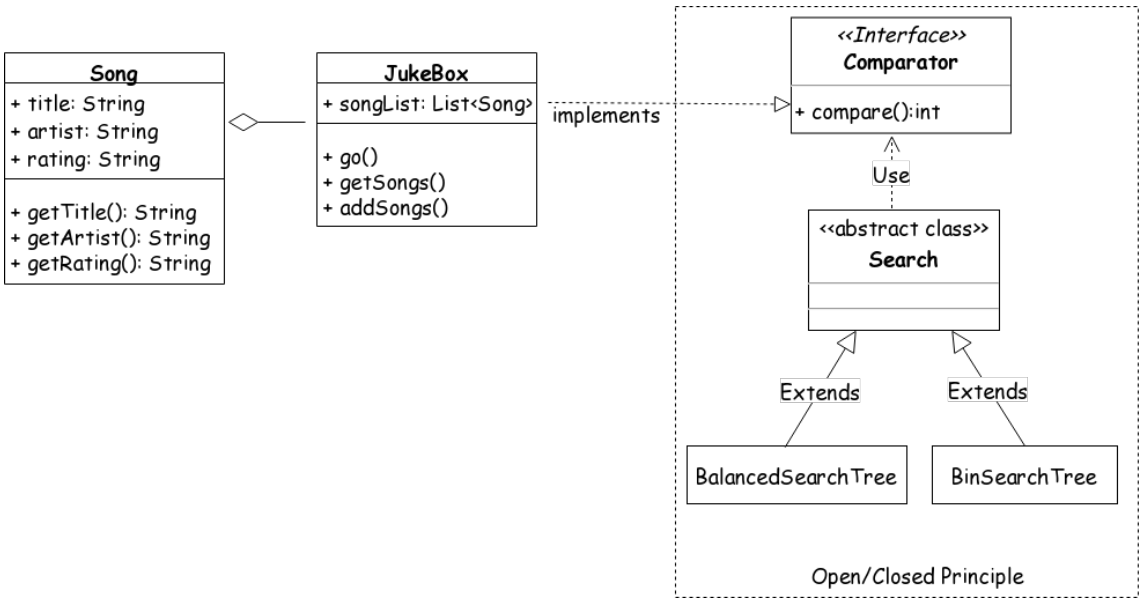

Figure 2.8: A example JukeBox program using Open/Close Principle

Consider an example of this principle in image 2.8, the Jukebox inner class implements Comparator interface, overriding compare method to search on different attributes of the class (eg. search on title, artist, etc). That method is invoked from concrete implementation of abstract Search class. algorithm upon calling Collections.search(). The search functionality offered by two concrete class vary independently and their runtime doesn't affect each other. Even more search algorithms can added without affecting existing search. The same comparators in JukeBox can also be used to perform sorting on songs called by respective sorting algorithm overriding sort method from sort interface.

2.6.2 Dependency Injection

Dependency Injection(DI) promotes open-closed principle and reduces loose coupling. It is one of the most important topics applicable to almost all software produced. DI provides references to objects the class depends on instead of allowing class to gather dependencies by itself. The dependent class doesn't take into account the how, where and what of implementation. This makes great impact on the flexibility of software design.

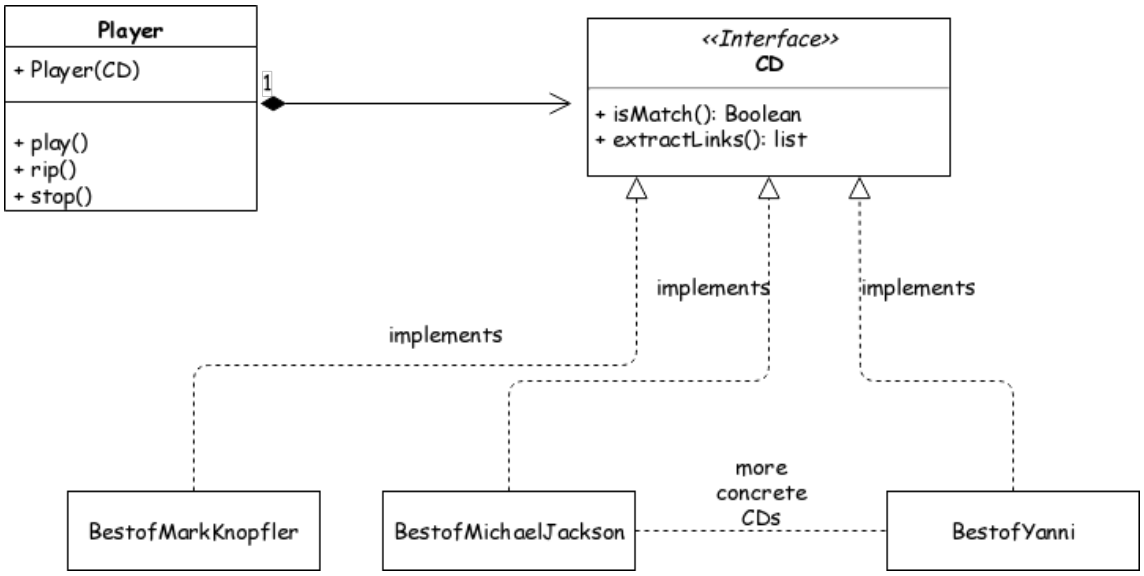

Figure 2.9: CD Player using Dependency Injection

Figure 2.9 shows an implementation using Dependency Injection. The Player class is a dumb box; it does not know anything about Compact Disc(CDs) shown above i.e BestofMarkKnopfler, BestofMichaelJackson; it is dependent on its client to provide with working instance of CD. Without DI, the Player can create and play only records that were harcoded in its implementation. Anytime, a new records need to be played, Player class needs changes to its code. This violates Open-Close principle. DI makes the code reusable and increases unit tests.
DI can be achieved in multiple ways depending on the programming language in use. For dynamic languages like javascript and python, the support for higher order functions can perform dependency injection whereas static, class based language such as java, DI is achieved using DI frameworks.

2.6.3 Dependency Inversion

Dependency Injection is subclass of broader principle called Dependency Inversion. Its aim is same as DI - to make the class as simple and less coupled to the rest of the system. Inversion concept is observed in MVC frameworks. Its promotes the philosophy of coding to contract. Contract being the creator of plugins and not worrying who will use them; the inversion figures out which class to instantiate. It can load dependencies on a scale of 100.

Inversion is identified as 'Hollywood Principle' - Don't call us, we will call you. Features found in a piece of software supporting DI framework include:

- expands code only through plugin/extensions
- plugins are independent and can be added or removed any point of time.
- system auto-detects plugins, configures which plugins should be used and how
- defines interface for each plugin type

2.6.4 Principles for designing scalable systems

When a piece of software needs to grow to a size requiring horizontal scalability, careful thought must be given to tradeoffs between endless scalability and practicality of each solution. Also assessment of not overengineering the solution needs to taken into account. Three techniques that help design scalable systems. Each one has different advantages and different cost.

- **Adding identical copies of components**: This is easy and most common scaling strategy applied to application build from scratch. Identical copies of components or server equally serve incoming request. A client request to any one of random cluster of clones yields correct results. This scaling strategy is not restricted only to client-server applications but also applicable to autonomous program like web crawlers. Mercator crawler described in this report uses exact copies of crawler onto individual machines and distributes traffic through its host splitter component.
- **Functional partitioning**: Identifying parts of the system focused on specific functionality and creating independent subsystems out of them. Figure 2.10 shows

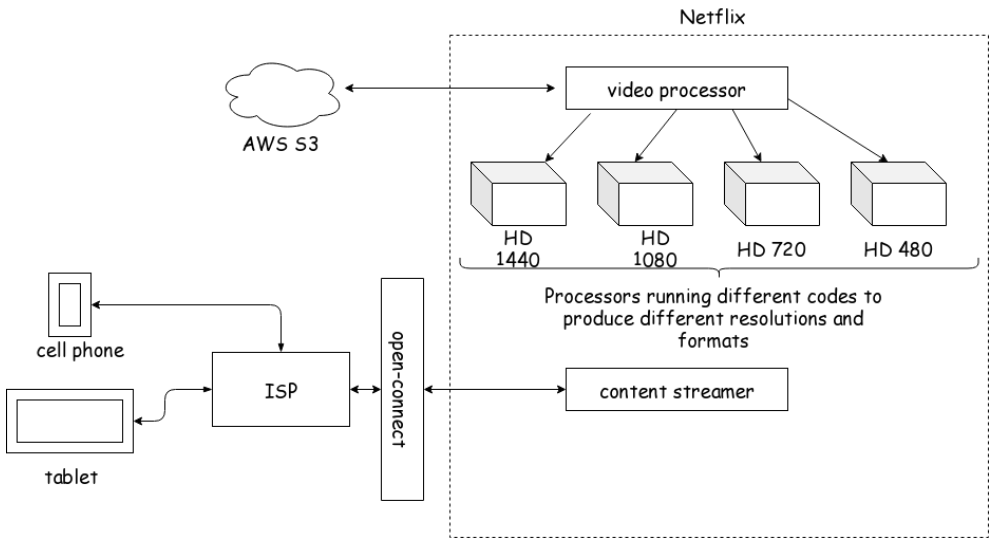

Figure 2.10: Functional Partitioning in Netflix as an example

how a netflix streaming service is divided into two major subsystems - the video processor and the content streamer. This brings a lot of benefits but also increasing engineering challenges. The main benefits are both service operate independently, parallel teams work on independent codebases. Each service scales independently of other.

- **Data partitioning**: Refers to maintaining subset of data onto each machine and controlling its state independent of other machines. This is the application of shared-nothing principle. With no data sharing, there is no data synchronization, no need for locking on global data store, and so failures can be isolated because nodes are not dependent on each other. A simplistic approach of data

partitioning is distributing objects among machines based on first english alphabets of lookup key which maps to actual machine. A more sophisticated technique will use consistent hashing to partition subset of data fairly among machines and keeps distribution fair when capacity is increased/decreased. Hence, data when partition correctly, provides endless scalability - adding more users, handle more parallel connections, collecting more data and deploy program onto more servers.

2.6.5 Managing State

Managing state is often overlooked aspect when addressing the scaling problem of the application. If not done properly, it can create barriers to scale the application well. In a program environment where identical copies are used to perform some computation, nodes using its own local/in-memory to cache objects are extremely difficult/tricky to coordinate cache invalidation. In such cases, it is best to maintain a global shared cache store so there is only one copy of each object and easier to invalidate.

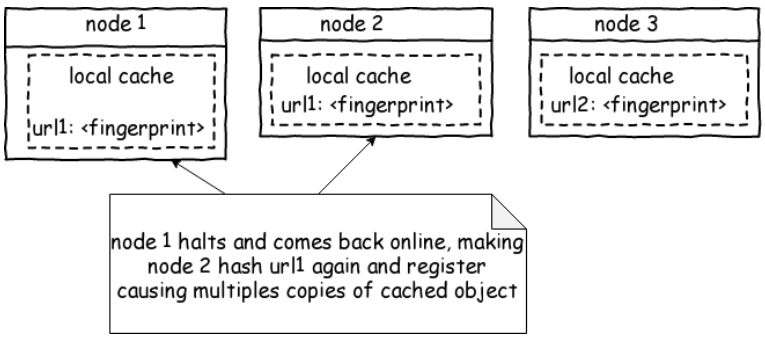

Figure 2.11: redundant copes of same cached object

Once you have a shared resource such as shared cache, for accessing them, locks are used to prevent race conditions and to synchronize access to shared resources. To achieve Horizontal Scalability(HS), distributed locks systems such as zookeeper [15] or chubby [16] can be considered. Many apps use local locks which causes bottlenecks. Instead of trying to share locks on web servers, you should push the state out of applns servers similar to http session data store.

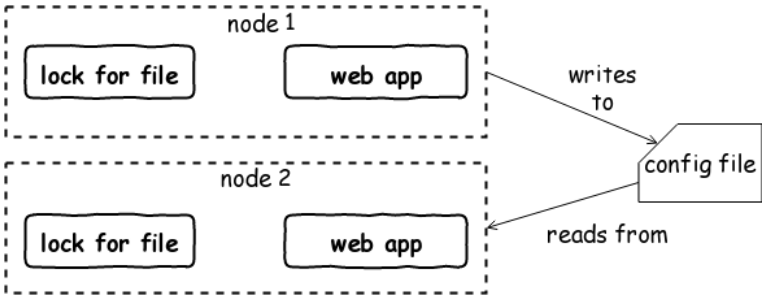

Figure 2.12: local locks preventing HS

To prevent the condition shown in figure 2.12, rule of thumb is to use a combination of functional partitioning and scaling out using clones. Remove the locking functionality from the application code and create an independent service from it. Then lets the program clones use shared lock service to share locks globally.

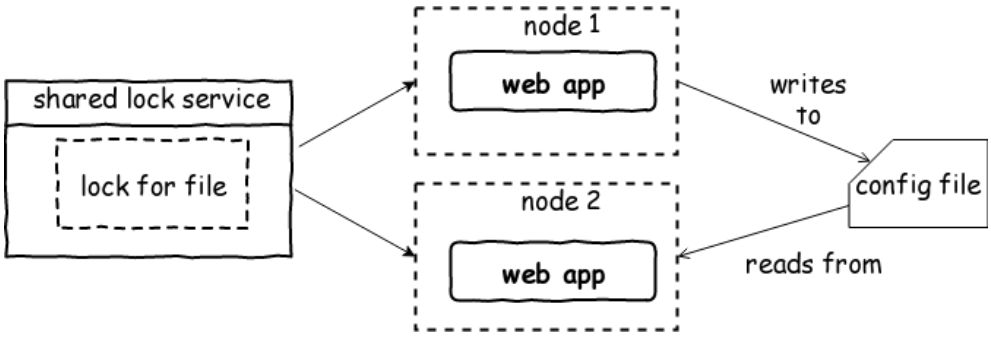

Figure 2.13: clones using shared locking mechanism

Potential disadvantage to using shared lock management is the increased latency. Locks can be easily implemented with memcached/redis but not as robust as Zookeeper, as it offers useful functionality for example - receiving notifications when a lock gets released.

2.7 Event-Driven Architecture

This style is different from traditional request/response programming model where the interaction between different components are accomplished by announcing events that have already occurred. An event does not hold any logic but contains a piece of data describing something has happened inside the application. Being able to identify parts of the application that fit into this model can positively impact on scalability.

2.7.1 Message Queues (MQ)

MQs are one such tool for achieving asynchronous behavior in the application even if the programming language in use does not support it.

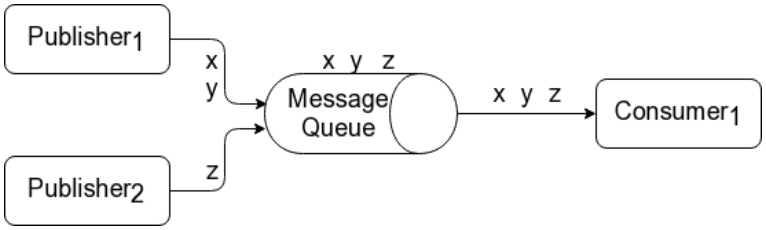

Figure 2.14: Message Queue with Producers and Consumers

MQ buffers messages and distributes it in asynchronous fashion. A message is XML/JSON data containing all information required to perform an operation. Requests served through MQs are always unidirectional, one-way, fire-and-forget style. Figure 2.14 shows MQ involves a producer/publisher on one side and consumer on other side. Producers push message to the queue which is buffered and finally delivered to active consumer.

decoupling
: message queues deliver the highest degree of decoupling. The biggest one is different language runtime.

scaled independently
: Producer and consumers can be horizontal scaled separately without overloading the system. The processes can also be hosted on completely separate machines.

balancing traffic
: Immediate advantage after scaling to separate machines is evening out spikes in capacity handling. This increases availability because messages produced are enqueued quickly before consumer can process it at its capacity.

fault-tolerance
: publishers and consumers are not affected by each others failure instances as they are not directly bound to each other but through intermediary such as queue, therefore, message processing can be stopped at any time for maintenance. Similarly, a hardware failure can be replaced with new machine without bringing the entire application to halt.

async processing
Producers and consumer work independently and constraint only by adhering to request format. This separation between producer and consumer enables non-blocking, asynchronous processing. Neither of them wait for each other to become available.

2.7.2 RabbitMQ as a Message Broker

A message queue can simply be seen as utility powered with SQL database. But there exist dedicated message broker which houses essential functionality such as message routing and delivery, permission control, and failure recovery. All this makes working with them easy. Brokers are sophisticated implementation of message queues. They are also called "message-oriented middleware".

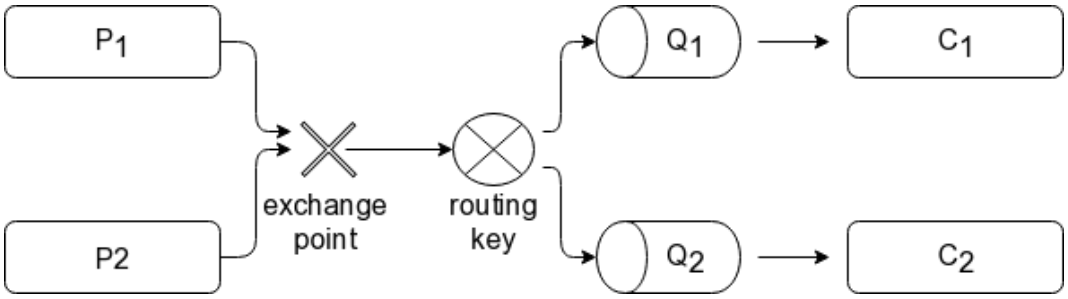

Figure 2.15: RabbitMQ broker terminology

RabbitMQ is a high-performance, generic message passing platform. This broker uses configuration over code to perform request routing and delivery. It exposes an API to dynamically configure routing. It speaks AMQP, STOMP messaging protocols, various client libraries exist to communicate with rabbitmq. The separation between publishers and consumers is accomplished through a concept called as exchange points as shown in figure 2.15. Producers P_1, P_2 has to known the name of the exchange only to enqueue the message in the right queue which is later consumed by consumer C_1, C_2. Moreover, RabbitMQ also supports complex routing through custom routing key patterns, along with built-in patterns like direct worker queue and pub/sub.

2.8 Amazon Web Services(AWS)

AWS is an on-demand cloud computing platform delivering technical infrastructure to customers through its web services. The customers are billed on a metered pay-as-you-go basis. Following is a short summary of list of services applied to run the whirlpool web crawler.

Identity & Access Management (IAM)
IAM is a free of charge access control service. Allows aws account holder to create and manage AWS users groups, and roles. Also enforce permissions on them to allow/deny access to AWS resources by attaching to it managed and custom policies.

Virtual Private Cloud (VPC)
VPC is a logically isolated virtual network environment that solutions architect provisions within which you can launch other AWS resources. A VPC is typically composed of public and private subnets. You can attach route tables to faciliate communication between different subnets and add a combination of stateful/stateless firewalls.

Internet Gateway (IGW)
IGW is a VPC component which is attached to the VPC that establishes connection between your running instances in the VPC and the internet. IGW supports IPv4 and IPv6 addresses.

Network Address Translation (NAT)
NAT component has its place in public subnet of the VPC to enable instances in the private subnet to initiate outbound traffic to the internet but deny inbound traffic initiated by external machine on the internet. There exist NAT gateway and NAT instance. Both serve same purpose but differ by features and how they are handled. Just to note, NAT gateway is managed service by AWS whereas NAT instance is managed entirely by entity using it.

Elastic Compute Cloud (EC2)
EC2 is most widely used, highly available compute capacity of AWS delivered through its own web service. Its the first service of AWS that changed the economics of rental computer hardware by allowing customers to pay only for what they used and how long they used.

AWS Monthly Calculator
This is a tool allowing customers to estimate monthly charges against their use of AWS services. This helps organizations identify areas where it can cut cost and plan according to their budget.

2.8.1 AWS Cloudformation

According to aws documentation [26], cloudformation(CF) is a service which lets you model and set up aws resources so that you spend less time managing resources and focus more on application logic that runs on those resources. Basically, in a template you describe all resources that make your architecture. You dont need to individually create and configure aws resources and figure what what's dependent on what.

some scenarios that demonstrate how aws cloudformation can help.

- simplify infrastructure management
- quickly replicate infrastructure
- easily control and track changes in your infrastructure

Some of the big concepts in CF are *Templates*, *Stacks*, and *ChangeSets*.

- **Templates:** CF uses templates saved in TXT, YML, or JSON as a blueprint for building AWS resources. The templates can be reused on altogether different regions by supplying input parameters at the time of stack creation.
- **Stacks:** CF manages related resources as a single unit called a stack. Creating, updating, and deleting a collection of resources is done by creating, updating, and deleting stacks. All the resources in a stack are defined by the specified CF template. To create resources which involves - Auto-Scaling group, Elastic Load Balancer(ELB), and an Amazon RDS database instance, create a stack by submitting the template consisting of those resources and CF provisions all those resources on your behalf. This requires a service role to be present before CF can actually create resources, refer to section 4.5 under whirlpool operations. To addon, Stacks can be operated using AWS CF console, APIs, and AWS CLI.
- **ChangeSets:** Making changes to the running resources in a stack, updates the stacks. Before changes are applied to the resources, a change set is generated which is a summary of proposed changes. ChangeSets makes the operator see how the his/her changes might impact current running resources, especially for critical resources, before implementing them. For example, if the name of the db identifier of Amazon RDS instance is changed, CF will create a new database and delete the one one. Data in the old database gets lost unless it was already backed up. If the given changeset is generated, operator will see that the change will cause database to be replaced and therefore the operator will be able to plan accordingly before update to a stack is applied.

2.9 Docker Containers

Containers are used to develop and build apps once, locally and run anywhere. Being flexible and lightweight, they are used to containerized complex programs such as this. Docker has its own terminology and concepts to be able to use it correctly. So a docker image is a executable package that includes everything required to run the program. A running instance of an image is called a container.

2.9.1 DockerFile

```
FROM python:3.7.4-buster as whirlpool-urlfilter-base

ENV PYTHONDONTWRITEBYTECODE=1
ARG WH_URLFILTER_ROOT=/home/whirlpool/whirlpool-urlfilter
WORKDIR $WH_URLFILTER_ROOT

RUN apt-get update \
  && apt-get install -y --no-install-recommends netcat \
  && rm -rf /var/lib/apt/lists/* \
  && useradd --create-home --shell /bin/bash whirlpool \
  && chown -R whirlpool:whirlpool $WH_URLFILTER_ROOT

# files necessary to build the project
COPY .pylintrc ./
COPY requirements.txt ./

RUN mkdir logs/ \
  && pip3 install -r requirements.txt

COPY scripts/ scripts/
COPY urlfilter/ urlfilter/

# docker image for dev target
FROM whirlpool-urlfilter-base as whirlpool-urlfilter-dev

COPY scripts/wait-for-it.sh scripts/wait-for-it.sh
ENTRYPOINT [”bash ./scripts/wait-for-it.sh”]

# docker image for prod target
FROM whirlpool-urlfilter-base as whirlpool-urlfilter-prod

COPY scripts/wait-for-it-prod.sh scripts/wait-for-it-prod.sh
ENTRYPOINT [”bash ./scripts/wait-for-it-prod.sh”]
```

This is a dockerfile for **whirlpool-parser** docker image, it defines what goes on in the environment inside your container.

2.9.2 Docker Compose

docker-compose is not bundled with docker utility and therefore installed separately. Compose is a tool for defining multi-container applications. For e.g, rest api, database server, redis cache are the services that make up application stack and can be launched with single compose command.

```
version: '3'
services:
  rest-api:
    build: .
    ports:
    - "5000:5000"
    volumes:
    - .:/code
    - logvolume01:/var/log
    links:
    - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

3 | Implementating Whirlpool

3.1 Mission

Domain	Okay to crawl ?
CareerBuilder	yes
indeed.com	yes(restrictive)
job.com	yes
ladders	yes
linkedIn	No
glassdoor	yes(restrictive)
monster	yes(restrictive)
simplyhired	yes
us.jobs	yes
dice.com	yes(restrictive)
idealist.com	yes

Figure 3.1: Topical crawling seed set

3.2 Characteristics

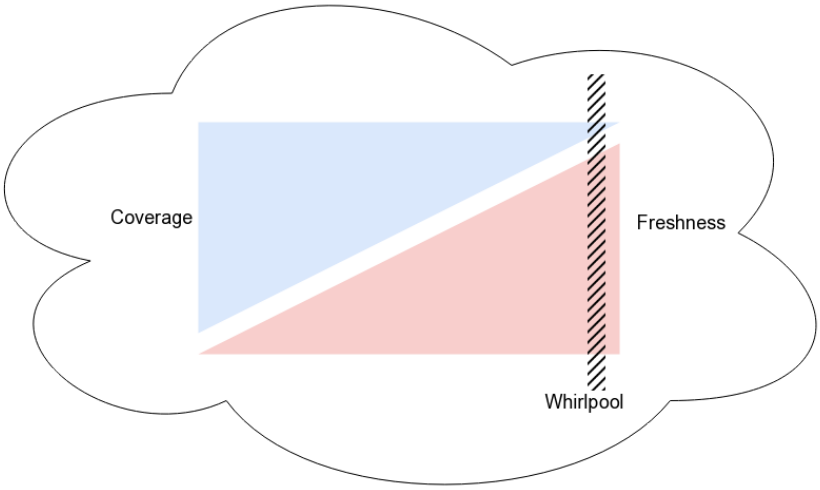

Figure 3.2: Crawl order policy of Whirlpool

3.3 Internals

3.3.1 RabbitMQ as a Central Message Bus

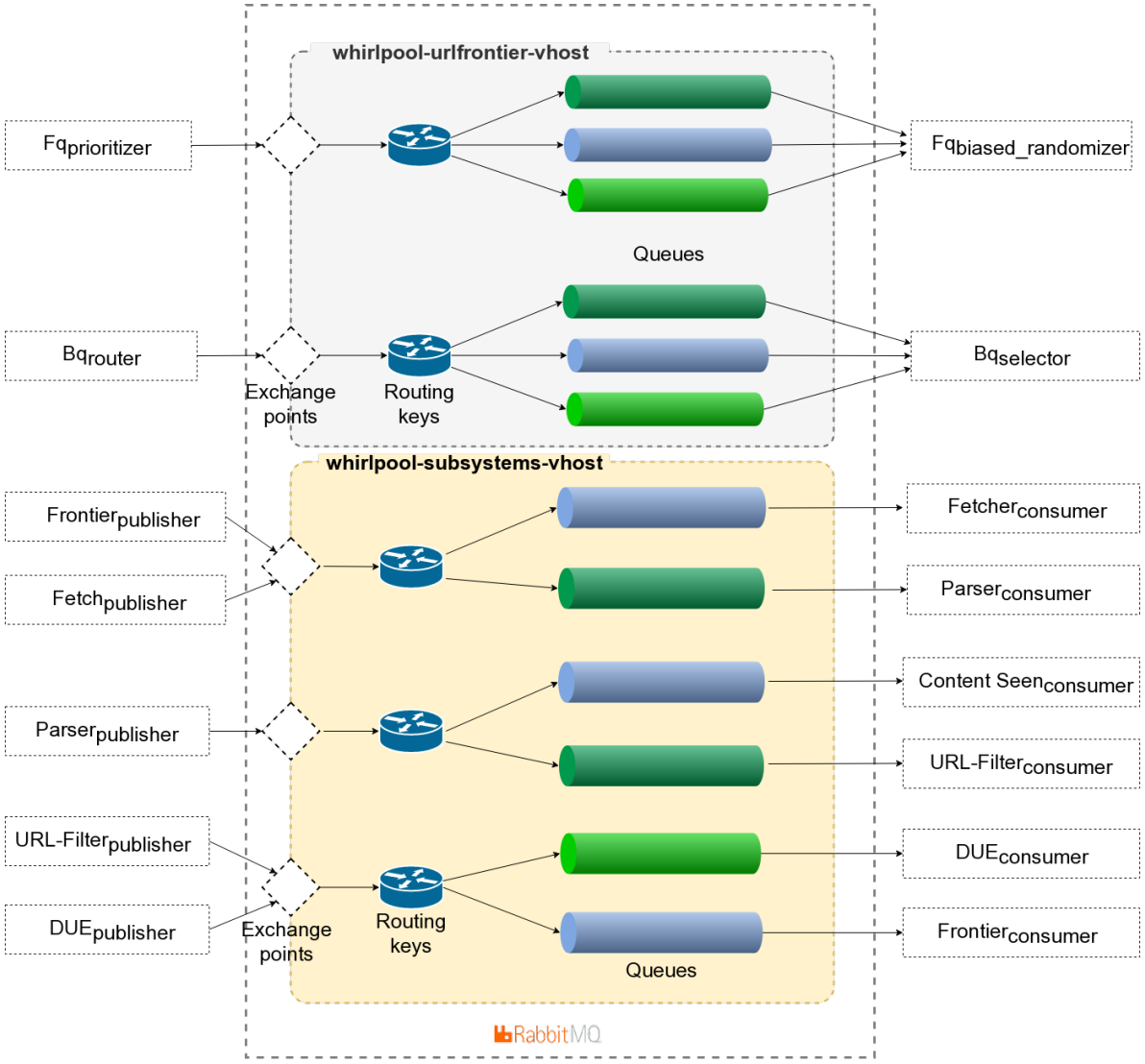

Figure 3.3: Message Distribution using RabbitMQ Direct Exchange

The initial rabbitmq configuration is shown in the **`https://github.com/rihbyne/whirlpool-rmq/blob/master/setup.j`**. The subsystems only need to know the exchange points and submit message payload to it. The exchange points take care of routing payload to its recipient queue. The payload is static, and doesn't contain any business logic, this leaves the subsystems highly decoupled from each other.

3.3.2 Shared Services with Docker

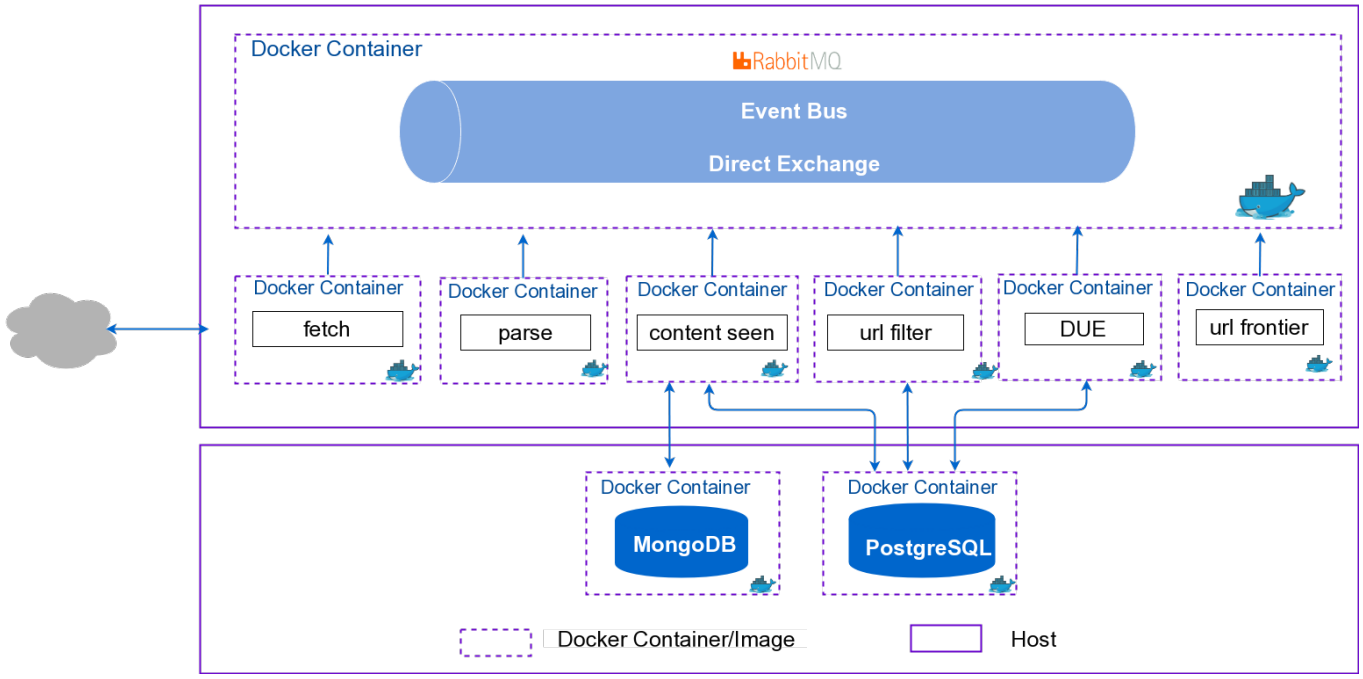

Figure 3.4: Single-node Whirlpool Subsystem Isolation using Docker

3.3.3 Developing code with Docker

At the foundation of any dockerized program, **dockerfile** is a place to package source code. The biggest advantage of using containerization as discussed in section 2.9, is to get same program behavior under development and production environments, making deployment process easier. At first, a developer will use **dockerfile** for developmental use. This approach, however, is time-consuming, slows speedy development as docker engine takes a while to build & re-run on every change. Whats even worse is that the size of development image gets bigger on each build as dependencies are added/removed, updates to the base images are installed. This actions aren't required when developing code.

The best practice when developing code with docker is to use **docker-compose.yml** to define the environment at different layers. The version 2 of compose file allows to extend and reuse existing layers. Version 2 is more developer friendly while version 3 is geared towards production use. As you can in below image under **services** directive - install and quick-up project services extend base project.

```
version: '2.4'

networks:
  default:
    external:
      name: whirlpool-net

services:

  base:
    image: node:10.16.0
    command: bash -c "useradd --create-home --shell /bin/bash whirlpool && chown -R
    ↪  whirlpool:whirlpool /home/whirlpool/whirlpool-fetcher"
    volumes:
      - .:/home/whirlpool/whirlpool-fetcher
      - wh-fetch:/home/whirlpool/whirlpool-fetcher/node_modules
    working_dir: /home/whirlpool/whirlpool-fetcher
    environment:
      - NODE_ENV=development
    networks:
      - default

  install:
    extends:
      service: base
    command: npm install --no-audit

  quick-up:
    extends:
      service: base
    command: npm start
```

```
31
32 volumes:
33   wh-fetch:
34     external: true
```

The environment defines external **network** sharing it with containers defined outside and inside the compose file. This is specific to development setup where one component is actively getting developed while others are in ready state. Similarly, the external **volume** persist javascript(in this example) dependencies outside the container, thus allowing sharing among other containers such as install and quick-up services.

```
1  install:
2    docker-compose -f docker-compose.build.yml run --rm install
3
4  quick-up:
5    docker-compose -f docker-compose.build.yml run --rm quick-up
6
7  prod-build:
8    docker build -t whirlpool-fetch-prod:latest --target whirlpool-fetch-prod .
9
10 tag-prod:
11   docker tag whirlpool-fetch-prod:latest rihbyne/whirlpool-fetch-prod:latest
12
13 push-prod:
14   docker push rihbyne/whirlpool-fetch-prod:latest
```

The use of **Makefile** makes docker commands easier to remember when typing on the command line. Note that the same commands can also be integrated into IDE of choice.

```
$ make install
```

Trying the above command for the very first time will establish network, pull node image only once and install dependencies to the specified external volume. This is for the project defined in **docker-compose.build.yml**.

```
$ make quick-up
```

Once the packages are installed, and after having made some changes to the code. The program is launched using the above command. It runs inside the container with installed packages shared by base servive. The flow is pretty fast as the install and run operations are isolated and there is no time wasted in building and packaging on every iteration of code change. This is very much ideal and correct way to use docker while writing code.

```
$ make prod-build && make tag-prod && make push-prod
```

This command clubs 3 commands - packages code for running on production machine, labels the image and pushes it to a central docker repository.

3.3.4 Policy of assigning priorities & picking queues

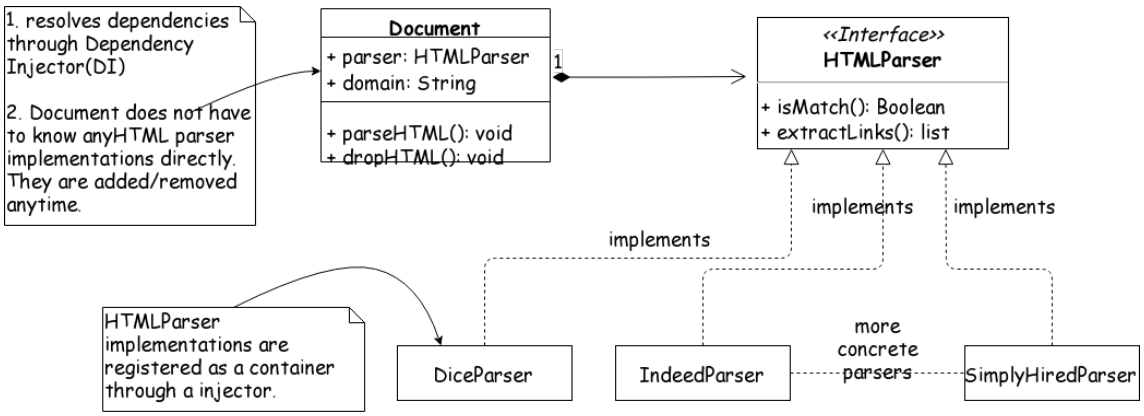

Figure 3.5: Whirlpool Parser subsystem using Dependency Injection

3.3.5 URL Frontier

3.3.5.1 Prioritizer Policy

3.3.5.2 Front Queue Biased Selection Policy

3.4 Distributed crawling(splitting by key range vs. hash of key)

This section presents strategies to distribute crawling activity across many nodes. Each participating node is given the responsibility of downloading a slice of seed set. The main reason to distribute crawling is to make it scalable.

Mercator's[8](1999) host splitter component distributes load by hostname. Limitations of this way of distribution and alternatives are discussed below. Ubicrawler[9](2004) achieved linear scalability, fault tolerance through consistent hashing[25]. Both designs mention some form of load balancing however papers fall short of explanation on technicalities surrounding it.

When the goal is to distribute load evenly across finite number of nodes, say, 5 nodes should handle 5 times the throughput of a single node. One way to split the load is to assign a range of request boundaries from minimum to maximum to each node. Examples of load distribution based on range of keys can be hostnames, alphabets A-Z, etc. This works as long as there exist calculated risk of every node getting a fair share of the load but in many cases, load is unbalanced which causes one crawler node to compute more data than others causing skewed workloads[25]. In extreme cases, only a specific boundary(e.g A-D) assigned to a node is taking all the load while other nodes responsible for handling boundary say X-Z, sits idle, such high disproportionate load becomes a hot spot[25] in distributed systems terminology.

To overcome problems encountered above with partitioning load by key range is to use hash function such that output of hash of a key is an integer which maps to a position somewhere on the line across the range of numbers, shown in figure 3.6. A hash function with low collision probability can turn a skewed load into uniformly distributed load. Each physical machine/node pn is hashed to a random integer between 0 to $2^N - 1$ where $N = 16$ using a 16-bit hash function that serves range of hash values falling within its request boundaries RB_i. Figure 3.6 shows RB_i can be psuedorandomly or evenly spaced to fairly distribute load across pn_i.

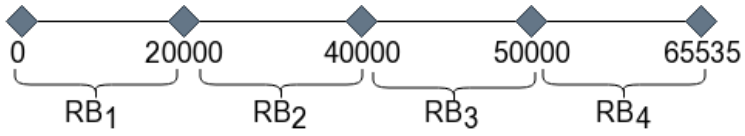

Figure 3.6: partitioning by hash of key

Certain attributes of a machine can be used as a input key to a hash function mapping to a integer on the line figure 3.7. From left to right, when hash of a absolute URL(e.g at 23000) used as a key falls within a nodes range, task request is fulfilled by that node(at 40000). This way of distributing the load evenly across machine is called consistent hashing[25]. Hashing reduces skewed workloads and hot spots but they cannot be 100 percent eliminated.

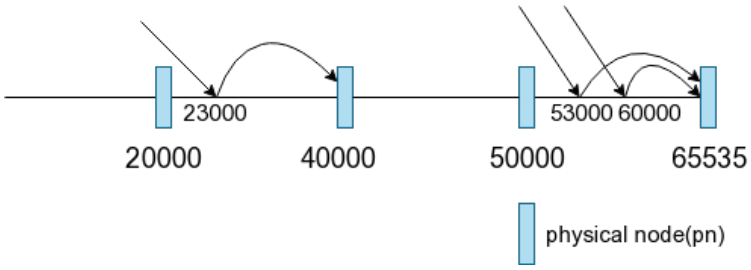

Figure 3.7: Rebalancing physical nodes by mod N

So far the assumption is parallel crawling will always be performed on a fixed number of pn which is quite unlikely. Overtime, a crawler's seed set may expand or shrink or altogether change its characteristics(see figure 3.2 requiring more CPU to cope up with or a node can crash for some reason(e.g RAM overflow). All these changes mean moving messages between nodes. Once rebalancing is successfull, crawling load is distributed fairly among available nodes.

absolute URL	m = hash(absolute URL)	m mod N(1)	m mod N(2)	m mod N(3)	m mod N(4)
jobs.com/browses	24	n1	n1 0	n1 0	n1 0
dice.com/browse-titles	59956	n1	n2 1	n1 1	n4 1
monster.com/categories	10090	n1	n1 0	n1 0	n3 1
simplyhired.com/find-jobs	23000	n1	n2 1	n2 0	n4 1
indeed.com/jobs/unix-admin	41103	n1	n2 1	n1 1	n4 1
glassdoor.com/hires/software-developer	35114	n1	n1 0	n3 1	n3 0
data movement		0%	50%	50%	60%

Figure 3.8: data movement when using modulo N

The easy strategy to rebalance request boundaries assigned to a node pn_i is through $ph_i \equiv h(url) \mod N$, where N is total number of available pn_i. For e.g, $h(url) \mod 5$ would direct service request to any pnodes between $0 < pn_i < 4$. Figure 3.8 shows a troubling trend with mod N where rebalancing of N nodes causes half of service requests to redirect to another nodes. Every addition of new node pn_i has 50% shift in service request, same happens in reverse. Note that the crawler program implemented for this thesis keeps state of URLs local to its message queue, therefore making rebalancing with mod N an expensive operation.

The problem is countered by iterating each pn_i through few different hash functions and marking its positions on the line as shown in figure 3.9. The values obtained are called virtual nodes(vnodes) vn_i which are just essentially integers of different hash functions(in this case - h_2, h_3, h_4) and not virtualboxes running on real hardware. A rule of thumb is to have more number of vn than there are pn so that several vn are assigned to pn. For e.g a cluster of 3 pnodes has 12 vnodes where 4 vnodes are assigned to each pnode.

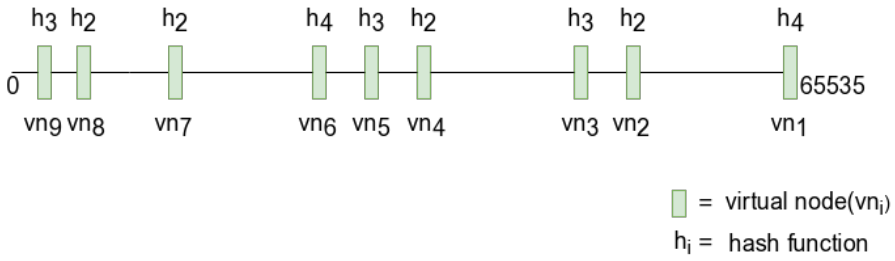

Figure 3.9: virtual nodes to reduce shifts in data movements between nodes

Once vnodes have positions on the line, pnodes are finally placed on the line with hash function quite different from that used for respective vnodes. Figure 3.10 shows 4 physical nodes, each running an instance of rabbitMQ with crawler subsystems bound to them. This works differently from previous modulo N strategy. When a request comes in and its hash(44000) falls somewhere near to vn_3 who's hash value is 46000, it is consumed by pnode pn_1 at hash value 60000. Similarly, a hash value of 10 is near to vn_9 is served by pn_4. When a new pnode is added to the existing cluster, fewer vnodes from every pnode pn_i in the cluster get reassigned to newly added pnode and url is consumed by first pnode available on the line, from left to right. The same thing happens in reverse when node is being removed from the cluster. The criteria for load distribution can be, more powerful pnodes can take greater share of the load meaning assigning more vnodes vn for that particular pnodes pn.

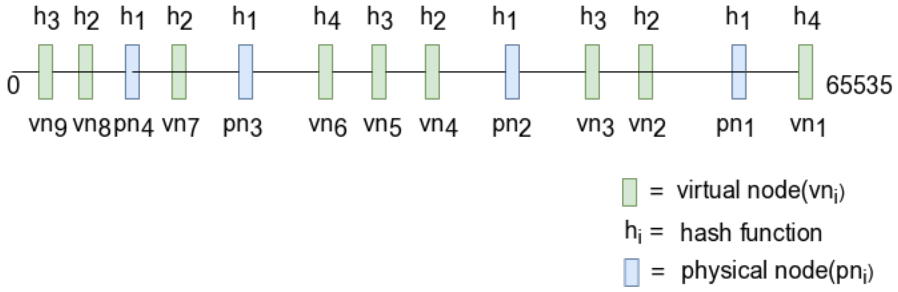

Figure 3.10: Rebalancing physical nodes with virtual nodes

In order for crawler subsystems to know whether a current url is for itself or other pnodes in the cluster (pnodes are actually crawler subsystems), even when pnodes are rebalanced and assignment of vnodes to pnodes change, there exist a service discovery tool which keeps up-to-date information on ip address and port number of pnodes in cluster and can thereby direct url request to the right pnode. The host splitter makes the routing decision and learns about changes in the assignment of vnodes to pnodes. It does so through zookeeper[15] which is a coordination service for keeping metadata about pnodes cluster.

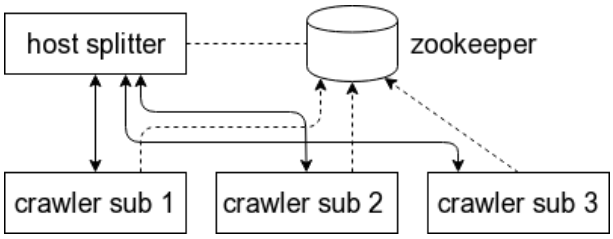

Figure 3.11: zookeeper used to maintain up-to-date information on which vnodes are assigned to pnodes

Each pnode pn when available, registers itself with zookeeper. The zookeeper table in figure 3.12 maintains mapping between vnodes vn to pnodes pn. Host splitter subscribes to zookeeper server. When a pnode is added or removed and vnodes are reassigned, zookeeper notifies host splitter component.

hash range	vnode(vni)	pnode(pni)	IP
0 - 3k	vn9	pn4	10.0.0.4
3k - 6k	vn8	pn4	10.0.0.4
6k - 10k	vn7	pn3	10.0.0.3
10k - 15k	vn6	pn2	10.0.0.2
15k - 20k	vn5	pn2	10.0.0.2
20k - 35k	vn4	pn2	10.0.0.2
35k - 45k	vn3	pn1	10.0.0.1
45k - 55k	vn2	pn1	10.0.0.1
55k - 65k	vn1	pn4	10.0.0.4

Figure 3.12: zookeeper table

4 | Whirlpool Operations

This chapter discusses server farm assembly for running whirlpool on a on-demand hardware resources offered by AWS. It uses altitude analogy to illustrate important additions in infrastructure at various levels in detail. The last section explains hardware configuration of resources used and their pricing details.

4.1 Infrastructure from 10,000 feet

Given a working knowledge of AWS and behavior of application that will run on top of its resources, it becomes much easier to combine different tools to architect AWS solution. For this project, identical copies of crawler application are deployed onto more than one machine. Special attention is paid to managing the state of the crawler. Combining understanding of theory discussed in section 2.6.5, intuition, and experience, the program maintains a global datastore(ContentDB) shared across all crawler processes. A running crawler process dumps downloaded data since the scope of the crawler for this thesis is confined to only data-collection. Also, the message broker RabbitMQ which forms the communication backbone is stateful(local) to a crawler process. So spinning each new crawler node will have its own RMQ bound to it. The internal metadata that crawler maintains which leverages a relational database and in-memory cache is also shared across crawler nodes to overcome scalability challenges and issues addressed in the theory of managing state in section 2.6.5. Lastly, the crawler program which forms the core business logic of this thesis shouldn't be part of DMZ zone.

Keeping above paragraph in mind, figure 4.1 shows a AWS assembly containing four subnets numbered 1-4 within a VPC. The Internet Gateway(IGW) is active and attached to this VPC. Public subnet 1 is explicitly associated with route table 1 which consist of route rule that directs traffic from subnet 1 to the outer public internet. The data transfer in crawling process is such that it will make connection requests to the web sites and pull data into the amazon cloud, thus the inbound data transfer cost from the internet into amazon cloud is free. Security-wise, the crawler nodes should never except incoming connections from internet. These nodes are placed in private subnet 4. By default a private subnet within a AWS VPC is accessible to/from other subnets only within its VPC CIDR block and is denied any inbound/outbound connections to the internet. As a best practice, subnet 4 is allowed only inbound connections. This is done by associating subnet 4 with route table 2. A route in table 2 directs internet traffic to NAT instance deployed in public subnet 1 which further takes from Internet Gateway(IGW) attached to the VPC all the way into public internet.

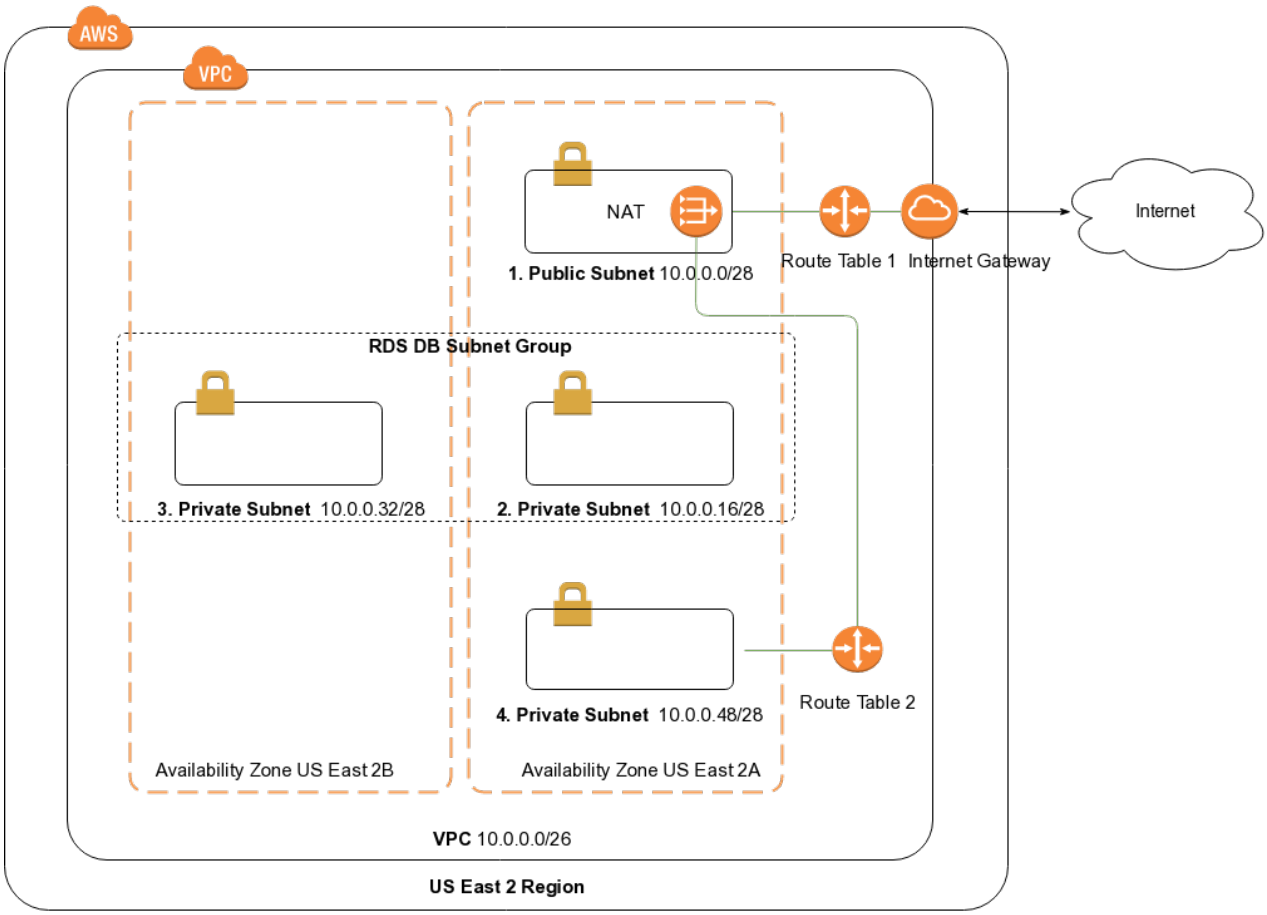

Figure 4.1: Whirlpool Infrastructure using VPC, AZ, and Subnet Sizing

The VPC in figure 4.1 uses Classless Inter-Domain Routing(CIDR) block of **`10.0.0.0/26`** which gives a range of **64** IPv4 addresses. This thesis uses a tool [19] to calculate CIDR blocks. AWS VPC reserves first four and last IPv4 addresses of each subnet created for internal usage and therefore cannot be assigned to any instance of any resource. The primary CIDR block size of VPC is then used to create 4 subnets each with a non-overlapping CIDR block size. The minimum subnet size allowed is **28** and **16**.

CIDR block size is given by $2^{(32-x)}$, substituting $x = 26$ yields,

$$2^{(32-26)} = 64 \qquad \text{IP addresses}$$

$$\text{4 subnets formation} = \underbrace{16}_{\text{public subnet}} + \overbrace{\underbrace{16}_{\text{private db subnet 1}} + \underbrace{16}_{\text{private db subnet 2}}}^{\text{DB Subnet Group}} + \underbrace{16}_{\text{private crawler subnet}}$$

$$\text{Actual IP addresses available} = \underbrace{11}_{\text{public subnet}} + \overbrace{\underbrace{11}_{\text{private db subnet 1}} + \underbrace{11}_{\text{private db subnet 2}}}^{\text{DB Subnet Group}} + \underbrace{11}_{\text{private crawler subnet}}$$

Public subnet IP range = **`10.0.0.0/28`** → (0 - 15)
Private db subnet 1 IP range = **`10.0.0.16/28`** → (16 - 31)
Private db subnet 2 IP range = **`10.0.0.32/28`** → (32 - 47)
Private crawler subnet IP range = **`10.0.0.48/28`** → (48 - 64)

4.2 Infrastructure from 5,000 feet

Figure 4.2 shows the final assembly of whirlpool crawler project.

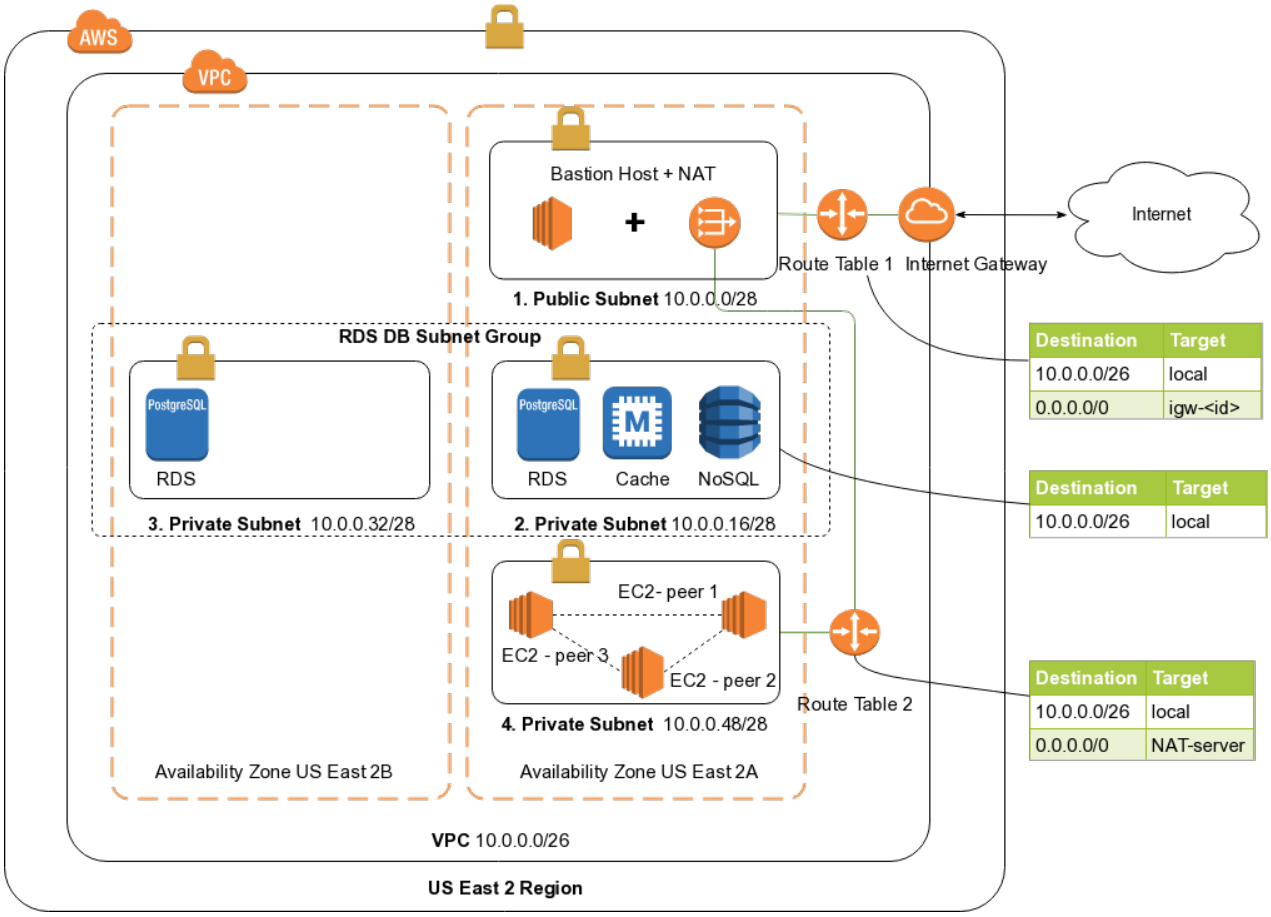

Figure 4.2: Whirlpool infrastructure with Route tables, NAT

Each subnet gets assigned a main route tables which cannot be deleted but instead can be overridden with custom route tables immediately effective on subnets as evident in figure 4.2 with route tables 1 & 2. Speaking of NAT instance in public subnet 1, is like any other AMI linux image but ships with pre-configured iptables. NAT forwards traffic from instances in subnet 4 to the internet and send the response for corresponding request back to those instances in subnet 4. It won't allow outside clients to initiate connections with instances in subnet 4. The custom route(shown in route table 2) directs the traffic originated within any of the private subnet peers matching subnet mask **`0.0.0.0/0`** to NAT server. The custom route(show in route table 1) scopes all packets matching **`0.0.0.0/0`** route to IGW. The traffic from private crawler subnet flows to NAT Instance and then to the IGW. The NAT translate back-and-forth source and destination IPs of private instances assuming external property - source/destination check is disabled.

The subsystems of whirlpool leverage relational database(Amazon RDS) to maintain a history of URLs, NoSQL(EC2-mongoDB) to store extracted text. AWS does not have managed instance of MongoDB and requires the interested party to operate, maintain on an EC2 Instance. Also, later on, a cache server(AWS ElasticCache) can be adopted

to optimize lookup efficiency against relational database by those subsystems dependent on it. Given this requirement, its safer to form a private subnet 2 dedicated to placement of data stores and in-memory cache as shown in the figure 4.2. This way, the data store subnet would stay safer accepting network connections within the VPC, specifically only from instances allowed by statefull firewalls(VPC security groups) attached to it. For relational database, amazon's RDS DB subnet group mandates 2 subnets, each in different availability zones to successfully launch the instance. Thus, a private subnet 3 in the figure.

Following are a list of VPC security groups in place for figure 4.2 which regulate in/out flow of traffic attached to EC2 instances, elasticCache, and RDS instance respectively.

Type	**Protocol**	**Port Range**	**Source**	**Description**
SSH	TCP	22	sg-<bastion node>	only allows ssh to crawler nodes from bastion instance
Custom	TCP	8080	sg-<bastion node>	access RMQ management console access ssh tunneled to client host

Table 4.1: Inbound rules of Security Group(sg) for EC2 crawler in private subnet 4

Type	**Protocol**	**Port Range**	**Source**	**Description**
SSH	TCP	22	sg-<bastion node>	ssh only from bastion instance
Custom	TCP	27017	sg-<crawler node>	mongo traffic from crawler nodes
Custom	TCP	27017	sg-<bastion node>	egress mongo traffic tunneled to client host

Table 4.2: Inbound rules of Security Group(sg) for MongoDB in private subnet 2

Type	**Protocol**	**Port Range**	**Source**	**Description**
SSH	TCP	22	0.0.0.0/0	open to the world. Safely handle pvt. keys.
HTTP	TCP	80	sg-<crawler node>	accept non-ssl http traffic from crawler node
HTTPS	TCP	443	sg-<crawler node>	crawler downloads SSL HTTP traffic

Table 4.3: Inbound rules of Security Group(sg) for bastion host in pub subnet 1

Type	Protocol	Port Range	Source	Description
Custom	TCP	5432	sg-<crawler node>	accept sql request from crawler node
Custom	TCP	5432	sg-<bastion node>	egress rds tunneled to client host

Table 4.4: Inbound rules of Security Group(sg) for AWS RDS in pvt subnet 2 and 3

Type	Protocol	Port Range	Source	Description
Custom	TCP	11211	sg-<crawler node>	allow memcache traffic from crawler nodes
Custom	TCP	11211	sg-<bastion node>	egress cache traffic tunneled to client host

Table 4.5: Inbound rules of Security Group(sg) for AWS ElastiCache in pvt subnet 2

4.3 Bastion Host

As per the wikipedia arcticle, a person named Marcus J. Ranum defined the term bastion host while discussing a article on firewalls as

> "...a system identified by the firewall administrator as a critical strong point in the network security. Generally, bastion hosts will have some degree of extra attention paid to their security, may undergo regular audits, and may have modified software."

Not necessarily in the context of AWS but also in data centers of IT organization, a baston host sole purpose is to provide access to private network from external network such as internet. In a typical AWS VPC setup, an instance in a public subnet with a security group that includes a inbound rule to accept SSH traffic from trusted client host establishes secure remote connectivity after which the instance(bastion host) acts a jumping point to ssh into various private instances belonging to various private subnets within its VPC. With bastion host in place, a administrator uses ssh-agent forwarding to authenticate to private instances. Agent forwarding doesn't require administrator to store private keys on the bastion host and AWS best practices forbids storing private key files on servers. Figure 4.2 demonstrates a special network configuration where bastion host is placed in DMZ zone separated from private, trusted networks such as subnets 2, 3, and 4. Another benefit bastion hosts provide is not having to expose various management ports to internet to configure utility/application services on private instances of the infrastructure.

Agent forwarding is enabled by passing below flag when login into a remote node.

```
$ ssh -A user@ip
```

Since whirlpool assembly contains multiple networks, following ssh aliases in .ssh/-config saves from typing.

```
# access to whirlpool bastion and its private nodes
Host whirlpool-bastion
   HostName ec2-x-x-x-x.us-east-2.compute.amazonaws.com
   User ec2-user
   IdentityFile ~/.ssh/whirlpool-jumbox.pem
   ProxyCommand none

Host whirlpool-crawler-1
   HostName ip-x-x-x-x.us-east-2.compute.internal
   User ubuntu
   IdentityFile ~/.ssh/whirlpool-jumbox.pem
   ProxyCommand ssh whirlpool-bastion -W %h:%p

Host whirlpool-mongodb
   HostName ip-x-x-x-x.us-east-2.compute.internal
   User ubuntu
   IdentityFile ~/.ssh/whirlpool-jumbox.pem
   ProxyCommand ssh whirlpool-bastion -W %h:%p
```

4.4 Monitoring services with SSH Tunnels

This project steals concepts behind ssh tunnels [14] and applies them to its own AWS architecture, figure 4.2, to keep track of activities happening inside it. SSH tunnels are nicely explained by Scott Wiersdorf in his article; the source is listed in references. Image 4.3 shows a **`client-host`** connecting to an example **`web-server`** on port 80 tunneled through **`tunnel-host`** using ssh utility.

Figure 4.3: simple tunnel [14]

SSH tunnels involves two hosts - **`client-host`**, and **`tunnel-host`**. The **`client-host`** specifies the tunnel to be created. On the other hand, **`tunnel-host`** creates the tunnel connections. Both the hosts are defined as single bash command. The traffic between **`client-host`** to **`tunnel-host`** is secured by secure shell. All the TCP traffic past the **`tunnel-host`** is not secured by SSH. The connections in SSH tunnels are initiated at only one end and so the listening end of the tunnel is referred to as tunnel ingress and terminating end of the tunnel is referred to as tunnel egress as illustrated in the image 4.4.

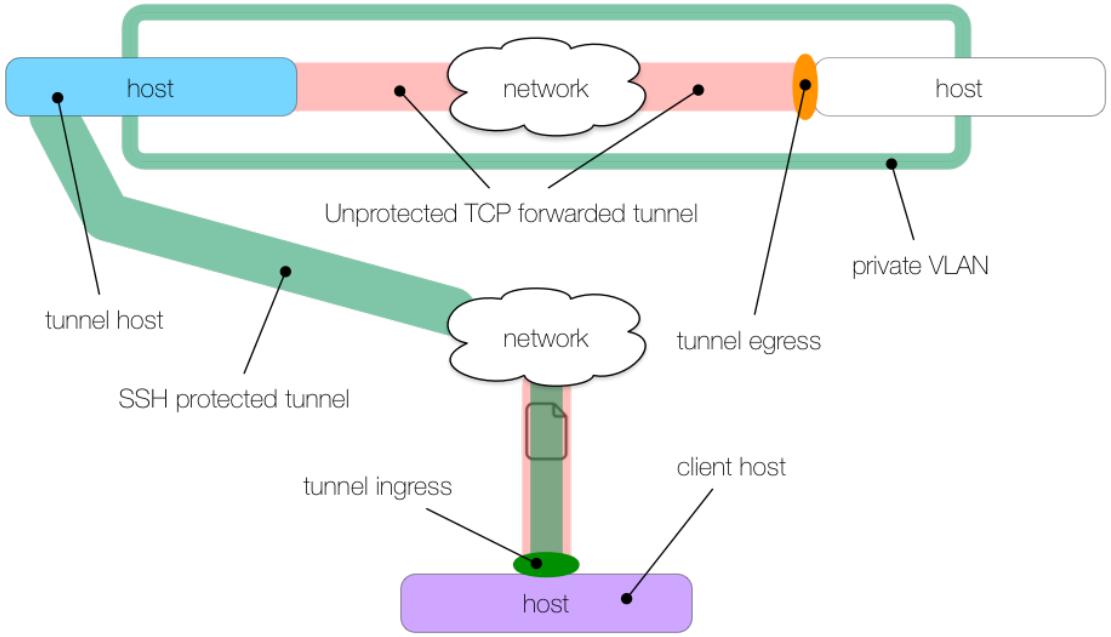

Figure 4.4: poor man's VPN [14]

In the case of whirlpool network topology, **client-host** is the machine used to ssh into **bastion-host**. The **tunnel-host** is the **bastion-host**(NAT instance) and this is where the SSH connection ends. Bastion-host forwards TCP connection to a specified private instances within whirlpool AWS deployment because SSH agent-forwarding is enabled, discussed in previous section. The specified private instance is the egress and ingress is the **client-host** bound to a particular port.

Following are the commands fired to monitor, diagnose services inside AWS. Note the egress **10.0.0.56:8080** and ingress is an omitted **localhost** on port **8080**. The bastion host **whirlpool-bastion** forwards TCP to egress. **N** indicates no commands over SSH. This command is used for monitoring RabbitMQ server running on **10.0.0.56:8080**. The corresponding firewall rules is enabled for the same.

```
$ ssh -L 8080:10.0.0.56:8080 whirlpool-bastion -N
```

This command is used for making private ec2-mongodb instance available on **localhost**.

```
$ ssh -L 27017:10.0.0.26:27017 whirlpool-bastion -N
```

This command is used for making private RDS instance available on **localhost**.

```
$ ssh -L 5432:whirlpool-postgres-prod.ytrewqgfdsa.us-east-2.rds.
    amazonaws.com:5432 whirlpool-bastion -N
```

This command is used for making private memcache instance available on **localhost**.

```
$ ssh -L 11211:whirlpool-cache.qwerty.0001.use2.cache.amazonaws.com
    :11211 whirlpool-bastion -N
```

4.5 IAM and roles

Showcase few IAM policies related to whirlpool project here.

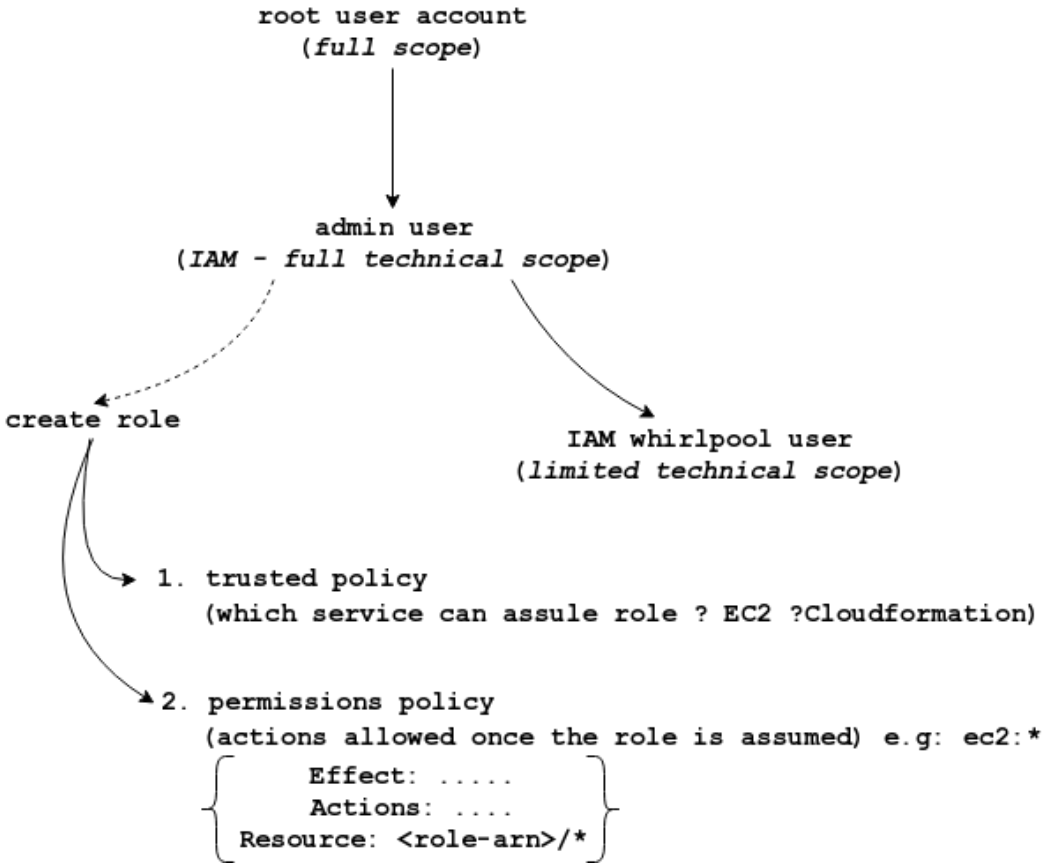

Figure 4.5: Existing accounts and service role

This project is run under IAM whirlpool user with limited technical privileges by enforcing IAM policies, delegated by IAM administrator. The IAM administrator has full technical access but not billing access granted by AWS root user account. A service role exist because AWS cloudformation as a service provisions other services on behalf of whirlpool user by assuming a service role. Use of service role in context of this project is shown in figure 4.5.

A service role contains a trust policy and permissions policy. Existing iam policies are added as permissions policy under 'actions'. Under 'resource/principal' property, cloudformation which assumes role is specified as a trust policy. A restricted IAM whirlpooluser doesn't need to be given permissions to create/modify/delete roles instead only list/describe/use roles. Note, this works only if IAM policies attached to whirlpool IAM account has **`IAM:passrole`** action included.

Following are the restrictive, custom, managed IAM policies effective under whirlpool IAM account.

```
- whirlpool-user-iam-cache-vpc-policy.json
- whirlpool-user-iam-cloudformation-policy.json
- whirlpool-user-iam-cloudformation-role.json
- whirlpool-user-iam-cloudformation-s3-policy.json
- whirlpool-user-iam-ec2-vpc-policy.json
- whirlpool-user-iam-policy.json
- whirlpool-user-iam-rds-vpc-policy.json
```

Below is the code snippet of **whirlpool-user-iam-ec2-vpc-policy.json**, one of existing IAM policies attached to IAM whirlpool account.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ec2:RunInstances",
                "ec2:RebootInstances",
                "ec2:StopInstances",
                "ec2:StartInstances",
                "ec2:TerminateInstances",
                "ec2:Describe*",
                "ec2:AttachVolume",
                "ec2:DetachVolume",
                "ec2:DeleteVolume"
            ],
            "Resource": [
                "arn:aws:ec2:us-east-2:612113937920:volume/*",
                "arn:aws:ec2:us-east-2:612113937920:instance/*",
                "arn:aws:ec2:us-east-2::image/*",
                "arn:aws:ec2:us-east-2:612113937920:network-interface/*",
                "arn:aws:ec2:us-east-2:612113937920:key-pair/*",
                "arn:aws:ec2:us-east-2:612113937920:security-group/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:*Route*",
                "ec2:DeleteNetworkAcl",
                "ec2:DeleteNetworkAclEntry",
                "ec2:DeleteRoute",
                "ec2:DeleteRouteTable",
                "ec2:AuthorizeSecurityGroupEgress",
                "ec2:AuthorizeSecurityGroupIngress",
```

```
                "ec2:RevokeSecurityGroupEgress",
                "ec2:RevokeSecurityGroupIngress",
                "ec2:DeleteSecurityGroup"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "ec2:vpc":
                    ↪ "arn:aws:ec2:us-east-2:612113937920:vpc/vpc-0c7a0ba5e8cad5b83"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": "ec2:RunInstances",
            "Resource": "arn:aws:ec2:us-east-2:612113937920:subnet/*",
            "Condition": {
                "StringEquals": {
                    "ec2:Vpc":
                    ↪ "arn:aws:ec2:us-east-2:612113937920:vpc/vpc-0c7a0ba5e8cad5b83"
                }
            }
        },
        {
            "Action": [
              "ec2:Describe*",
              "ec2:*KeyPair*",
              "ec2:*Nat*",
              "ec2:*Gateway*",
              "ec2:*subnet*",
              "ec2:CreateSecurityGroup",
              "ec2:CreateVolume",
              "ec2:ModifyVolume",
              "ec2:createTags",
              "ec2:DeleteTags",
              "ec2:CreateRouteTable",
              "ec2:ReplaceRouteTableAssociation"
            ],
            "Effect": "Allow",
            "Resource": "*"
        }
    ]
}
```

4.6 AWS Services Hardware Specfication & Cost Estimation

AWS resource specification

US East (Ohio) Region

Compute: Amazon EC2 Instances:

Description	Instances	Usage	Type	Billing Option
Ccrawler Node	3	50% utilization/month	Linux on t2.mciro*	On-Demand (No contract)
Content DB (MongoDB)	1	50% utilization/month	Linux on t2.mciro	On-Demand (No contract)
Bastion & NAT Instance	1	50% utilization/month	Linux on t2.mciro	On-Demand (No contract)

Storage: Amazon EBS Volumes

Description	Volumes	Volume Type	Storage	IOPS	Throughput	Snapshot Storage
crawler nodes	3	General Purpose SSD (gp2)	8GB	100	128 MiB/sec	0 GB/month
Content DB (MongoDB)	1	General Purpose SSD (gp2)	15 GB	100	128 MiB/sec	0 GB/month

Amazon RDS: On-Demand DB Instances

Description	DB Instances	Usage	DB Engine & License	Class & Deployment	Storage
Content Seen, URL filter, DUE	1	50% utilization/month	PostgreSQL	db.t2.micro standard (Single-AZ)	General Purpose 20 GB

Amazon Elastic Cache: On-Demand Cache Nodes

Cluster Name	Nodes	Usage	Node Type
used by fetcher, filter, due, contentseen	1	50% utilization/month	cache.t2.micro

Amazon VPC Service:

Data Transfer:

In	10 GB/Month

Figure 4.6: Whirlpool Hardware Specification on AWS

From table 4.6 under Amazon EC2 services, crawler nodes and the Bastion host sits on a on-demand EC2 instance which is of type t2.micro. The t2.micro has single core virtual CPU (vCPU) with 1 GiB of Memory. Each crawler node runs a message broker - RabbitMQ to interconnect the crawler subsystems. According to RabbitMQ Documentation [6], the host should have at least 128 MB memory available at all times. Moreover, it wont accept any new messages when it detects that its using more than 40% of avialable memory. Being said that, there is always an option to change the current instance type from t2.micro to t2.small and so on depending on usage. MongoDB which collects extracted text is also hosted on On-Demand t2.micro instance type, which is a 1 GB, dual-core vCPU operating at 50% of its capacity.

Coming to Amazon Storage, the crawler node is coupled with Elastic Block Storage(EBS) volume of 8 GiB. The volume is general purpose SSD capped at 100 IOPS giving a throughput of 128 MB/sec. For mongoDB an EBS volume of 15GB is used. It is formatted with XFS filesystem for its data directory as recommended in its production checklist.

Whirlpool uses one On-Demand RDS instance with PostgreSQL engine operating at 50% utilization monthly. The underline managed Operating System is a db.t2.micro deployed in a single AZ. PostgreSQL to store fingerprints of a web page content at a given URL, robots.txt attributes of each site, and URLs already enqueued to be crawled.

Amazon ElasticCache provides a choice between Memcached & Redis Instance. It will be used by URL filter subsystem of whirlpool.

Table 4.7 provides approximate monthly billing information of AWS services used by this project to build, test, and run experiments. The calculation was performed using AWS monthly calculator. At the time of this writing, the author is enrolled is 12-month AWS Free tier access which discounts most of the services the crawler system leverages.

AWS Resource Cost Estimation				
Service Type	Components	Region	Component Price	Service Price
Amazon EC2 Service (US East (Ohio))				$25.95
	Compute:	US East (Ohio)	$21.25	
	EBS Volumes:	US East (Ohio)	$4.70	
	EBS IOPS:	US East (Ohio)	$0	
Amazon RDS Service (US East (Ohio))				$8.89
	DB instances:	US East (Ohio)	$6.59	
	Storage:	US East (Ohio)	$2.3	
Amazon ElastiCache Service (US East (Ohio))				$6.23
	On-Demand Cac	US East (Ohio)	$6.23	
Amazon VPC Service (US East (Ohio))				$0.00
AWS Data Transfer In				$0.00
	US East (Ohio) Region:		$0.00	
AWS Support (Basic)				$0
	Support for all AWS services:		$0	
		Free Tier Discount:		$-26.82
		Total Monthly Payment:		$14.25

Figure 4.7: Whirlpool Hardware Cost Estimation on AWS

With the free-tier in use, the total price is dropped almost by 50 %. Under free tier, EC2 is limited to 750 hours/month which equates to approx. 24 hours/day for 30 days using only Linux, RHEL. Any combination of EBS (SSD/Magnetic) is 30 GiB. Amazon RDS again limited to 750 hours/month upto 20GB of general purpose SSD database storage. Amazon ElasticCache - 750 hours. NAT instance is priced similar to other EC2 crawler instance types. There is no charge for data transfer between cross-region multi-AZ as this implementation is only us-east-2 single AZ. Also, the data transfer into AWS cloud is free.

4.7 Boot Process

A fresh installation of whirlpool crawler comprising of multiple services begins by first packaging production code for each service as docker image. The below code snippet shows **Dockerfile** of **whirpool-fetch** component. Lines 1, 28, and 31 show **dev** and **prod** target images get derived from **base** image. The specialized image produced differ by configurations, evironment variables, etc.

```
1  FROM node:10.16.0 as whirlpool-fetch-base
2
3  ARG WH_FETCH_ROOT=/home/whirlpool/whirlpool-fetcher
4  WORKDIR $WH_FETCH_ROOT
5
6  RUN apt-get update \
7    && apt-get install -y --no-install-recommends netcat \
8    && rm -rf /var/lib/apt/lists/* \
9    && useradd --create-home --shell /bin/bash whirlpool \
10   && chown -R whirlpool:whirlpool $WH_FETCH_ROOT
11
12 # files necessary to build the project
13 COPY package.json ./
14 COPY .babelrc ./
15 COPY .eslintrc.js ./
16 COPY .eslintignore ./
17 COPY package-lock.json ./
18
19 RUN mkdir logs/ \
20   && npm install --no-audit
21
22 COPY config/ config/
23 COPY src/ src/
24
25 # docker image for dev target
26 FROM whirlpool-fetch-base as whirlpool-fetch-dev
27
28 COPY scripts/wait-for-it.sh scripts/wait-for-it.sh
29 ENTRYPOINT [”bash ./scripts/wait-for-it.sh”]
30
31 # docker image for prod target
32 FROM whirlpool-fetch-base as whirlpool-fetch-prod
33
34 COPY scripts/wait-for-it-prod.sh scripts/wait-for-it-prod.sh
35 ENTRYPOINT [”bash ./scripts/wait-for-it-prod.sh”]
```

The below command packages production docker image from base layer, skipping dev layer.

```
$ docker build -t whirlpool-fetch-prod:latest --target whirlpool-fetch
    -prod .
```

Production docker images for rest of the whirlpool components are packaged using same structure and pushed to dockerhub registry. One thing to note is its not recommended to always use the **latest** tag while publishing the package but instead use semantic version tags.

```
- rihbyne/whirlpool-rmq:latest
- rihbyne/whirpool-parse-prod:latest
- rihbyne/whirlpool-contentseen-prod:latest
- rihbyne/whirlpool-urlfilter-prod:latest
- rihbyne/whirlpool-due-prod:latest
- rihbyne/whirlpool-urlfrontier-prod:latest
```

Next, provisioning AWS resources by firing up stack creation using Cloudformation(CF) template and AWS-cli. All required AWS components for whirlpool project such as Internet Gateway(IGW), ec2-instances, VPC Security Groups, Subnets, etc are expressed in YAML cf template. Following represents justa gist of omitted cf template which spans multiple pages.

```
# ---------- defines AWS resources for whirlpool project ---------------
Resources:
  # define internet gateway to attach to already existing VPC
  whirlpoolIGW:
    Type: AWS::EC2::InternetGateway
    Properties:
      Tags:
        - Key: Name
          Value: whirlpool-igw

  # vpc internet gateway attachment resource
  WhirlpoolGatewayAttachment:
    Type: AWS::EC2::VPCGatewayAttachment
    Properties:
      VpcId:
        Ref: WhirlpoolVPCId
      InternetGatewayId:
        Ref: whirlpoolIGW

  # ------------- define 4 subnets within an existing VPC ----------------------
  whirlpoolPublicSubnet1:
    Type: AWS::EC2::Subnet
    Properties:
      VpcId: "vpc-0c7a0ba5e8cad5b83"
      CidrBlock: "10.0.0.0/28"
```

Given snippet from whirlpool CF template uses **`cloud-init`** scripts that run shell scripts at launch. automating postgres post-initialization, docker & docker-compose setup. Finally, the shell script log is redirected to log directory for traceback.

```
# --------- define crawler node 1 ------------
whirlpoolCrawlerNode:
  Type: "AWS::EC2::Instance"
  Properties:
    AvailabilityZone: "us-east-2a"
    BlockDeviceMappings:
      - DeviceName: "/dev/sda1"
        Ebs:
          VolumeType: "gp2"
          DeleteOnTermination: "true"
          VolumeSize: "8"
        NoDevice: {}
    ImageId: !Ref UbuntuImageId
    InstanceInitiatedShutdownBehavior: "stop"
    InstanceType: !Ref InstanceTypeParameters
    KeyName: !Ref SSHKeyPair
    Monitoring: false
    PrivateIpAddress: "10.0.0.56"
    SecurityGroupIds:
      - !GetAtt whirlpoolCrawlerSecGrpPvtSub4.GroupId
    SourceDestCheck: true
    SubnetId: !Ref whirlpoolPrivateSubnet4
    Tags:
      - Key: Name
        Value: whirlpool-crawler-node
      - Key: Capacity
        Value: 8GiB
      - Key: ec2-purpose
        Value: mercator crawler
      - Key: placement
        Value: pvtsub4
    Tenancy: "default"
    UserData:
      Fn::Base64:
        !Sub |
          #cloud-config
          repo_update: true

          packages:
            - apt-transport-https
            - ca-certificates
            - curl
            - software-properties-common
            - gnupg
            - netcat
```

```
674
675             runcmd:
676               - [ sh, -c, "export APT_KEY_DONT_WARN_ON_DANGEROUS_USAGE=1"]
677               - [ sh, -c, 'curl -fsSL "https://download.docker.com/linux/ubuntu/gpg" |
                  ↪  apt-key add -' ]
678               - [ sh, -c, "apt-key fingerprint F273FCD8" ]
679               - [ sh, -c, 'add-apt-repository "deb [arch=amd64]
                  ↪  https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"' ]
680               - [ apt-get, update ]
681               - [ apt-get, install, -y, docker-ce ]
682               - [ sh, -c, "groupadd docker" ]
683               - [ sh, -c, "usermod -aG docker ubuntu" ]
684               - systemctl enable docker
685               - [ docker, network, create, whirlpool-net ]
686               - [ sh, -c, 'curl -L
                  ↪  "https://github.com/docker/compose/releases/download/1.24.1/docker-compose-$(uname
                  ↪  -s)-$(uname -m)" -o "/usr/local/bin/docker-compose"' ]
687               - [ chmod, +x, "/usr/local/bin/docker-compose" ]
688               - [ sh, -c, 'curl "https://www.postgresql.org/media/keys/ACCC4CF8.asc" |
                  ↪  apt-key add -' ]
689               - [ sh, -c, 'echo "deb http://apt.postgresql.org/pub/repos/apt/
                  ↪  $(lsb_release -cs)-pgdg main" > /etc/apt/sources.list.d/pgdg.list' ]
690               - [ apt-get, update ]
691               - [ apt-get, install, -y, --no-install-recommends, postgresql-client-11 ]
692               - [ rm, -rf, "/var/lib/apt/lists/*" ]
693               - [ sh, -c, "cd /root; touch .pgpass" ]
694               - pwd
695               - [ sh, -c, "chmod 0600 /root/.pgpass"]
696               - [ sh, -c, 'echo
                  ↪  "${RDSEndpoint}:5432:*:${RDSMasterUsernameParam}:${RDSMasterPWDParam}"
                  ↪  > /root/.pgpass' ]
697               - [ sh, -c, "while !(nc -z ${RDSEndpoint} 5432); do sleep 3; done"]
698               - [ psql, -h, ${RDSEndpoint}, -U, ${RDSMasterUsernameParam}, -c, "CREATE
                  ↪  USER ${RDSWhirlpoolUserParam} WITH PASSWORD '${RDSWhirlpoolPWDParam}'"
                  ↪  ]
699               - [ sh, -c, 'psql -h ${RDSEndpoint} -U ${RDSMasterUsernameParam} -c "GRANT
                  ↪  ALL PRIVILEGES ON DATABASE ${RDSDBNameParam} TO
                  ↪  ${RDSWhirlpoolUserParam}"' ]
700             output: {all: '| tee -a /var/log/cloud-init-output.log'}
```

The template file needs to be uploaded to S3 before CF reads it to create stack. Appropriate IAM S3 policy and bucket level policy imposed from administrator account, is effective to limit GET, PUT, LIST only to given CF template file. A constraint under IAM CF policies is set to allow stack creation/deletion/update from existing CF template.

```
$ aws s3 cp whirlpool-cf-template.yml s3://whirlpool-cf-templates/
```

This command schedules stack creation on CF dashboard triggered using aws-cli. It takes couple of minutes before all defined resources get created. The **create-stack** command contains arguments in key, value pair which map to input parameters and AWS specific parameters in the template.

```
$ aws cloudformation create-stack --stack-name whirlpool-crawler \
--template-url https://wh-cf-templates.s3.us-east-2.amazonaws.com/cf-
    template.yml \
--parameters ParameterKey=rdspwd,ParameterValue=<password> \
ParameterKey=rdsdbname,ParameterValue=<mydb> \
ParameterKey=rdswhuser,ParameterValue=<exampleuser> \
ParameterKey=rdswhpwd,ParameterValue=<examplepassword> \
ParameterKey=rdsendpoint, \
ParameterValue=whirlpool-postgres-prod.cmogaprwtcsg.us-east-2.rds.
    amazonaws.com
```

Once the stack on the CF dashboard is flagged 'green' indicating complete, production docker-compose is securely copied over to private e2 instances in the private subnet 4. The second command starts all the services specified in the compose file configuration.

```
$ scp prod-docker-compose.yml whirlpool-crawler-1:~/
$ docker-compose -f prod-docker-compose.yml up -d
```

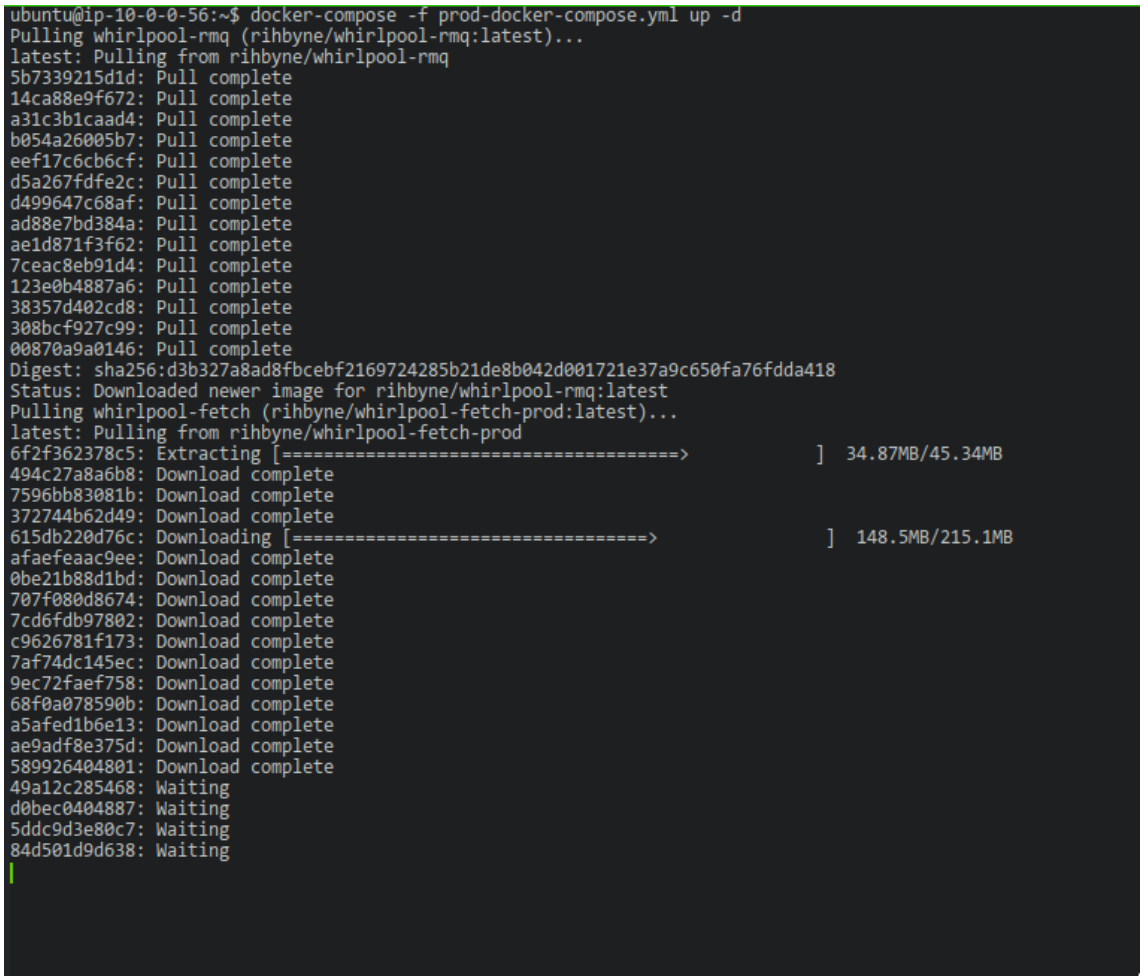

Figure 4.8: docker pull, extract, start containers

Deleting the stack **whirlpool-crawler** terminates ec2 machines and dependent components in chronological order.

```
$ aws cloudformation delete-stack --stack-name whirlpool-crawler
```

5 | Experiment Results & Justification

This section is based on lessons learned and any observations made in section 3.

5.1 RabbitMQ Dashboard

5.2 Fetcher: Response times for a sample of N requests to 3 different seed servers

5.3 Content SeenTest: Near Duplicate Detection with Simhashing

display table of simhashing and similarity with jaccard similarity

In Summary, when it comes to detecting similarity between two web pages, the pages are converted into bag of phrases. The challenge lies in storage requirements because the word size for a phrase p adds a space complexity of O(np), where n is no. of times we have to save the document. Applying a hash function outputs a integer and is able to improve space complexity of bag of phrases to O(n). Minhash[20] brings storage requirement to O(1) but time complexity of query documents is O(n). Simhashing[20] performs better than Minhash by querying on a fix q sorted list of hashes. Its time complexity is given as O$(q * log(n))$. Efficient Querying and Insertion of simhashes calls for project implementation of its own.

5.4 Hash based rebalancing

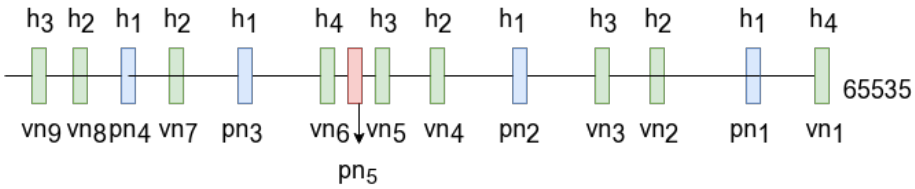

Figure 5.1: adding new node to existing cluster of nodes

Key observation while rebalancing a cluster when adding or removing a node like pn_5:

- entire vnodes are moved between pnodes
- number of vnodes present do not change
- assignment of hash of urls to vnodes do not change
- only the assignment of vnodes to pnodes is changed

The maximum number of pnodes that can be provisioned to manage scalability is equal to total number of vnodes. At some point if the crawler process changes its property from being just a topical crawler to also be comprehensive, its compute capacity can get overhauled, thus requiring more horizontal scalability. Determining how many machines the crawler can outgrow to depends on what needs to be accomplished with the crawler.

5.5 MongoDB documents

5.6 Postgres ContentSeen & DUE Fingerprints

5.7 Whirlpool as a Microserivce Application

To be completed

6 | Conclusion & Future Work

This chapter is a list of follow-up ideas that can be turned into potential projects. These are complex and important topics that deserve a report of their own. This paper wont wouldn't do them justice by making them superficial side notes.

Experimenting with Distributed Data

Data Indexing using ElasticSearch Engine

Adding comprehensive coverage using ML techniques
Discovering new relevant websites by adding Comprehensive Coverage using ML techniques

Web Scrapping
[21]

Categorizing Jobs using Supervised/RNN classifier

Whirlpool-SeenTest: Inserting and Querying Simhashes
To work on a performant solution to build the data store to query simhashes efficiently and achieve running time complexity of O($q*log(n)$) mentioned in section 5.3. To implement this, various backends can be taken into consideration such as Cassendra, MongoDB, or Riak.

7 | Timeline

I intend to graduate in fall 2019 i.e next semester. Below is my timeline to execute the thesis starting first week of June, 2019:

(Week 1 - 7) First get the single node crawler running on local development environment. To achieve this,

- (Week 1) Build docker images of data stores involved. I will be using PostgreSQL for fingerprintDB, Robots Exclusion Policy, and DUE url set. The job data will be stored in MongoDB(development & production). Each data store will maintain local.yml & production.yml version of dockerfile which will point to local and aws version of databases, respectively.
- (Week 1) Build docker image of RabbitMQ with default configuration. Followed by getting acquainted with enough terminology and tutorials to get started. This image will be shared with all subsystems of a crawler.
- (Week 2) Initialize git repository for each subsystem of the crawler. Build docker image for each subsystem, specify RabbitMQ as its dependency. Bind each subsystem to a queue through a message broker depending on its role as a consumer/producer. Test for open connections of each consumer/producer.
- (Week 2) Boot all the subsystems in sequence using docker compose. Test the message passing between services. This stage ensures the tooling required is up & running as expected.
- (Week 3) URL Frontier queue is the 1st step of the crawler subsystem. Initially, maintain a single FIFO queue under the frontier with seed sets listed in section 3.1. Code functionality for consumer & producer scripts bound to a queue of fetcher & DNS subsystem.
- (Week 4) Given the seed URL u and a page p, code functionality for consumer & producer scripts of Parser, Content Seen subsystems, respectively as separate services. Test message passing among fetcher, parser , & content seen.
- (Week 5) Next, once we have extracted links from page p, code functionality for consumer & producer scripts bound to a queue for URL filter subsystem. Test message passing between this filter and previous subsystems implemented.
- (Week 6) DUE tracks present & past history of URLs in Frontier regardless of whether they are 1-time crawl or continuous crawls. Implement consumer & producer scripts for DUE. Test message passing between subsystems implemented so far.
- (Week 7) Write code for front(priority) and back(politeness) queue of URL Frontier subsystem. Verify messages published & consumed. Test message passing between subsystems implemented so far.

(Week 8) At this stage, ensure the crawler as a whole is running as a single node on local machine.

(Week 8 & 9) Next, Setup AWS Infrastructure for whirlpool by following the diagram. Build and test the docker images in isolation pointing to AWS data stores within the pvt subnets.

(Week 10) Get crawler running on single node within a private subnet of AWS VPC.

(Week 10 & 11) Initialize a repo. for host splitting module. Implement consistent hashing algorithm. This will parallelize crawling.

(Week 11 & 12) Make 2nd identical clone of the crawler within a private subnet of AWS VPC. Observe & verify traffic is split equally between the nodes.

(week 13) Demonstrate Whirlpool working to Dr. Soltys. Make any changes requested in the code or the report.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*.

[2] Martin Kleppmann (2016). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*

[3] Jeffrey Barr (2010). *Host Your Web Site in the Cloud: Amazon Web Services Made Easy*

[4] Kyle Banker (2011). *MongoDB in Action*

[5] Martin Abbott, Michael Fisher (2011). *Scalability Rules: 50 Principles for Scaling Web Sites*

[6] Alvaro Videla and Jason J. W. Williams. (2012) *RabbitMQ in Action: Distributed Messaging for Everyone*

[7] Olston, C., & Najork, M. (2010). *Web crawling. Foundations and Trends in Information Retrieval, 4(3), 175-246.*

[8] Heydon, A., & Najork, M. (1999). *Mercator: A scalable, extensible web crawler.World Wide Web, 2(4), 219-229.*

[9] Boldi, P., Codenotti, B., Santini, M., & Vigna, S. (2004). *Ubicrawler: A scalable fully distributed web crawler. Software: Practice and Experience, 34(8), 711-726.*

[10] Lee, H. T., Leonard, D., Wang, X., & Loguinov, D. (2009). *IRLbot: scaling to 6 billion pages and beyond. ACM Transactions on the Web (TWEB), 3(3), 8.*

[11] Harth, A., Umbrich, J., & Decker, S. (2006). *Multicrawler: A pipelined architecture for crawling and indexing semantic web data. In The Semantic Web-ISWC 2006 (pp. 258-271). Springer Berlin Heidelberg.*

[12] Seeger, M. (2010). *Building blocks of a scalable web crawler. Master's thesis, Stuttgart Media University.*

[13] Karger, D.; Lehman, E.; Leighton, T.; Panigrahy, R.; Levine, M.; Lewin, D. (1997) *Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web*

[14] Scott Wiersdorf on system administration, ssh, security, featured *An Illustrated Guide to SSH Tunnels (2015)*

[15] Patrick Hunt; Mahadev Konar; Flavio P. Junqueira; Benjamin Reed *ZooKeeper: Wait-free coordination for Internet-scale systems*

[16] Mike Burrows *The Chubby lock service for loosely-coupled distributed systems*

[17] Sam Newman (2015) *Building Microservices: Designing fine-grained systems*

[18] Roberto Di Cosmo, Stefano Zacchiroli *Software Heritage: Why and How to Preserve Software Source Code*

[19] V. Fuller, T. Li, J. Yu, K. Varadhan (Sept. 1993) *Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy*

[20] Gurmeet Singh Manku, Arvind Jain, Anish Das Sarma (2007) *Detecting Near-Duplicates for Web Crawling*

[21] Matthew E. Peters, Dan Lecocq (May, 2013) *Content Extraction Using Diverse Feature Sets*

[22] Scott Kirkpatrick (2009) *Architecture of the internet archive*

[23] Monica Rogati (June, 2017) *The AI Hierarchy of Needs*

[24] Maxime Beauchemin (Jan, 2017) *The Rise of Data Engineer*

[25] David Karger, Eric Lehman, Tom Leighton, et al.(1997) *Consistent Hashing and Ran-dom Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*

[26] **https://www.docker.com/**

[27] **https://aws.amazon.com/**