Distributed Web Crawling Using Network Coordinates

Barnaby Malet
Department of Computing
Imperial College London
bwm05@doc.ic.ac.uk

Supervisor: Peter Pietzuch
Second Marker: Emil Lupu

June 16, 2009

Abstract

In this report we will outline the relevant background research, the design, the implementation and the evaluation of a distributed web crawler. Our system is innovative in that it assigns Euclidean coordinates to crawlers and web servers such that the distances in the space give an accurate prediction of download times. We will demonstrate that our method gives the crawler the ability to adapt and compensate for changes in the underlying network topology, and in doing so can achieve significant decreases in download times when compared with other approaches.

Acknowledgements

Firstly, I would like to thank **Peter Pietzuch** for the help that he has given me throughout the course of the project as well as showing me support when things did not go to plan.

Secondly, I would like to thank **Johnathan Ledlie** for helping me with some aspects of the implementation involving the Pyxida library.

I would also like to thank the **PlanetLab support team** for giving me extensive help in dealing with complaints from web masters.

Finally, I would like to thank **Emil Lupu** for providing me with feedback about my Outsourcing Report.

Contents

Chapter 1

Introduction

The Internet has become an integral part of everyday life and today is the way in which we store and share all of the information that our society has to offer. The widespread use of it is evident and because of its popularity, search engines now process hundreds of thousands of keyword searches per second (in March 2009 Google reached 293 Million searches per day).

Companies which provide these services are required to do an enormous amount of background work in order to produce the results which we take for granted: The underlying web pages on the Internet must be processed, analysed and indexed in order to determine the best answer to a keyword search and this analytical process can't even begin until the underlying web pages have been downloaded.

The size of the Internet can only be described as massive: Netcraft[1] now reports over 236 million unique hostnames and in a recent interview, Google engineers stated that the number of unique URLs were in the trillions and growing by a billion every day[2].

In order to download this vast amount of data, search engines employ web crawlers. A web crawler can be described as an automated system which downloads parts of the Internet by following hyper-links between web pages; because of the rapid growth of the Internet these systems generally run on a 24/7 basis.

Designing an effective crawler is a non-trivial task: the crawler must efficiently process and store the large amount of data contained within the Internet. Furthermore, deciding which web pages are relevant and which are not is difficult: the Internet is not only home to massive "spam-farms" designed to artificially manipulate search engine rankings but to genuine web sites such as Yahoo and Ebay which report hosting millions of web pages.

Early crawlers designed for the infant web were implemented as centralised systems located in a single data-centre. However because of the wide spread growth the Internet, the task of crawling it has become far too much for a single crawler to handle. As such, more recent efforts have focused on the design and development of distributed systems composed of many crawlers located around the world.

The creation of a distributed crawling system requires us to address problems such as scalability and fault-tolerance which are ubiquitous in distributed computing. But it also requires us to address problems such as balancing the assignment load, reducing communication overhead and optimising network proximity between crawlers and web servers. The algorithm which influences these factors is called an assignment strategy and is the job distribution policy adopted by the system (e.g. it decides which crawler should download which web pages).

There have been numerous publications which address the balance and communication overhead issues, and more recently researchers have begun looking at optimising network proximity. Currently we can distinguish three schools of thought when implementing distributed crawler assignment strategies:

1. **Hash based** assignment strategies (Section 2.2.1) involve computing a unique hash function for each crawler in the system such that the set of URLs seen during the crawl can be hashed and assigned to the crawlers. Such strategies are easy to implement, scalable and provide good solutions for the balancing and communication overhead problems but do not in general provide a way to cater for network proximities (although some use ICMP Pings as a heuristic).

2. **Geographic based** assignment strategies (Section 2.2.2) use location data (obtained from Internet registrars) to assign targets to crawlers based on minimum geographic distance. Such systems work well and have been shown to both minimise communication overhead (because web sites tend to reference web sites in the same geographic location) and are able to achieve a decrease in download time relative to hash based assignments.

3. **RTT based** assignment strategies (Section 2.2.4) use ICMP Ping latencies obtained either before or during a crawl to assign target URLs to the crawler with the lowest latency to the web server. This method has also been shown to achieve a reduction in download time over the hash based assignment.

Using either a Geographic and RTT based assignment strategy has been shown to decrease in download times relative to hash based assignments.
Although geographic based assignments download web pages from close crawlers, the information used is obtained from human inputed data (registrar data) and as such is slightly prone to error, but more importantly geographic distances give no indication of the speed of the underlying link between crawler and target web server.

RTT based assignments give a better indication of the properties of the underlying network link used, but in the case of off-line measurements they are not suited for large crawls where there is little or no a-priori knowledge about the target web servers. In the case of RTT strategies which use active probing, this generates more, un-necessary network traffic between the crawler and the web server. Furthermore, it has been established that ICMP Pings are not strongly correlated with throughput along a network path. Specifically they give some indication of throughput up to the point where the network path becomes congested [23]. Also because ICMP Pings are transmitted at the Network layer of the OSI model, they do not give an indication of the response time of the web server or computational load of the crawler.

In addition, we are unaware of any assignment strategy which is well suited for a system composed of heterogeneous resources or one which has shown the ability to dynamically compensate for adverse changes in the underlying crawlers of the system (such as decreases in available bandwidth or increases in computational load). This is an important problem to address because exclusive access to massive data-centres is reserved for organisations who can afford it. Smaller organisations wishing to crawl the Internet are likely to use shared systems composed of different hardware whose load and available bandwidth is less predictable.

Network coordinate systems (NCs) fall into an area of research that is increasingly attracting more attention from the scientific community; they provide the means in which to assign a set of coordinates to a set of network hosts in a Euclidean space such that the distance between two coordinates accurately predicts the latency between the two hosts; latencies estimated this way can either be connectionless ICMP Ping latencies. Or the round trip time of a message transmitted through a TCP connection using user level time-stamping [37]. By using the second kind of latency, it has been shown that a host which undergoes high computational load will see its coordinate move further away in the NC space and one which is under low computational load will see its coordinate move closer. Coordinates such as this are termed load-aware network coordinates (LANCs).

NC systems previously required all nodes to participate actively in measurements in order to compute their coordinates. Recently the concept of Proxy Coordinates [44] has emerged wherein a set of oblivious hosts (not running any special software) can be given accurate network coordinates by a set of hosts which are part of an NC overlay. This is achieved by measuring the latencies to the oblivious hosts from multiple vantage points in the overlay and computing their coordinates accordingly, however the concept has only

been shown to work where the latencies between the overlay hosts and the oblivious hosts are obtained using ICMP Pings.

Network Coordinate systems have the ability to predict latencies even in rapidly changing network conditions. Furthermore algorithms such as Vivaldi [33] can use messaging that the application would have sent out anyway had it not been using an NC overlay.

As such network coordinate systems seem well suited for a distributed crawler assignment strategy: If we were to assign each crawler in a distributed system a load aware network coordinate, then the system could quickly determine which crawlers are under low load and which ones were under high load. As such an assignment strategy could avoid ”bad” crawlers by assigning URLs to crawlers with low load.

If we were then embedded target web servers into the same Euclidean space as the crawlers using Proxy Coordinates in such a way that the distance between a crawler and a target web server gave an estimation of the time it would take for the crawler to download from the web server. Then we could develop an assignment strategy which could quickly and accurately determine which crawler would be best suited to download from a given web server simply by measuring their respective distances in the NC space. We will do this by using the download times observed between crawlers and web servers as latency measurements for the Proxy Coordinate computation.

In this report we will describe the Background Research (Chapter 2), Design (Chapter 3), Implementation (Chapter 4) and Evaluation (Chapter 5) of a distributed crawling system which uses a network coordinates to accurately predict download times and which can detect and compensate for changes in network throughput and computational load. Our system is able to achieve this without additional communication with web servers, using download times obtained by fetching content from web servers.

Contributions

- A way in which to assign coordinates to web servers and crawlers in a Euclidean space such that the distance between a crawler and a web server gives an accurate prediction of the download time and does not require any additional communication with web servers.
- A fully distributed & fault tolerant crawling system which is able to use the Network Coordinate system to find optimal assignments for crawlers and which can adapt to changes in the underlying crawlers.
- A Test & Distribution framework designed for the rapid deployment of software to PlanetLab nodes.
- An evaluation of the behaviour of the underlying network coordinate system.
- An evaluation of the assignment strategy relative to other assignment strategies.

Chapter 2

Background

Before we begin to develop our assignment strategy, it is important that we understand the different approaches in creating web crawlers as well as the common problems faced when designing them. As such we will begin our discussion with a survey of non-distributed web crawlers. We will follow this with an in depth look at current distributed crawling assignment strategies, which will give us an understanding of the state of the art and give us a valuable basis on which to compare and evaluate our work. We will then look at Network Coordinate systems, focusing on Vivaldi[33] in particular. This will help us comprehend the concepts and the applications of these systems and will give us a valuable starting point for our work. Following this, we will briefly look at the different metrics which can be used for determining network distances. And finally, we will briefly present the technology and the tools used to implement, test and evaluate our system.

2.1 Web Crawling

There are several properties of the Internet that make web crawling very difficult: its vast amount of data, its rapid rate of change and its dynamic page generation.
As such it is required that a web crawler be robust, scalable and make efficient use of available bandwidth. There exist many different designs for a web crawler, however all crawlers are built around a core set of components. In this section, we will briefly present these components as well as some common issues that influence crawler design and operation.

2.1.1 Web Crawler Architecture

```
URLSeen  = [];
URLQueue = ['www.google.com', 'www.xyz.com', ...];

while (URLQueue != []){
        currURL  =  URLQueue.getNext();
        targetIP =  DNS.resolveHostname( currURL.getHostname() );
        html     =  downloadPage( targetIP + currURL.get );

        URLSeen.add( currURL );

        outLinks = html.extractURLs();

        foreach (nextURL in outLinks){
                if ( !URLSeen.contains( nextURL ) ){
                        URLQueue.add( nextURL );
                }
        }
}
```

Figure 2.1: Basic Crawling Algorithm

The operation of a basic web crawler (Figure 2.1) can be broken down into several phases:
(1) The crawler queue (URLQueue) is initialized with a set of seed URLs. (2) The next URL (currURL) is fetched from the queue. (3) the URL hostname is resolved to an IP address using a DNS client. (4) the HTML is downloaded and the out-linking URLs are extracted. (5) currURL is added to the set of previously seen URLs. (6) Each URL extracted from the HTML is added to the URLQueue. (7) steps 1 to 6 are repeated until the URLQueue is empty or some heuristic is met.

There are four core components in any crawler design: the URL Queue, The Downloader, The URL Seen data structure, and the DNS Resolver. We will now discuss these four components in greater depth.

URL Queue

The **URL Queue** is the data structure which stores the set of web pages that the crawler has yet to download. Its ordering has multiple implications on the behaviour of the crawl such as *Politeness* and Type of Search. Most crawlers search the web using breadth-first-search starting from a *Seed* set of URLs, this can easily be implemented using a FIFO queue.

In a standard FIFO queue, elements are retrieved in the order that they were inserted, however when a large number of web pages hosted on the same web server are located at the head of the queue, care must be taken so as not to overload the web server by requesting all of the web pages simultaneously.[1]. This is an issue of politeness which we will discuss further in Section 2.1.2.

Because the number of URLs encountered during a crawl can become very large, the URL Queue must be implemented in such a way as not to overload the crawlers RAM. Mercator's[47] queue is split into a set of per-domain sub-queues, a buffer of 600 URLs per sub-queue is held in memory and the rest are cached to disk. Koht-arsa et al.[12] construct an AVL tree of URLs, which only stores the differences between URLs; each element in the tree is individually compressed using the LZO algorithm. This method seems to be quite effective: it takes an average of 92 microseconds to store a URL and 6 microseconds to retrieve

[1] A large number of simultaneous requests to a web server is often considered as a denial of service attack.

a URL. Like Mercator, IRLBot[49] also maintains only a small subset of its Queue in memory, the bulk of which is stored on disk using a method called *Disk Repository With Update Management*(DRUM) which we will discuss in the next section.

URL Seen

The **URL Seen** data structure stores previously downloaded URLs and allows the crawler to determine whether a given URL has been crawled before. it should support two operations:

- **insert(URL)** which adds a new URL to the data structure.
- **contains(URL)** which returns true if the given URL has previously been added to the repository.

The implementation of this data structure is not trivial because the amount of URLs encountered during a crawl can reach into the billions[49] as the crawl progresses. As such, iterating through the whole set of crawled URLs each time a **contains()** call is made is not an option.
Mercator[47] stores fixed-sized checksums of the URLs in a memory buffer. Each URL checksum is associated with a pointer that references its text on disk. When the buffer becomes full, it is merged with a disk store in one pass, duplicate URLs are removed in the process. Mercator also maintains a buffer of popular URLs (as determined by their in-link factor) in order to speed up query times.
PolyBot[11] and the crawler described by Koht-arsa et al.[12] store the full URL text in memory which is organized into a binary tree and AVL tree respectively. Once the trees become too large, they are compressed and merged with a disk store. It is noted in [49] that this is more computationally expensive than the method used in Mercator because of the tree searching and compression algorithms.

DRUM IRLBot[49] uses *Disk Repository With Update Management*(DRUM) which permits large scale storage and retrieval of $< key, value, aux >$ tuples, where key is a hash value of the URL, $value$ is unused, and aux is the URL text. From a high level perspective, tuples are streamed into memory by the downloader. As the memory fills up, each $< key, value >$ pair is moved into a pre-allocated bucket on disk based on its key, the aux portion is fed into a separate file in the same order.
Once a bucket on disk reaches a certain size, it is moved into a memory bucket buffer and sorted. The previously stored disk cache is then read into memory in chunks of Δ bytes and compared with the bucket. Duplicate keys are eliminated, and unique ones are merged into the disk cache. The bucket containing the aux portion is also read into memory in chunks of size Δ, the aux portions which are associated to keys that were found to be unique are passed to the URL Queue for downloading. It is important to note that all buckets are checked in one pass through the disk cache. It is shown in [49] that DRUM outperforms both the methods used in Mercator[47] and in Polybot[11]: the URLSeen overhead is 5 times less than Mercator's and 30 times less than Polybot for 800 Million Bytes of URL data.

HTML Client

The **HTML Downloader** is an HTTP client that is responsible for fetching content from the web servers. There are two main approaches that can be adopted when designing it: (1) Multiple threads that synchronously request one URL from the queue at a time. (2) A single thread that polls a large number of asynchronous HTTP connections.
Mercator's[47] downloader is built using multiple threads, each processing a synchronous HTTP request, their downloader supports the FTP and Gopher protocols on top of the standard web HTTP protocol. Shkapenyuk et al. [11] adopt the polling approach: a Python script simultaneously opens up 1000 connections to different servers and polls the connections for arriving data.
There is no obvious performance to be gained by adopting one approach over the other, however it is noted in [47] that a multi-threaded approach leads to a much simpler program structure. Switching between threads is left down to the operating system scheduler, and the sequence of tasks executed by each thread is much more self evident.

DNS Resolver

The **DNS resolver** is an important part of the system and should also be implemented using a multi-threaded asynchronous approach. In Mercator, the authors were initially using the Java DNS interface to perform lookups; however, they discovered that this interface was synchronized and was making DNS lookup times very slow. It turned out that the DNS interface distributed in Unix (gethostbyname() function provided by BIND[51]) was synchronous. Although this has now been fixed in more recent versions of BIND, a call to the function can block for up to 5 minutes before a time out. Mercator's authors ended up writing their own multi-threaded DNS resolver:
The resolver would forward DNS requests to a co-located local name server, which would then do the actual work of contacting the authoritative server for each domain. The resolver would cache previous requests and could process multiple requests in parallel using multi-threading, this resulted in a 25% decrease in DNS query time.
Shkapenyuk et al. [11] implemented a DNS resolver in C++ using the GNU adns[52] asynchronous client library to query a co-located DNS server. The adns library does the same job as the resolver implemented in Mercator. Shkapenyuk et al note that DNS lookups generate a large amount of network traffic which may affect crawling speeds due to a limited router capacity and as such DNS requests should be cached in an effort to reduce this traffic.

2.1.2 Issues in Web Crawling

Politeness

An important issue which needs to be addressed when designing a crawler is politeness. Crawlers must be considerate of web servers, they should not overload a web server by requesting a large number of web pages in a short space of time, adhere to restrictions outlined by web site administrators and identify themselves when requesting pages.

Request Intervals In order to throttle requests to web servers, crawlers generally observe a waiting time τ between two simultaneous requests to a web server. The crawler described by Shkapenyuk et al.[11] waits $\tau = 30secs$ between two downloads. To enforce this waiting time, Shkapenyuk et al[11] implement a shuffling mechanism inside of their queue, wherein the queue is scrambled into a random order so that URLs from the same web server are spread out evenly throughout the queue.
Mercator[47] implements its URL queue as a collection of sub-queues: each unique domain has its own queue; when a URL is added to the queue, its canonical domain name is extracted, and it is placed on the queue corresponding to that domain. Each downloader thread is assigned one sub-queue, it only removes URLs from that queue and waits a time $\tau = 10\kappa$ between requests, where κ is the previously observed time it took to download a page from the server.
In IRLBot[49] it is noted that enforcing a static waiting time, can "choke" the crawlers performance when crawling a very large web site (for example Yahoo which reports 1.2 billion objects within its own domain). For this reason, Lee et al use domain popularity as determined by the domain in-link factor to define an adaptive waiting time τ. Low ranked domains are assigned $\tau = 40sec$, high ranked domains are assigned a scaled down τ relative to their in-link factor (a domain with a few million pages would have a relatively high in-link factor). The intuition behind assigning a much lower τ to high ranked domains is that their web servers are able to handle a much larger number of requests.

Robots Exclusion Protocol Although observing an interval τ between two successive requests to the same web server is a must, there is another important standard that should be integrated into the design of a web crawler: the robots exclusion protocol[50]. It provides a way for web site administrators to indicate which parts of their sites shouldn't be crawled. It takes the form of a file (robots.txt) which resides at the root level of a domain (e.g. http://www.xyz.com/robots.txt) and should be the first thing that is retrieved when a crawler encounters a new domain. It can then be used to filter restricted URLs from the queue

([12]). The robots.txt file is retrieved using the downloader component and is usually cached in a special data structure[47].

We would also like to note that a crawler should identify itself to web servers. This is achieved by including a contact email address and/or URL in the HTTP request header. This is so that system administrators can contact the crawler writers if it is hitting their web server too frequently or consuming too much bandwidth.

Quality

The quality of the data collected during a crawl is important and can be improved by the crawler. The ordering of the URL queue determines the type of search of the web graph, for example, if it is implemented using FIFO ordering, then the crawler will perform a breadth-first-search of the Internet.
The queue can also be ordered by taking into account the in-link factors of pages (using PageRank[48] for example), however Najork et al[46] have argued that a breadth-first-search of the web yields high quality pages early on in the crawl. UbiCrawler[4] uses a BFS search with a per-host depth limit of 8, which at the end of the crawl achieves a cumulative page rank similar to that of a theoretical PageRank based search.
It is important to note that there exist a large number of infinitely branching "crawler traps" and "spam sites" on the Internet whose pages are dynamically generated and designed to have a very high in-link factor. Because pages are free, it is easy to develop such a system. It is noted in [49] that by calculating in-link factors based on resources that spammers have to pay for such as domain names or IP addresses, it makes it a lot harder for people to artificially manipulate page rankings.
IRLBot[49] maintains multiple queues, URLs are added to queues based on their domain in-link factor (or budget). The queue which contains the URLs with the highest budget is processed first, newly arriving URLs are spread amongst the remaining queues based on their budget. Once the current queue is empty, the other queues are prioritized and downloaded. Because newly seen URLs will have a relatively low domain in-link factor, they are kept in one of the lower queues, as their budget increases (because they are seen more and more during the crawl) they become more likely to be downloaded.

2.1.3 Discussion

The main obstacle in designing a crawler is managing extremely large amounts of data in an efficient and robust way. Many crawlers are implemented using a common architectural philosophy which addresses this issue using decoupled components that manage memory well. IRLBot[49] has shown that it is possible to develop a streamlined, highly efficient crawler using highly coupled components. The IRLBot crawl is the fastest published crawl to date, over 6 billion pages were crawled over a two month period (or 1,789 pages/s). Other published crawlers did not achieve as much, Mercator[47] crawled 473 million pages in 17 days, and PolyBot[11] crawled 120 million pages over a period of 18 days.
The goal of this project is not to design a highly efficient crawler, but to design a highly optimal crawler assignment strategy based on network coordinates. As such we do not intend to rival previous crawlers in terms of crawl size; we will integrate some of the simpler ideas discussed in this section into our crawler so that we can evaluate our assignment strategy without the crawler component being a performance bottleneck.

2.2 Crawler Assignment Strategies

The Internet is constantly growing and as such has become far too vast for a single web spider to crawl[3]. Recent research has focused on the design and implementation of distributed crawling systems that crawl the web using a set of crawlers located in different data centers around the word. Such systems can be designed using a centralized approach, wherein a central server is responsible for determining assignments[11] based on some predefined strategy, or a decentralized approach with no central point of control[4]. In a decentralized distributed crawling system, each crawler will have knowledge of the assignment strategy,

when a crawler encounters a URL that it is not responsible for, it will transmit the URL to the crawler that is responsible for it.
There are several desirable properties for an assignment strategy[4, 21]:

- **Balancing**: Each crawler should be assigned approximately the same number of hosts.
- **Contravariance**: As the number of crawlers in the system increases, the percentage of the web crawled by each agent should shrink. And similarly, if the number of agents decreases, then the percentage of the web crawled by each agent should expand. In other words, by changing the size of the crawler set, the balancing property should be maintained.
- **Optimizing Network Proximity**: When a target is assigned to a crawler, it should be done in such a way as to pick a crawler that can download the target in the shortest space of time (e.g. choose a crawler that has low latency to the target or one that is in the same geographic region as the target).

Assignment strategies can be evaluated using the following criteria [3]:

- **Coverage** is how many pages the crawling system has actually crawled relative to its target. It is defined as $C = \frac{c}{u}$ where c is the number of actually crawled pages and u the number of pages as a whole the crawler has to visit. A good assignment strategy should achieve a coverage of 1, e.g. every page that was assigned has been downloaded.
- **Overlap** is defined as $O = \frac{n-u}{u}$ where n is the total number of pages crawled, and u is the number of unique pages that the crawler as a whole was required to visit. $u < n$ happens if the same page has been erroneously fetched several times. It is debatable as to how much overlap is good, and how much is bad.
- **Communication Overhead** is defined as $E = \frac{l}{c}$, where c is the number of pages downloaded by the system as a whole and l is number of URLs that the crawlers have exchanged. A good assignment strategy should aim to minimize E as much as possible.

In this section, we will discuss common assignment strategies and compare them where possible.

2.2.1 Hash Based

An early method of assigning a set of targets to a crawler was to use a Hash function to map either URLs or domains to crawlers [3, 7, 4, 12]. A common approach in hash based assignments is to use a modulo based hash function of the target hostname, the bucket mapped to by the key corresponds to a unique crawler in the system; this approach has very good balancing properties provided every crawler has absolute knowledge of all others and that the hash function is consistent among the crawler set.

Loo et al. [7] present an evaluation of a decentralized Peer2Peer crawling system based on a Distributed hash table: each crawler is a bucket in the table. They compare Hostname and Url based hashing techniques, it is shown that both techniques achieve a similar throughput ($1000Kbps$ in an 80 node system). But, as expected, a URL based hashing strategy greatly increases communication overhead because self-links (i.e. links from http://foo.bar/1.html to http://foo.bar/2.html) result in re-assignment; URL based hashing is shown to consume 6 times more communication bandwidth than hostname based hashing.

In hash based assignments, the contravariance property is not easily satisfied: It is not easy to cater for the case when a new crawler is added to or removed from the system. If the hash function was recomputed every time the crawler set changes, the set of targets assigned to each agent would vary in a non-deterministic way.
A possible outcome of this would be that fetched content is stored by a crawler that is no longer responsible for the target at the origin of the content, this then leads to that content being re-fetched by another crawler thus the duplication of fetched content and an increased Overlap.
Boldi et al. [4] Address this problem by using a technique called consistent hashing[5, 6] over domain

names. In a typical hashing function, a number of keys (targets) will map to one bucket (crawler), however in consistent hashing, each bucket is replicated a fixed number κ times. Each replica is mapped randomly on the unit circle. When a key needs to be hashed, its value v on the unit circle is computed, and its hash value is obtained by choosing the bucket whose value is closest to v. When a new crawler is initialized, each crawler generates a fixed set of replicas for that crawler. It is shown in [4], that for a large set of crawlers and $\kappa = 200$ replicas per crawler, the hashing function achieves almost perfect balance (the deviation from perfect balancing is less than 4.5%). The consistent hashing technique also handles failure well: when a crawler goes down, other crawlers will simply assign its targets to the next closest crawler on the unit circle, and as long as the replicas are evenly spread out on the unit circle, the contravariance property is maintained.

UbiCrawler varies the number of replicas κ proportional to a crawlers capacity. Hence, a crawler with high capacity (Large Memory, CPU, Bandwidth...) will be assigned a larger proportion of the web, similarly a crawler with lower capacity will be assigned a smaller proportion.
Another advantage of hashing is that each crawler has absolute knowledge of who is responsible for each newly encountered target. In terms of communication overhead, this means that a target will only ever need to be communicated to at most one other agent, no discussion or gossip between agents is necessary.
UbiCrawler provides a mechanism for page recovery: thanks to the consistent hashing technique employed, a crawler c_{new} has knowledge of the crawler c_{old} who would of been assigned a given target before c_{new} had been instantiated; this is achieved by finding the next nearest replica c_{old} ($c_{old} \neq c_{new}$) on the hash unit circle. c_{new} can query c_{old}, and if c_{old} reports that it had downloaded that target, c_{new} will skip it, intuitively this means that communication between crawlers will increase linearly with respect to the number of pages crawled, but overlap will decrease.

Using Consistent Hashing, Boldi et al produce a fully decentralized, scalable and fault tolerant crawler. Hashing provides good balancing and, in the case of consistent hashing, good contravariance properties. However it does not exploit network proximity between crawlers and targets. Loo et al. [7] propose a way to take into account network proximity when using a hash based assignment strategy: Redirection, essentially, if a crawler is assigned a target to which it does not deem itself 'close' enough, it will redirect that target to another agent that is likely to be closer to the target. The downside of doing this is that it greatly increases communication overhead, they propose setting an upper bound to the number of redirections allowed per crawler, but this solution is far from optimal. It is also noted that this redirection technique can also be used for load balancing and also to satisfy politeness constraints.

Hashing is a very easy assignment strategy to implement, however it has the major disadvantage that there is no easy way of exploiting network proximity without greatly increasing communication overhead. This is why more recent work has focused on Geographic and Latency based assignment strategies.

2.2.2 Geographic Based

This strategy consists of the analysis of the target URL or the target HTML to derive the geographic location of a target. Once this information has been determined, the target is assigned to a crawler in or close to that location; this is based on the assumptions that (1) geographic proximity between two Internet hosts indicates that they are on a close or same network, and (2) that pages which refer to a given geographical region are usually located in or close to that region[8]. This type of assignment strategy is often used in *focused web crawling*[2]: In [8] it was used to gather information about different US cities, and in [14] it was used to collect data about the Portuguese web. There are several heuristics that can be used to extract locality data from a target:

- **URL Based**: Involves the extraction of geographic information solely from the URL. In [14], they use the URL heuristic to extract institution names from URLs and map them to a geographic location; for example the URL http://imperial.ac.uk would be mapped to London as it is known that Imperial College is a London university.

[2]Web crawlers that attempt to download web pages that are similar to each other are called focused web crawlers.

- **Extended Anchor Text Based**: Involves extracting geographic information from HTML anchors[3].
- **Full Content Based**: Involves semantically analysing the full content of a web page to extract geographic information. Gao et al[8] count occurrences of city-state pairs, for example, a web page that has multiple occurrences of the word "New York" is considered to be located in New York.
- **IP Address Based**: The resolved IP address of a URL is matched against a database which maps IP addresses to Latitude and Longitude (such as hostip.info[17], GeoIp[16] or NetGeo [18]). The reader is referred to Section 2.4.1 for an explanation of how such databases work.
- **DNS RR LOC**: Is the *DNS Location Record*[15] and can be obtained by querying the DNS server, it contains the Longitude and Latitude of a domain, it is optional and generally human inputted.

An interesting point noted in [8, 14] is that pages tend to reference other pages in the same geographic location. Hence, the intuition is that by adopting a geographic based assignment policy, one can significantly reduce communication overhead as the probability that an out-link refers to the same geographic location as its parent page is very high; this is shown in [8]: by adopting a geographic assignment strategy (using the NetGeo[18] database) in a 6 crawler system, the communication overhead is very low ($E = 0.15$) when compared to a URL hash based assignment strategy ($E = 13.89$).

Exposto et al[14] use IP Address based and URL based heuristics to extract locality data from targets. In their evaluation, 88% of target locality data could be determined using the NetGeo[18] database, a URL based heuristic accounted for just 17%[4], DNS Location Record and other content based heuristics(such as counting occurrences of airport names in the target HTML) yielded little or no results (<0.1%). The accuracy of the IP address locality data obtained using the hostip.info[17] database is shown to be around 50% on a city level in [8], however an approach wherein the system is trained[5] to recognize websites yields a far better accuracy (80%), but the training is computationally expensive and has to be performed on a central server.

It is shown in [14] that a geographic assignment (Using the NetGeo[18] database) performs significantly better than a hash based assignment: with 32 crawlers, the average download time is 1000 seconds for geographic based, relative to 57000 seconds for hash based. It is noted that Exposto et al's partitioning scheme also factors inter-partition links into the assignment as in [21] which we will discuss in Section 2.2.3.

Geographic based assignment strategies are very promising: They seem to exploit network proximity well whilst reducing communication overhead. Balancing and contravariance are not discussed in [8, 14]. However in the case of [14], the set of target websites is partitioned off-line before the crawl using a graph-partitioning strategy(See Section2.2.3) which could be altered to achieve a good Balance.

2.2.3 Link & Hybrid Based

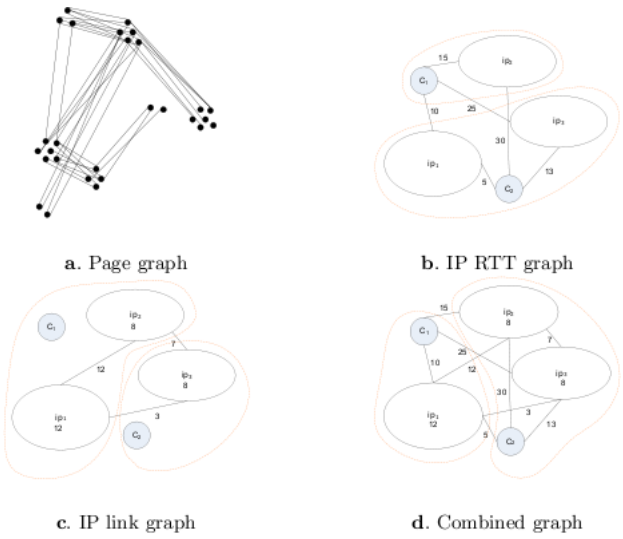
Another method of assigning crawlers to targets is to exploit web page link topolgy[14, 21]. Target sets are partitioned so as to minimize the out-links between partitions which has the major advantage of minimizing the exchange overhead between crawlers. In [21], Exposto et al. use a combination of link structure and off-line RTT measurements to divide the target set amongst the crawlers, however the link structure between pages can only be obtained once all the targets have been downloaded.

Exposto et al. [21] begin with a set of previously downloaded targets and crawler/target latency measurements obtained using the Traceroute[10] tool. From this data, two graphs are generated (Figure 2.2): an

[3] In New York Photo Gallery, 'New York Photo Gallery' is the anchor

[4] Exposto et al were only extracting academic institution names from the URLs, so if a domain was not that of an academic institution, it would not yield any results at all.

[5] A set of known websites relating to a set of city/state pairs is used to train a Naive-Based Bayes Classifier[45]



a. Page graph **b**. IP RTT graph

c. IP link graph **d**. Combined graph

Figure 2.2: Depicts the partitioning algorithm used in [21]

IP RTT graph (with vertices as crawlers/targets and weighted edges denoting the RTT between them) and an IP link graph (with web hosts as vertices and weighted edges denoting the number of links between them). The two graphs are combined into one, and using a partitioning algorithm [24] the combined graph is split up amongst available crawlers so as to both minimise RTT between crawlers and web pages and to minimize out-links between partitions. This same approach is used in [14], but with geographic distance instead of RTT.

Exposto et al. show that by considering link topology between web hosts, one can decrease the exchange costs (With 19 partitions, the total exchange time is 18 minutes with the Link/RTT based partitioning compared to 20 minutes with a Hash based assignment).

The major disadvantages in using link topologies this way is that (1) the partitioning is computationally expensive and has to be done on a central server, and (2) the crawling system must have previously download all pages in order to accurately determine link topologies.

2.2.4 RTT Based

The final assignment strategy that we will discuss is the RTT based or network aware assignment strategy which considers latency measurements between crawlers and targets when assigning a target to a crawler. Loo et al.[7] study the effects of network locality on partitioning schemes. They measured the RTT between a target set and a set of 70 crawlers using the Ping[9] utility. Given a target and the latency between it and the set of crawlers, they consider the following strategies for which to assign a crawler to a target: (1) **Optimal**: Pick the crawler with the lowest latency to the target. (2) **Random**: Pick a random crawler from the set. (3) **Worst**: Pick the crawler with the highest latency value. (5) **Hostname Hash**: Pick a crawler using a distributed hash table.

Although they do not show how these different assignments affect crawling performance, they show how these assignments can be used to assign crawlers to targets in order to minimize latency (essentially answering the question: is the result of a previous latency measurement between a crawler and a target correlated to the value of a future latency measurement between the crawler and the target?).
They note that the Optimal assignment can select a target within a $50ms$ latency for 90% of the crawlers; the performance of the Hash assignment was closely related to that of the Random assignment (choosing

a target within $200ms$ for 90% of the crawlers). Note that Loo et al.[7] are using these results to develop a set of heuristics to use in their hash based assignment strategy and do not base the overall assignment on RTT measurements.

As noted in Section 2.2.3, Exposto et al.[21] consider RTT when partitioning the target set; their evaluation does show some download speed gains when compared to hash based partitioning (using 27 partitions to crawl $\approx$ 17million, there is roughly a 75% decrease in download time when using an RTT/Link based assignment over a hash based assignment). It also shows some improvements in communication overhead (using 27 partitions, total exchange time is 16 minutes for the RTT/Link based assignment versus 19 minutes for a hash based assignment). However as link structure is also factored into their assignment strategy, it is unclear just how much RTT affects the total download and exchange times, also these performance gains vary between approximately 0% and 75% improvement depending on the number of partitions.

Rose et al.[22] use offline RTT measurements determined using the King method[25][6] in a latency based assignment of crawlers to RSS feeds. They note that by using an RTT based assignment in a crawl of 34092 RSS feeds located accross 481 unique IP addresses using 11 crawlers, they achieve an 18% reduction in the median crawl time when compared to a non RTT aware crawling assignment. Although the latency measures used for the assignments are approximated, it has been shown that Kings has a 75th percentile error of 20% of the true latency[25].

In [13], Tongchim et al. use two crawlers: one in Thailand and one in Japan to crawl the Japanease/Thai web (Japanease(Thai) web is assumed to be all the domains under the .jp(.th) TLD). They determine network proximity by measuring the latency (using the Ping[9] tool) from each of the crawlers to the targets and choose the crawler with the lowest latency to assign the crawler to a target. This is achieved through active probing: When a crawler sees a new domain it measures its latency to it and asks the other crawler to do the same, the crawler which observed the lowest latency to that domain is assigned all URLs under it (this is called active probing). They observe that after a time of 250 minutes, the Network Proximity based assignment crawled 51000 web pages and a Hash based assignment crawled 50000. This is only a small increase in favour of the Network Proximity based assignment, however we must note that only 2 crawlers were used in the system, and we would anticipate the difference between the two strategies to increase if the number of crawlers was increased.

2.2.5 Discussion

Early research focused on hash based assignment strategies, mainly because they are easy to implement, intuitive, provide good balancing properties and in the case of UbiCrawler[4] also provide good contravariance properties. However, there does not appear to exist an effective way of integrating network proximity into a hash based assignment strategy.

For this reason, more recent efforts have focused on geographic based assignment strategies as close geographic distance appears to be strongly correlated with rapid download speeds and web pages tend to reference other pages in the same geographic location thus reducing communication overhead. These techniques are promising and show significant performance increases over hash based approaches, however we are not aware of any publications which discuss issues such as contravariance and balancing in geographic assignment strategies.

Considering link topology in crawler assignments is a good way of reducing communication between crawlers (as the web can be partitioned in such a way as to keep out-links between partitions to a minimum) but has the major disadvantage of requiring the set of web pages to be downloaded prior to the partitioning and thus limits its usefulness.

[6]The reader is referred to Section 2.4.2 for a definition of the Kings method

Network latency based assignments also shows promise over hash based assignments, but they require either (1) an all pairs measurement between targets and crawlers to be performed before the crawl which makes them unsuited for long crawls or (2) active probing which must be performed during the crawl as new domain names are seen. In both cases measuring ICMP Ping latencies requires additional communication with web servers.

Although it is noted that Geographic distance is strongly correlated to network latency (see Section 2.4.3), we are unaware of any research that compares latency based assignments to geographic based assignments.

2.3 Network Coordinate Systems

Network Coordinate Systems (NCs) are a novel way of assigning a set of Euclidean coordinates to a set of network hosts such that the distance between two hosts in the space accurately predicts the latency between them. In this section we will describe various NC systems, focusing on Vivaldi in particular and discuss issues such as accuracy and stability.

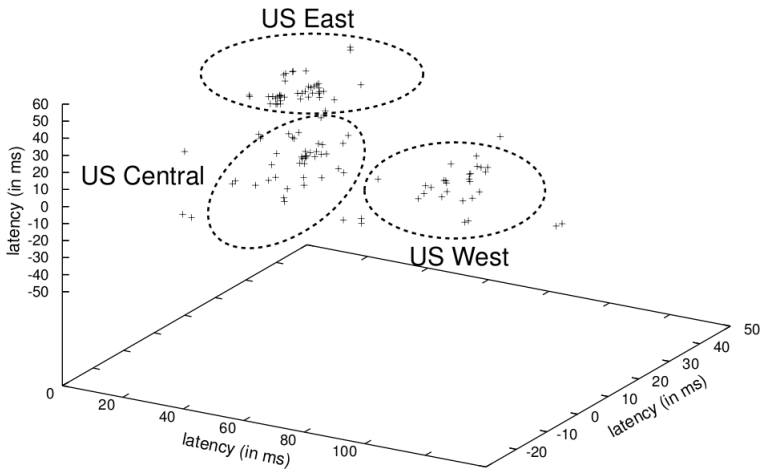


Figure 2.3: A network coordinate space on PlanetLab.[34]

The Network coordinate systems which we will discuss work along a similar principal: each node in the system holds a representation of a geometric space which maps coordinates to nodes in the system. Each node performs latency measurements in order to determine the distance between itself and its neighbours. Using the observed latency, the node then adjusts its coordinate using a minimisation technique.

Network Coordinate systems can be grouped into two separate categories: (1) Centralised coordinate systems, where nodes rely on a set of specialised hosts (landmarks) to determine their own coordinates. (2) Decentralised coordinate systems which are fully distributed systems and require no distinguished hosts or specific infrastructure.

It is noted that these types of systems are not without error, as network latencies often violate the triangle inequality. Zheng et al[43] have found around 18% of triangle violations in an overlay of 399 PlanetLab[35] nodes.

2.3.1 Centralised Network Coordinates

In early Network coordinate systems such as NPS[30] and GNP[28], nodes used a set of designated landmarks upon which to compute their coordinates. These were shown to be accurate methods of estimating the latency between two Internet hosts. However, GNP is a centralised system wherein all hosts rely on a small set of landmark hosts to accurately calculate their coordinates, as such it is not scalable because the landmarks are a central point of failure and can become overloaded when large numbers of hosts are used. Furthermore, inaccuracy in one of the landmark coordinates would have repercussions throughout the rest of the system, also the landmarks have to be carefully placed in the network so to best provide an optimal set of measurements.

In NPS, a node can use any other node in the system upon which to base its latency measurement. As such, NPS is a improvement over GNP as there is much less reliance on the set of landmark nodes, however it is still not completely decentralised because when a node bootstraps, it must contact a membership node to receive a set of landmarks. Once it has computed its coordinate it may itself act as a landmark for other joining nodes. But even though nodes are not directly reliant on the landmarks, a failure in one of them or a membership server will still have repercussions throughout the rest of the overlay.

2.3.2 Decentralised Network Coordinates

We will now discuss fully decentralised network coordinate systems that do not rely on a set of central nodes and as such are much more fault tolerant.

PIC

PIC [31] is an attempt to develop a fully decentralised network coordinate system using the same mathematical principals used in GNP and NPS.
PIC maps each node to a set of coordinates in d-dimensional space. When a node n joins the system, It measures its latency to a set L of landmarks, where $|L| > d$.
Once n has measured its latency to the set L, it computes its coordinate(c_H) by minimising the following equation using the Simplex downhill algorithm[32].

$$f_{obj2}(c_H) = \sum_{i \in \{1..N\}} \varepsilon(d_{HL_i}, \hat{d}_{HL_i}) \tag{2.1}$$

Where $d_{H_iH_j}$ is the measured RTT (using ICMP pings) between host i and host j and $\hat{d}_{L_iL_j}$ is the distance between host i and host j as measured in the coordinate system. ε is the squared relative error function:

$$\varepsilon(d_{H_1H_2}, \hat{d}_{H_1H_2}) = (d_{H_1H_2} - \hat{d}_{H_1H_2})^2 \tag{2.2}$$

When PIC bootstraps, inter node RTT distances between the set of all nodes N in the system are computed. The global minimisation algorithm computes the set of all coordinates in N using the Simplex downhill algorithm[32] to solve a global version of Equation 2.1.

A new node n can choose any subset L from the set of all nodes N in the system with known coordinates. In PIC, L is a combination of "close" nodes and randomly chosen nodes. The authors argue that this strategy provides a good balance of estimation accuracy for short distances (provided by the close nodes) and for long distances (provided by the random nodes).
Alternate strategies (such as choosing the set of closest landmarks or choosing landmarks randomly) for selecting L are presented in [31], as well as ways of selecting the set of closest nodes to n, however these will not be discussed here.

In [31], PIC (using $|L| = 16$ landmarks, $d = 8$ dimensions and 40,000 nodes) is compared to GNP (with optimal landmark positions determined offline). PIC achieves roughly the same accuracy as GNP (90th percentile relative error of roughly 0.37 for GNP versus 0.38 for PIC). For short distance estimations, PIC greatly outperforms GNP: 95% of estimations have a relative error of below 50% for PIC versus 35% for GNP.

PIC is a fully decentralised, robust and accurate network coordinate system. However, It remains to be shown how quickly PIC can adapt to volatile network topologies. As it uses the Simplex minimisation procedure to determine network coordinates after a full set of measurements have been taken, this causes it to react very quickly to changes in the network topology.

Vivaldi

Vivaldi [33] uses a different approach to those previously discussed: It computes network coordinates using a spring relaxation technique.
Nodes are modelled in a d-dimensional Euclidean space as a physical mass spring system: nodes are point masses interconnected by springs. The Euclidean distance between a pair of nodes, or the length of the spring interconnecting them is an estimation of the latency between the two. The errors in the coordinate space are characterized by the following squared error function:

$$E = \sum_i \sum_j (L_{ij} - ||x_i - x_j||)^2$$

Where x_i and x_j are the coordinates assigned to nodes i and j respectively, and L_{ij} is the true latency between nodes i and j.
This squared error function is equivalent to the energy in a physical mass spring system, it can be minimized by simulating the movements of the nodes under spring energy. Finding a low-energy state of the spring system is equivalent to finding a low error estimation for the RTT distances.
Vivaldi, like PIC is a completely decentralized implementation: each node runs an identical algorithm, and calculates its own coordinates based on perceived RTT measurements and the coordinates of its surrounding nodes.
When node i with coordinates x_i receives a new RTT measurement L_{ij} to node j with coordinates x_j, it updates its own coordinates using the following:

$$x_i = x_i + \delta \times (rtt - ||x_i - x_j||) \times u(x_i - x_j)$$

The unit vector $u(x_i - x_j)$ gives the direction of the force exerted on i by j. The value of the timestep δ determines by how much the coordinate should change at each iteration. In order to ensure that node coordinates converge quickly and accurately, Vivaldi uses an adaptive timestep:

$$\delta = c_c \times \frac{e_l}{e_i - ej}$$

Where c_c is a constant fraction (c_c <1), e_i is the error estimate of the local node i, and e_j is the error estimate of the remote node j. This value of δ ensures that an accurate node sampling an inaccurate node will not change its coordinates too much, and an inaccurate node sampling an accurate node will change its coordinates a lot. It is shown in [33] that a c_c value of 0.25 yields quick error reduction and low oscillation.
In order for a node to estimate the accuracy of its own coordinates, it keeps a record of previously observed RTT values (L_{ij}) and compares them to previously estimated RTTs($||x_i - x_j||$). The error values, e are an average of recently observed relative errors.

Accuracy of Vivaldi Vivaldi competes in accuracy with the other NCs previously discussed, in [33], Vivaldi is shown to have a similar accuracy to GNP. More precisely, coordinates in Vivaldi exhibit median errors between 5-10% using 3-5 dimensions on a wide area network.
In any NC system, a small set of abnormally high or low latency measurements can completely distort the system as a whole. Ledlie et al[42] presented a method of filtering latency measurements using a Moving Percentile(MP) filter to improve accuracy. It is a statistical filter that uses recently observed latency measurements to predict the value of the next measurement. Their results show that using the MP filter with Vivaldi in a 270 node system, 14% of nodes experienced a 95th percentile relative error greater than 1; without it 62% did.

Stability of Vivaldi Although in [31], Costa et al. state that Vivaldi nodes can become confused when a large number of nodes join the overlay, in [33], it is shown that using the δ adaptive timestep, a previously stable set of 200 nodes can take on an extra 200 new nodes without destabilizing the pre-established order; using the adaptive timestep, the extended system can converge to an accurate set of coordinates 4 times faster(within $80sec$) than with a constant timestep.
Ledlie et al [42] present a heuristic called ENERGY to increase stability in coordinates. It is a heuristic that compares the previous set of observed latencies to the current set and only considers the new measurements if they are over a certain distance from the previous ones. Using this heuristic, Ledlie et al achieve a much smoother metric space.

Vivaldi With Changing Network Topologies Vivaldi is particularly well suited to rapidly changing network topologies: Vivaldi can use measurements of traffic sent out from the application that is using it. Furthermore, as opposed to the other NC systems previously discussed, a node does not have to sample all of its neighbours in order to adjust its coordinates.
In another experiment, Dabek et al. construct a 100 node Vivaldi system in $d = 6$ dimensional space with each node sampling 32 randomly chosen neighbours; after the coordinates stabilize they increase one of the inter-node lengths by a factor of 10, it is shown that the system can quickly re-converge to a stable state(within 20 seconds); after 180 seconds, the link is returned to its original size and the system quickly re-converges to its previous accuracy level. This is remarkably better than NPS, which was shown to take 1 hour to converge to a stable state after a drastic network topology change.

Vivaldi Height Vector We would like to note an interesting feature of the Vivaldi algorithm[33]: Height Vectors.
Conceptually, nodes have two properties: an entry point, and a core. Previously, we were considering inter-node distances as the time it would take for a package to propagate between the entry point of two nodes (using network level Pings). By adding a positive height value to each nodes coordinates, we can consider the inter node latency as the time it takes for a message to travel from the source nodes core, down its height, across the inter node link and up the destination nodes height to its core.
The vector does not affect the inter node distance when two nodes have the same height; however, when a node finds itself too close to all of its neighbours in the standard Euclidean model, it has nowhere to go without adding an extra dimension to the space. With height vectors, a node in this situation can increase its height vector, moving the same distance away from all of its neighbours. Similarly, if a node finds itself too far away from all of its neighbours, it can decrease its height, moving it closer to all of its neighbours.

Load Aware Network Coordinates Traditional network coordinates are computed using latency measurements obtained using ICMP echo requests; these kinds of measurements capture pure network latencies along the path between two hosts. Ball et al[37] introduce the concept of *Load Aware Network Coordinates*(LANCs) which are intended to capture the load of hosts in the overlay as well as the latency between them. They are calculated using application level perceived round trip times between hosts using a dedicated TCP connection with user-space timestamping. When a host is overloaded, it will affect the value of

the user space timestamps leading to higher perceived RTTs.
LANCs are used in [37] to compute coordinates for *Content Delivery Servers*; servers that are under high computational load are assigned more distant coordinates, moving them further away in the space. In [37], an overlay of 6 nodes is set up using load aware RTT measurements with a 100Kb payload; one node is designated as the content server and the remaining 5 as clients, when the server is put under theoretical load (using a CPU bound process), it is shown that it accurately moves away from all client nodes, after a period of 10 minutes, the process is terminated and the server node returns to its original position in the NC space.

Discussion

Decentralized network coordinate systems have shown to be accurate ways of estimating inter-host latencies without relying on a set of central nodes. PIC nodes compute their coordinates using an arbitrary number of landmarks and are therefore much more robust to noise, PIC is shown to have an accuracy comparable to that of GNP. However, it does not seem very well suited to rapidly changing network topologies as nodes only run the minimisation process after they have obtained a full set of measurements to their landmarks.
Vivaldi nodes can also use an arbitrary number of landmarks to compute their coordinates, Vivaldi has been shown to be stable and have an accuracy comparable to that of GNP. Furthermore, a node does not require to have a full set of measurements in order to adjust its coordinates and hence can react quickly to changes in the network. Also, it has been shown that proxy network coordinates(see Section 2.3.3) can be integrated accurately into a Vivaldi overlay.

We will use a Vivaldi overlay in our distributed crawling system because of its simplicity, versatility and ability to react quickly to network changes. And we will use user space time-stamping for our latency values to give our crawlers LANCs so that a crawler which is under high computational load will move further away in the NC space, and vice versa.

2.3.3 Proxy Network Coordinates

NC systems are a powerful and versatile tool for modeling inter-node latencies in a distributed system; However, the systems described in the previous section require that all nodes in the network actively participate in the measurements. Ledlie et al [44] introduce the concept of proxy network coordinates: a method that allows a set of nodes in an overlay to determine coordinates for a set of oblivious nodes (i.e. nodes that do not run any NC software) in the same coordinate space.
In [44], overlay nodes use application level pings to determine their coordinates, oblivious node coordi-

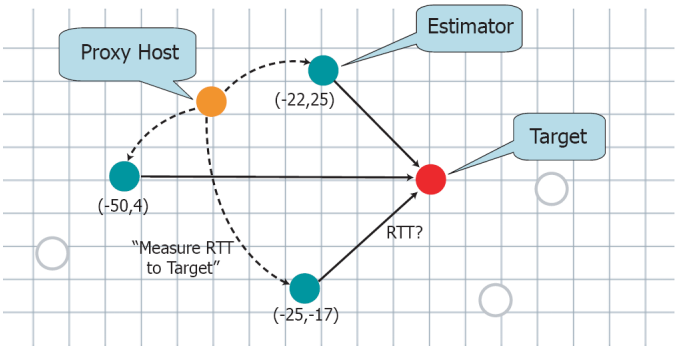


Figure 2.4: Computing a Proxy Network Coordinate[44]

nates are estimated using measurements from different vantage points:

When an overlay node (referred to as proxy node) wishes to determine the coordinates of an oblivious node (referred to as the target), it contacts a number of other nodes in the overlay network (referred to as estimators). The estimators measure their round trip time to the target and return the value along with their coordinate and their confidence in their coordinate to the proxy node. Figure 2.4 illustrates this process. The proxy network coordinates are then computed by the proxy host using the standard Vivaldi algorithm outlined in Section 2.3.2.
The only difference between Proxy Network Coordinates and standard Network coordinates is that the measurements only occur in one direction; Experimentally, this has not been shown to be an issue, Ledlie et al[44] show that the relative error is under 50% for over 80% of the measurements for 1591 targets using an overlay of 166 nodes. Furthermore it is shown that when a proxy network coordinate is instantiated in an overlay of 166 nodes, it converges to a set of coordinates with a 95th percentile relative error of between 0.1 and 0.25 within 10 minutes. The convergence time of a proxy network coordinate is comparable to that of an overlay node coordinate.

2.3.4 Message Routing in NC Spaces

In this section we will present a set of message routing techniques that are well suited for communication in NC spaces. Although these have not been used in our implementation, we could envisage applications for them in further work.

θ-Routing

θ-Routing makes the assumption that each node has a set of coordinates in a Euclidean space and that there is no maximum communication range between nodes, as such it is particularly well suited for message routing in network coordinate spaces.
In order for a source node n_s to route a message to a destination node n_d using θ-Routing, it must first construct a θ-graph[41] spanner: The space around the node n_s is divided into a set of $\frac{2\pi}{\theta}$ sectors of angle θ. For each sector, node n_s adds its closest neighbour n_c in that sector to its routing table.
Messages from n_s to n_d are routed through n_s's closest neighbour n_c that is in the same sector as the destination node n_d. Similarly, the closest node n_c will then route the message to its closest neighbour that is in the same sector as the target n_d. This process is repeated until the message reaches the destination.
The θ-graph spanner has the property that for any two nodes n_s and n_d, the length of the routing path (n_s, n_d) is no more than a constant factor times the Euclidean distance $|n_s n_d|$ [39]. Hence the hop count between n_s and n_d will be proportional to the diameter of the network.
The advantages of this method are that each node only needs to keep a small routing table, and the routing paths are quite efficient(because they closely follow the line between the source and target). The main issue with θ-Routing is that when the NC system is scaled up, the minimum hop count will increase linearly relative to the diameter of the network.

$\hat{\theta}$-Routing

The linear hop-count of θ-Routing is improved in [40] where messages can traverse long distances in a single hop. Yassin and Peleg propose a method called scaled θ-Routing ($\hat{\theta}$-Routing) where the θ-graph spanner is further split up by exponentially expanding rings:
Nodes split their surrounding space into a set of $\frac{2\pi}{\theta}$ sectors and r rings. The area delimited by the intersection of a sector and a ring is termed Zone. Nodes add the nearest neighbours that they know in each zone to their routing table. Figure 2.5 illustrates this process.
To forward a message, node n_s calculates the zone of the target node n_d and greedily forwards it to the closest node in the furthest zone that does not exceed the target's zone. $\hat{\theta}$-Routing improves over θ-Routing by reducing the hop count to logarithmic complexity, but it requires each node to maintain a larger routing table.
In order for a node to accurately maintain its routing table, Ledlie et al [39] assume that a gossip mechanism exists that allows nodes to exchange routing tables. To obtain an accurate routing table, a node queries its

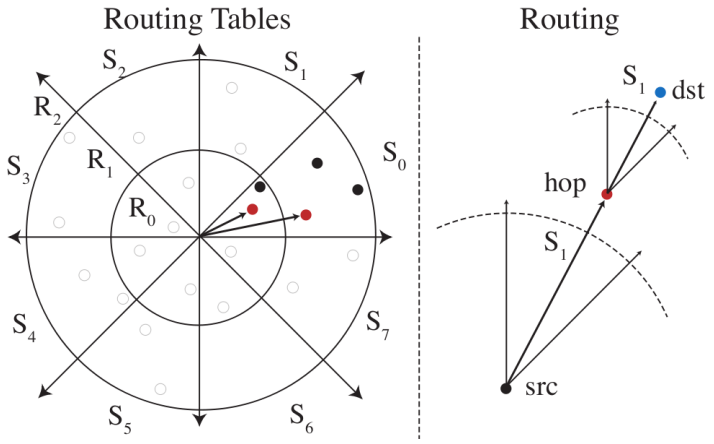


Figure 2.5: $\hat{\theta}$-Routing[39]

neighbours for its own coordinates. This special query is flagged so that each node along the routing path attaches its routing table to the message. The intuition is that when the message comes back to the node, it is likely to contain the routing table of a nearby neighbour that is similar to what its own routing table should be.

Finding nearest neighbours in $\hat{\theta}$-Routing

It is clear that $\hat{\theta}$-Routing is well suited to find a node close to a given target, because when a node routes a message, it calculates the sector in which the target is situated, and forwards the message to a node in that same sector(hence a close node), however there may exist a node that is closer to the target and not in the sector(take for example the case were the target node is on a boundary between two sectors).
Using greedy-by-distance routing, it is shown that the system is able to correctly find nearest nodes. In an experimental evaluation[39], a 1740 node system was created with 10% of them designated as non-participating targets. Employing a θ-graph spanner with 8 rings and 6 sectors, they found that all queries found the nearest or second nearest node to the target, but because of errors in the underlying NC space, the target node was often not the closest node. To correct this, each node along the routing path would measure its latency to the target, the node with the smallest latency is then designated as the closest node. The results showed, that using this technique, 90% of the queries yielded a target within a 30ms latency difference to the actual closest node.
Under volatile network conditions, it was shown that the 80th percentile latency penalty was between $15ms$ and $20ms$ using the gossiping technique.

2.4 Network Metrics

In this section we wish to present the various methods used to measure latency and geographic distance in the NC systems and the crawler assignment strategies previously discussed. We will then briefly discuss the correlations between them.

2.4.1 Measuring Geographic Distance

There exist a multitude of services on the Internet that provide a mapping between IP addresses and geographic information[17, 16, 18]. These services mainly parse Whois registration data to derive longitude-latitude from registrar address data (City, Zip Code, Country...). In our experience, this method yields a small set of distinct latitude and longitudes pairs relative to the number of distinct IPs. The reason being

is that multiple IP addresses will be registered with the same ISP, which will register a large number of IP addresses to a small number of distinct addresses. That said, if two hosts share a common latitude and longitude, chances are they are managed by the same ISP, and are on the same or a near by AS. Once the latitude and longitude have been obtained for a pair of Internet hosts, their geographical distance can be calculated using spherical coordinates on the earth.
As for the accuracy of these services, that of NetGeo[18] is unknown as it does not appear to be actively maintained. hostip.info[17] is an open project, [8] estimate the accuracy to be around 50% on a city level for the US; no estimated level of accuracy is given on the hostip.info website, however they do note that their solution is not as accurate as GeoIP. GeoIP[16] seems to be the most accurate service, there are two versions available: a commercial and a free version. For the free version, stated accuracy is over 99.5% on a country level and 79% on a city level for the US within a 25 mile radius. For the commercial version, accuracy is over 99.8% on a country level and 83% on a city level for the US within a 25 mile radius.

2.4.2 Measuring Latency

There are various ways of determining Round Trip Time(RTT) between two Internet hosts. The most common ways are to use either the Unix Ping[9] utility or the Traceroute[10] utility. Although these tools are widely used and accurate, Traceroute sends out TTL restricted UDP packets which might be blocked by some routers. the Ping utility uses ICMP ECHO requests, however the ICMP replies are sometimes blocked or manipulated by ISPs.
In some of the research that we have discussed so far, authors use the King[25] method which can estimate inter-host latencies with a relatively high accuracy[7].

> King is based on two simple observations: most end hosts in the Internet are located close to their DNS name servers, and recursive DNS queries can be used to measure the latency between pairs of DNS servers. Thus, it should be possible to estimate the latency between two end hosts by locating nearby name servers and measuring the latency between them.
> To measure the latency between two name servers, King issues a recursive DNS query to one name server, requesting it to resolve a name belonging to a domain for which the other server is authoritative.
> King measures the amount of time it takes to issue a recursive query to name server A for the name xyz.foo.bar, given that name server B is an authoritative server for the domain foo.bar. Name server A resolves the query by interrogating name server B, and forwarding its reply back to the client. Next, King measures the latency between the client and the name server A, either by using an ICMP ping, or perhaps by issuing an iterative DNS query. By subtracting these two latencies, King produces its estimate of the latency between the two servers.
>
> *Definition of the King method for DNS based latency[25].*

2.4.3 Correlation between Metrics

Correlation Between Latency and Geographic Distance Subramanian et al. [27] suggest a strong correlation between linearized distance[8] and RTT. more precisely, they make three important observations: (1) At lower values of linearized distance, the correlation between distance and RTT is stronger. (2) linearized distance along a path implies a minimum end-to-end RTT. (3) Linearized distance and RTT are more strongly correlated than end-to-end distance and RTT. The analysis of their data set shows that 90% of nodes within 5 ms RTT are located within a radius of 50 km and 90% of nodes within 10 ms RTT are located within a radius of 300 km.
Correlation between RTT and geographic region is also noted in [19]. They measure the RTT between a set of targets in San Fransicco, USA, and monitor nodes in both San Diego USA and Hamburg Germany.

[7] It has been shown that Kings has a 75th percentile error of 20% of the true latency

[8] Linearized distance is obtained using the Traceroute tool and the GeoIP database. It is the sum of the geographical distances separating the routers in the routing path between two Internet hosts.

It is shown that the distribution of RTTs observed from San Diego Peaks at a relatively small RTT value (50ms), whereas the RTTs observed from Hamburg Germany Peak at a much higher value (350ms).

Correlation Between Latency and Throughput In [21], RTT measured using the Unix Traceroute[10] is used in the optimal assignment of crawlers to targets: they state that available bandwidth between a crawler and a target is bounded by RTT but do not provide an explicit proof as to why. In [22], they show that a latency aware assignment outperforms a non-latency aware assignment. Intuitively, this leads us to believe that RTT is indicative of throughput.

Figure 2.6 illustrates some general patterns of response time and throughput as a function of load along a

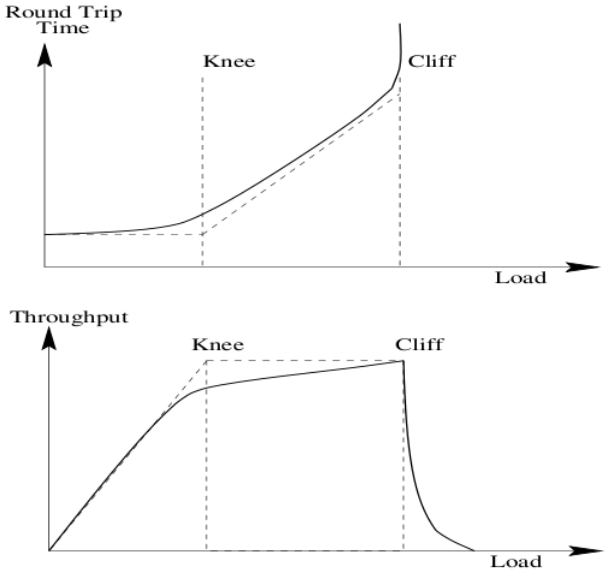


Figure 2.6: Throughput and RTT as a function of Network load [26]

given network path. Comparing the RTT delay graph to the Throughput graph, we observe that whilst the load remains below network capacity(the knee), throughput increases whilst RTT only increases slightly. Once the load starts to reach the maximum capacity of the network(past the knee), the RTT begins increasing linearly. Once the load reaches the cliff, the network is termed congested, the throughput drops and the RTT increases drastically (possibly due to packets getting dropped). [26] goes on to describe a congestion control scheme to operate the network at the knee, but we will not discuss it here.

These observations are further supported in a study of 14,218 TCP connections over 737 Internet paths in [23] where they find that RTT is sensitive to load presented by a TCP connection, however they note that the correlation between the load presented by a single TCP connection and RTT is not strong enough unless the TCP connection consumes a large fraction of the available bandwidth.

Although the work in [23, 26] is geared towards congestion avoidance techniques(CATs), we can conclude that low RTT values indicate available bandwidth along a network path (hence, faster download times), similarly high RTT values indicate little available bandwidth(hence, slower download times). We must emphasise however that even if available bandwidth appears to be ”sensitive” to latency, there is no empirical proof that RTT is strongly correlated to available bandwidth.

We would assume that ICMP Pings are not strongly correlated to download speeds or throughput because the default size of an ICMP Ping packet is 64 Bytes relative to a few hundred Kilobytes for the average web page, as such Ping Packets are less likely to get dropped by routers along a congested path because they are transmitted using far fewer frames.

2.5 Technology

We will now briefly discuss the technology which we have used in our implementation.

2.5.1 PlanetLab

PlanetLab[35] is a global research test bed for distributed systems research. It is composed of 1011 nodes at 475 sites. It is used in industry as well as academia for both short and long running experiments. Users are granted super-user access to a set of virtual machines running on all nodes.
Because of the control it gives its users it is a very useful tool however it is a very unreliable and unpredictable service: Nodes can be rebooted without any prior notice and can undergo very sudden and drastic changes in load. Furthermore there is a limit to the amount of resources and bandwidth that any one user can consume on each node, if a user exceeds that limit all of their processes are terminated.

muSSH

muSSH[55] is a shell script that allows the execution of a command or script over ssh on multiple hosts with one command. When possible mussh will use ssh-agent and RSA/DSA keys to minimise the need to enter passwords more than once.
We used muSSH to execute commands on multiple PlanetLab nodes concurrently and to start and stop our crawler.

2.5.2 Programming Languages

Perl

Perl is a high-level, general-purpose, interpreted, dynamic scripting language. We used it extensively to process and analyse the huge amounts of logs generated by our system.

Java

We used Java 1.6 to implement our crawling system because Pyxida was implemented in Java and it also handles multi-threading and communication well. Care must be taken however in the versioning when communicating between two Java applications, because if the two applications are using different versions of Java then this can cause Serialization errors. To avoid this, we installed our own version of Java on all of the PlanetLab nodes used.

Apache Ant

Apache Ant[58] is a software tool for automating the Java build process. It is similar to make but requires the Java platform. We used Ant to automate our compilation process and also integrated our distribution scripts into it so that with one command we could build our software and distribute it to any number of PlanetLab nodes.

2.5.3 Supporting Libraries

HTML Cleaner

HtmlCleaner[59] is open-source HTML parser written in Java. HtmlCleaner handles broken and malformed HTML code well because it reorders individual elements and produces well-formed XML. By default, it follows similar rules that most web browsers use in order to create Document Object Model. However, user may provide custom tag and rule set for tag filtering and balancing.

Jakarta Commons HttpClient

The Jakarta Commons HttpClient[56] component provides an efficient, up-to-date, and feature-rich package implementing the client side of the most recent HTTP standards and recommendations.
Because writing a robust HTTP client is a difficult task, we used it as the main component in our Downloader.

Ehcache

Ehcache[54] is a widely used Java distributed cache for general purpose caching, Java EE and light-weight containers. It is a very feature rich and very powerful cache implementation. We used it mainly to store the vast amounts of URLs seen during a crawl.

Pyxida

Pyxida[57] is an open source implementation of the Vivaldi Network coordinate algorithm. Each node must have its own instantiation of the Pyxida *NCClient* class which holds a representation of the network coordinate space.

A node x updates its coordinate using node y by passing the observed latency between x and y as well as y's coordinate and error to the *NCClient* class. Pyxida will then update x's the coordinate using the Vivaldi algorithm.

There are three important parameters which can be tweaked in Pyxida: the *Minimum Window Size*, *Maximum Window Size* and the *Smoothing Value*. Pyxida stores the set of previously observed latencies between any two hosts in the overlay in the *Window* (see Section 2.3.2), its minimum size is the minimum number of latencies that a host x needs to see for host y before it can consider it for its coordinate computation. The maximum size is the maximum number of latencies that the window can contain. The Smoothing value is the sample percentile of the window which is taken as the latency between host x and host y when host x updates its coordinate. Larger sizes of the window allow for a more stable coordinate system and lower values of the smoothing percentile allow the system to achieve a low error state when there is a high variance in the latency samples. We had to modify Pyxida slightly so that these parameters could be specified when the *NCClient* class was instantiated.

Chapter 3

Design

A fundamental aspect of any crawler, whether it be a single crawler or a distributed system, is the time it takes to download an individual web page. In the case of a single crawler, there is no room for optimising network proximity between crawlers and targets and research in this area is mainly focused on developing innovative ways to store and process large amounts of data to stream-line downloads. In distributed systems, there is much more scope to optimise network proximity, because in a system comprised of multiple crawlers, there is surely one which will be able download a web page in less time than the others.

In Section 2.2, we discussed various strategies [21, 22, 14] which use network metrics when assigning targets to crawlers. In these strategies, two very different metrics are used: (1) Geographic distance (obtained using Whois data) between a crawler and a target and (2) the return trip time (RTT) of ICMP pings between a crawler and a target.

The results are promising: geographic distance was shown to perform significantly better than a hash based assignment in terms of download time[1] in [14]; and in [22] it was shown that by using an RTT based assignment, one could also gain considerable decreases in download time[2]. It is important to note however that in both of the above, the latency measurements were performed off-line, before the crawl, and then used to direct the crawl.

A second important question which is not answered is how much correlation there is between RTT/Geographic distance and bandwidth or throughput along a network path (which is arguably the deciding factor for download times). Geographic distance has been shown to be correlated to an extent with RTT[19], and RTT in turn has been shown to have some correlation with throughput[23]. But although correlations have been shown, no correlation between Geographic distance and throughput has been shown, and the correlation between RTT and throughput has not been shown to be strong or very reliable[23]. Our aim is not to explore correlations between metrics but to find a way of best determining assignments that decrease download times as much as possible.

Network coordinate(NC) algorithms such as Vivaldi are able to predict RTT between Internet hosts well; Vivaldi can also be used to compute proxy coordinates for oblivious hosts (not running any measurement software) to a high degree of accuracy.

NC algorithms are well suited for our goals of developing a download time predicting assignment strategy; We will build a distributed crawling system using NCs: our crawlers will cooperate to compute their coordinates and proxy coordinates for oblivious web servers. We will design our system such that the distance between a target web server and a crawler in the NC space gives an accurate estimate of the time that it would take for the crawler to download a page from the web server. Furthermore because these systems

[1] In a system of 32 crawlers the average download time was 1000 seconds using geographic based assignment compared to 57000 seconds when using a hash based assignment.

[2] Rose et al[22] achieved an 18% reduction in the median crawl time over a random assignment when using an RTT metric.

can adapt rapidly to changes in network topology, it should enable our crawler to detect and compensate for changes in the performance of the underlying nodes.

A very attractive aspect of network coordinate systems is their ability to use existing network traffic to compute coordinates. We would like to carry this feature forward in our design because we do not want to take away valuable bandwidth from crawlers that could otherwise be used to download web pages. In this section, we shall outline the design and architecture of our fully distributed and decentralised crawling system which requires no central point of control, predicts download times "on the fly" and will not require substantially more network bandwidth for communication than a system without network coordinates.

3.1 Design Decisions

In this section, we will justify the major design decisions which have motivated our implementation. We will begin with what brought us to implement a decentralised over a centralised system. We will then discuss how we have used Network Coordinates to predict download times between a crawler and a web server. Finally, we will discuss some crucial aspects of the design of the distributed crawler built around the network coordinate system.

3.1.1 Decentralised Control Versus Centralised Control

One of the first things that we had to decide was whether to implement a fully decentralised crawling system, with no central point of control. Or a managed system with one or more control nodes which would direct the crawl. A system with one or more points of control would have been easier to implement, however, it would not have the fault tolerance and scalability properties of a fully decentralised system.
The deciding factor in this decision was the environment on which our software would run: PlanetLab, a global research test bed for distributed systems research. PlanetLab users are given super user access to a set of virtual machines co-located on servers (nodes) all around the world. It has many advantages, however reliability is not one of them: nodes can be rebooted without prior notice and can see sudden and drastic changes in load. It was crucial that our crawler be robust against such phenomena. As such we chose to implement a fully decentralised system, with no central point of control, thus if single node fails, it would not affect our system overall.

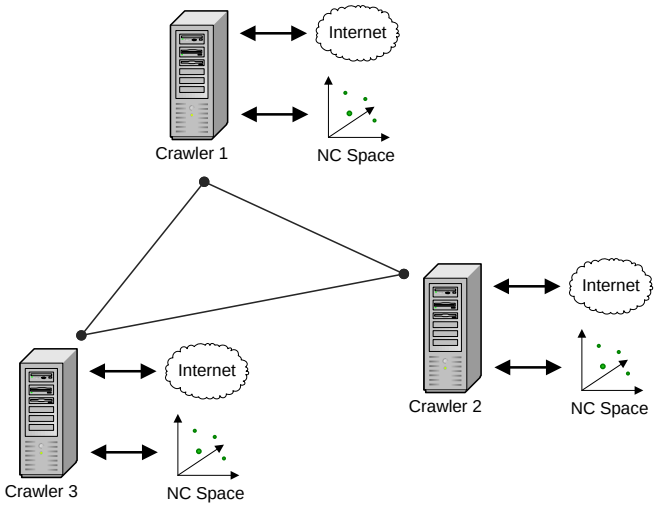


Figure 3.1: Depicts our decentralized crawler design. Each crawler downloads web pages and is responsible for managing a set of proxy coordinates for web servers.

3.1.2 Network Coordinate System

We have embedded our crawlers and target web servers into the same network coordinate space. The crawlers participate actively in measurements to compute their coordinates, whereas web servers are assigned proxy coordinates computed through estimation similarly to what is described in [44].
Because the load of a system affects its ability to download and process content from the Internet, we use load aware network coordinates(LANC)[37] for the crawlers - which also gives our system some load balancing properties. For the web server coordinates, we use a very different metric: download time.

Overlay Coordinates

We used the Pyxida library to assign load aware network coordinates to each crawler in our system: Crawlers periodically exchange their coordinates through gossip messages. The return trip time (RTT) of these gossip messages is used by each crawler to adjust its distance in the NC space relative to its neighbours. Because we use application level time stamping for our RTT measurements like in[37], a crawler that is under high computational load should move further away in the space, whereas a crawler that is under low load should move closer.

Proxy Coordinates

We assigned Proxy Coordinates (PC) to target web servers in our system. A crawler that is close (respectively far) to a web server in the NC space should observe fast (respectively slow) download times from that server.

Choosing a Metric Because of this requirement we needed to choose an appropriate metric on which to base our PCs, in our early experimental work we explored the use of three metrics: (1) ICMP Ping RTT time (2) Geographic distances obtained using the GeoIP database[16] (3) Previously observed download times. Evidently the third metric proved to be the best. To see why, consider the following example: We have two crawlers A and B, and wish to determine which one is best suited to download a 1 Gb file from a web server W. W and B are located in the UK whereas A is located in the US. W and A are connected through a high speed optical link whereas B is connected to W through a low speed wireless link. B will be closer to A geographically and will also observe lower ICMP Ping RTT because there will be less routing hops between B and W than between A and W. That said, A will be able to download the file faster because of the bandwidth capacity of the underlying link; assuming no knowledge of the underlying network topology, the only way to check this is by measuring download speeds.
Although this example might seem a bit extreme, it is realistic because massive data centres connected through high speed links are expensive and thus only affordable for large corporations. Smaller organisations wising to crawl the Internet might only have access to a small set of collocated virtual machines running on disparate hardware, as such it is important for them to maximise their available resources.

Proxy Coordinate Computation We will now describe exactly how we used download times for our Proxy Coordinate Computation. In the rest of our discussion, we follow the intuition that download speeds should be measured between a web server and a crawler, so Proxy Coordinates will be computed for web servers. We will have to be careful though, because many large web sites and hosting services often employ thousands of web servers located in different locations to serve content through the same domain name (for example Google.com), so a domain name might resolve to two separate IPs on two successive attempts; to get around this we maintain PCs for web server IPs rather than domain names.
PC computation requires multiple latency samples from different vantage points in order to converge a

coordinate to an acceptable error level. Thus we need to download from a server multiple times using multiple crawlers, we could do this by downloading the same URL over and over again, however this would generate redundant traffic. Instead, to compute a coordinate, a crawler will collect μ URLs hosted on a web server (where $\mu \geq \sigma$ and σ is the mean number of updates required to converge a coordinate to an acceptable error level). Each crawler is responsible for its own set of PCs and will follow the methodology outlined in [44] to compute its PCs: It will periodically send a set of URLs to a randomly chosen neighbour (estimator), the estimator will download the URLs and return the times taken to the requester. The requester will then adjust its set of PCs using the Vivaldi algorithm.
We would like for the LANCs and PCs to marry well: a crawler with high load should move further away from the PCs, whereas one with low load should move closer to them, we will evaluate this feature in Chapter 5.

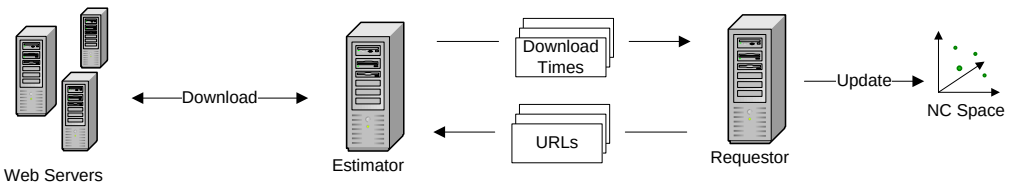


Figure 3.2: Depicts a crawler (requestor) updating its proxy coordinates through estimation.

3.1.3 Crawler System

We will now describe the design of the distributed crawler that uses the network coordinate system for its assignments. We have designed a crawler which is robust and fast enough for our assignments as well as a way of selecting candidate web servers for which we can compute PCs and whose URLs can then be assigned to other crawlers.

Crawler Component

The crawler component contains all of the elements described in Section 3.2.2: (1) A Queue to hold the crawlers workload, (2) A multi-threaded HTTP client to communicate and download content from web servers, (3) A URLSeen data structure to store which URLs have been visited and (4) an HTML parser to extract hyper-links from downloaded HTML content.
The crawler does not rival [47, 49, 11] in terms of download speeds and throughput because we are looking to compare assignment strategies, but it is robust enough to run for an extended period of time and process tens of thousands of URLs. It is also able to determine assignments and communicate these to other crawlers in a robust way.

Assignments

Throughout a crawl we will see thousands of URLs, we needed a way of choosing which URLs crawlers should assign to the overlay, and which ones it should keep for itself. Ways in which this is decided is not widely discussed in the literature in Section 2.2, obviously there is some trade off between communication overhead and speed gained by passing targets to other crawlers. For example, in a geographic assignment, each crawler could assign each new incoming URL to the closest crawler, however this would be very expensive in terms of communication. Furthermore, some web servers may host a handful of web pages whereas others might host millions, if we were looking to crawl both of the servers, it might make more sense to download content from the large web server using a crawler that is optimal because the time gained over the whole set of URLs is considerable. Whereas it might not be as advantageous to download

the small web servers content using an optimal crawler because in the time taken to transfer the work the crawler might have been able just to download the URLs itself.
Thus, the way in which we choose assignment candidates is based on URLs per domain: A crawler bootstraps with a set of seed URLs, it crawls them by adding children to its URLSeen data structure. This is organised in the form of a hash map, where keys are domain names and values are buckets of URLs (this type of organisation for sets of URLs is widely used throughout our implementation). Once the crawler has seen a set of domain names that is over a certain size δ each hosting a number of URLs μ, we assign the whole batch of size $\delta \times \mu$ to other crawlers in the system (δ and μ are configurable parameters).
We only consider web servers which host a large number of distinct URLs for assignment so that the time gained by choosing an optimal crawler is noticeable. Furthermore large web servers are more likely to be able to handle the increase in traffic generated by our crawler.
In addition to our Proxy Coordinate assignment method, we implemented two further assignment strategies: Geographic and Random. In the former, we assign targets to crawlers based on shortest geographic distance; in the latter we assign targets to crawlers randomly.

3.2 Architecture

In this section we will take a brief look at the major components in our system as well as the relationships between them. First we will describe the Network Coordinate System which handles the communications and coordinate computations, secondly we will describe the Crawling System which handles the crawling part of our software.

3.2.1 Network Coordinates & Communication

Communications Manager

In our design, communication is used for three things: (1) To obtain RTT measurements for LANC computation (2) To compute Proxy Network Coordinates and (3) To assign targets to other crawlers. Each crawler maintains a full list of it's neighbours, communication is based on TCP with a message queuing system which is described in Section 4.3.1.

Proxy Coordinate Manager

The proxy coordinate manager handles the computation of the web server coordinates. Each crawler holds a set of URLs for each web server that it maintains a Proxy Coordinate for. The proxy coordinates are held in a map and are updated in increments using download times obtained from the other crawlers:
The crawler (requester) periodically sends a set of URLs (one for each web server) to a randomly chosen estimator. The estimator downloads the URLs and returns the set of download times as well as its coordinate and error to the requester. The requester then adjusts each web servers coordinate using the observed download time and estimators coordinate.
On each update, the median error of the set of coordinates is checked and, if it is below an acceptable threshold, the remaining URLs for each web server are assigned to the crawler whose coordinate is closest to that web server in NC space.

Coordinate Client

The coordinate client is responsible for maintaining the crawlers LANC. This is updated through gossip with randomly chosen neighbours: Each crawler periodically sends out a gossip message to a randomly chosen neighbour. Upon receipt of a gossip message, the neighbour replies with its coordinate and error. The requesting crawler uses the return trip time of the message and the neighbours coordinate and error to adjust its own coordinate.

3.2.2 Crawler System

Our crawling system architecture is similar to those discussed in Section . It is comprised of the following components:

- A Queue Manager to hold the crawlers pending workload.
- A DNS Resolver to translate hostnames into IP addresses.
- A pool of downloader threads to fetch content from the Internet.
- A URL Manager to organise the vast amounts of URLs seen during the crawl
- A blacklist to determine which pages should not be crawled
- A set of assigners to compute assignments based on various strategies.

Queue Manager

The queue manager holds the crawlers current workload. A URL is placed in the Queue as the result of one of the following events: (1) parsing of downloaded HTML (2) an assignment from another crawler in the system or (3) a proxy coordinate estimation request from another crawler in the system.
In [47, 49, 11] there is only one queue, but because our crawlers see URLs that have come from very different places we will need to consider the importance of each and and develop a strategy to crawl them accordingly:

- In our design, all URLs which are obtained from HTML parsing fall into the same category. This workload is common in all crawler designs. However, because we are not interested in individual performances of crawlers, we only process this workload in order to obtain a set of URLs which can be used for our assignments.
  The number of URLs obtained this way grows exponentially as the crawler runs: with an average of 350 out-links per web page the number reaches into the millions very quickly.
- Proxy request estimations are the result of some other crawler requesting us to download (estimate) a set of web servers. Although the other crawler will not be waiting on a response, it is important to send a response quickly to ensure the fast convergence of proxy coordinates.
  The size of the workload obtained this way depends on the systems configuration (specifically the number of domains δ in a batch), but in practice is quite small (in the hundreds). Because a set of proxy coordinates takes between 30 and 60 updates to converge, if a crawler holds 100 proxy coordinates, it will issue 60 requests (of 100 URLs each) to randomly chosen crawlers.
- URLs that have been assigned by other crawlers need to be crawled with some sense of urgency (and we should also try to crawl all of them) because this will allow us to test and quantify our assignment strategy. The size of this workload depends mainly on the size of the batches used in our assignment. In most of our crawls, we set the number of domains δ to 100 and the number of URLs μ per domain to 300 URLs, assuming we have all three assigners (geographic, random and PC) running each crawler would have $100 \times 300 \times 3 = 90000$ URLs to assign; with 10 crawlers running, these will be spread more or less evenly amongst the overlay, thus the number of URLs assigned to a single crawler rarely exceeds 50000.

All of these workloads are important and need to be crawled, but obviously cannot all be kept in one queue: if they were, the number of URLs from HTML parsing would eclipse those obtained from assignments which would eclipse those obtained from estimation requests.
To ensure fairness in the queue service, we organised the queue into three sub queues, one for each of the different events. We developed a selection policy based on probabilities to decide which queue a request should be served from.

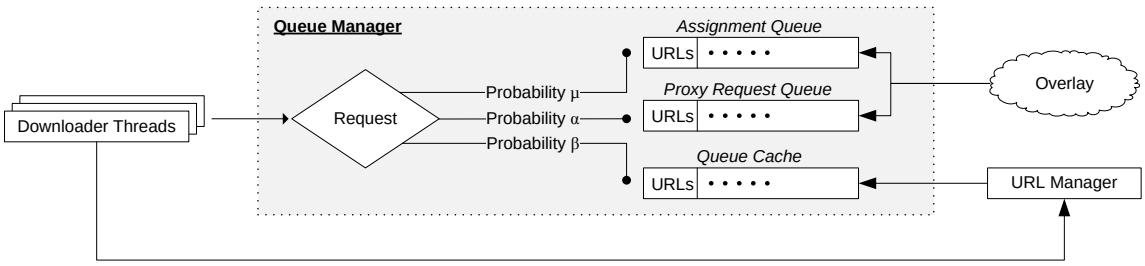


Figure 3.3: Shows the organisation of our QueueManager. It is composed of three sub queues, one for each possible URL origin. The downloader threads poll the queue through a single method. Which sub-queue the request is served from is determined through probabilities. In practice, we set the probability of a request being served from the assignment queue to a high value because we would like to crawl them as quickly as possible. The probabilities for the other two queues is usually lower due to the low numbers of URLs present in the proxy estimation request queue and the relative unimportance of the HTML parsing queue (Queue Cache).

DNS Resolver

The DNS resolver is used throughout the system to resolve host names into IP addresses. Originally we envisaged building a multi-threaded DNS resolver as using a single threaded DNS resolver is known to be slow[47]. However, after implementing a simple multi-threaded resolver, we noticed that it did not give us that much more of a performance increase over its single-threaded counterpart, probably because of the scale of our crawls which are not as large as those discussed in Section 3.2.2. Also, our overall strategy is to crawl servers which host large amounts of web pages under one IP. For this reason we decided to implement a single-threaded resolver with lazy caching.

Crawler Control

The crawler control manages the pool of downloader threads. Each thread works independently to download and parse HTML data. Once the data has been processed it is passed to the URL management system. Each thread holds an instantiation of the Downloader object and a reference to the URL Manager. We kept the amount of responsibility of a thread to an absolute minimum so that the failure of a thread will not affect the operation of the overall system. The number of threads that can be running at any given time is configurable.

Downloader

The Downloader is our HTTP client and HTML parser. The reason we choose to parse HTML within this object is so that downloaded content could be processed as soon as it is downloaded and would not be kept in memory any longer than absolutely needed.

URL Manager

The URL manager component is responsible for organising the URLs that are obtained through parsing as well as deciding which URLs should be assigned to other crawlers in the overlay.
There is one instance of the URL Manager shared amongst the downloader threads. Which URLs should be assigned are chosen using a simple ranking system: if the crawler has seen a number μ of URLs from the same domain, the URLs are added to a *DomainReady* data structure. Once the data structure contains δ domains. It is emptied and its contents assigned to the overlay using different assignment strategies (described in the next section).
Once the Assigners have successfully processed the batch, the URL Manager is notified and the process repeats.

Assignment Strategies

We decided to build two other assignment strategies in addition to our PC strategy: (1) A random strategy which assigns targets to crawlers randomly and (2) a geographic assigner which assigns targets to crawlers based on their geographic distance determined using the GeoIP database[16].
Each assigner receives the same batch from the URL Manager, and assigns it differently based on its policy: The random assigner iterates through all domains and assigns the corresponding URLs to a randomly chosen neighbour. The geographic assigner assigns all of a domains URLs to the crawler which is geographically closest to the hosting web server. The PC assigner first computes proxy coordinates for each web server and assigns all corresponding URLs to the crawler which is closest in the NC space. This assignment is made only when the median error of the set of Proxy Coordinates has fallen below a certain threshold.
The relationship between the assigners and the URL Manager follows the observer pattern, each assigner implements the *IAssigner* interface, the *URLManager* notifies them when it has collected enough domains. The *URLManager* in turn implements an *IAssignerWatcher* interface and is notified by the assigners when they have successfully assigned the batch.

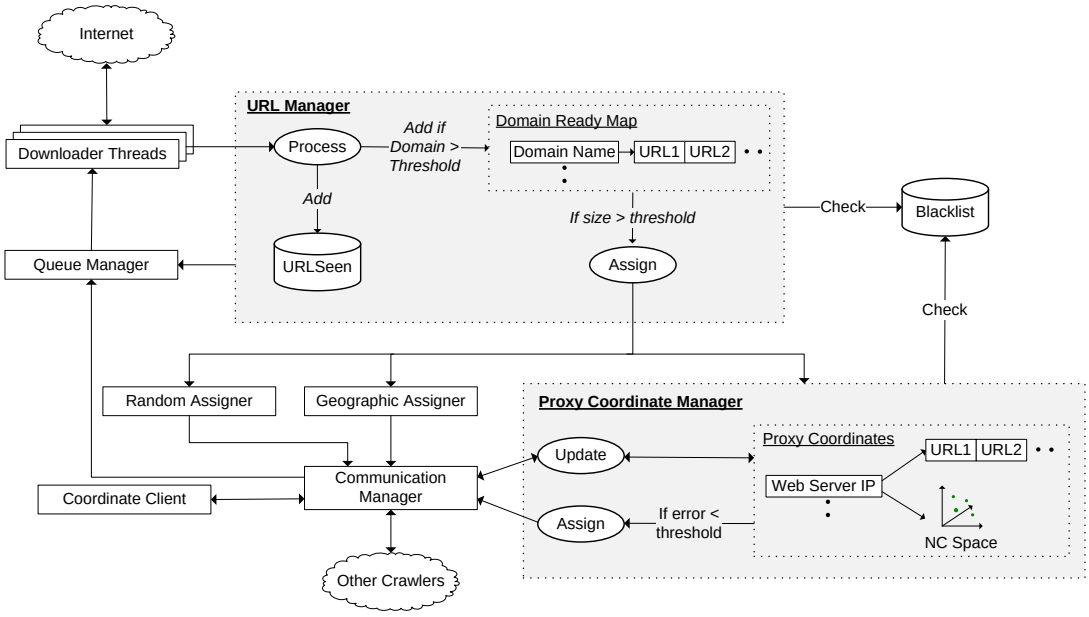


Figure 3.4: Gives a global overview of our system as well as the relationships between the different components.

Blacklist & Politeness

Because we were not planning to run very long crawls(such as those in IRLBot and UbiCrawler) and due to the limited time frame at hand, to handle politeness, we focused on maintaining domain blacklists and limiting the number of simultaneous download requests to a web server.
Domains are added to the blacklist in two ways: (1) The crawler parses the blacklist portion of the robots.txt file, if the crawler is blacklisted in robots.txt for a given domain, the domain is added to the blacklist. (2) As a further measure, we decided to implement an interactive blacklist: Each crawler periodically downloads a file from the DoC web servers containing a set of blacklisted domain names and IPs, web masters can add to this list through an online form whose address is referenced in the crawlers User Agent string.
The reason for the second blacklist is so that web masters can force the crawler to immediately cease crawling their domain. The blacklist is checked by both the Proxy Manager and URL Manager and a URL is dropped if its domain is blacklisted.

So that our multi-threaded downloader would not find itself in a position where each thread was downloading content from the same web server, we implemented a queue shuffling mechanism: The Queue Cache (HTML parsing queue) and assignment queue are maps with domain names as keys and buckets of URLs as values. When either one of these queues is served from, a random key is chosen and the first URL from the bucket returned.
This organisation for the queue was inspired from Mercator[47], however in their implementation each downloader thread is assigned a number of keys which it can choose from whereas in our implementation, the downloader threads can pick any keys.

3.3 Operation

Our system has one main mode of operation which we call cycle crawling, however for the purpose of testing & evaluation we implemented some other modes of operation. The crawler can be made to operate in a anyone of these modes by changing a configuration file. We will now describe these modes of operation.

3.3.1 Principal Mode of Operation: Cycle Crawling

The idea behind the way our crawler functions is as follows: The crawler should bootstrap and begin crawling. once it has a large enough set of URLs (a batch) it should assign them to other crawlers randomly, using geographic distances, and using proxy coordinates. The proxy coordinates will have to be computed (through estimation) before the crawler can assign the URLs this way, so the crawler should only assign the next set of URLs once the Proxy Coordinates have been computed and the URLs assigned. The crawler will repeat this process in "Cycles" until it is halted. The process is detailed below:

1. The crawler is seeded with a random set of URLs. These URLs are picked from a master URL file containing over 3 million URLs - this is so that the crawler will visit different parts of the Internet each time that it is run, limiting excessive visits to the same web site.
2. The seeds are added to the queue and the crawler begins to download and parse HTML content.
3. The URL Manager receives the URLs obtained through parsing and adds them to the queue.
4. The crawling continues until the URL manager has a batch of domains of size δ each with μ URLs.
5. At this point the URL Manager can optionally slow down the rate at which new pages are crawled so that the downloader threads do not interfere with the Proxy Coordinate computation.
6. The batch is assigned to the overlay using the assigners. Each assigner receives the same set of URLs, this is so that we can compare their relative performance.
7. The Random and Geographic assigners transmit the URLs to crawlers in the system based on their respective strategies.
8. The Proxy Coordinate manager will commence an estimation phase in order to determine coordinates for the web servers in the batch proxy. The Proxy Coordinate manager will assign the batch once the median error of the Proxy Coordinate set falls below a threshold θ. However if the Proxy Coordinates fail to fall below the threshold after π updates, they are assigned anyway, this is to prevent the crawler getting stuck on a set of web servers that will not converge and so to limit the number of requests made to a web server. This process can take up to 20 minutes, as such the URL Manager waits for the Proxy Coordinate Manager to finish before giving a new batch to the assigners.
9. When the Proxy Manager finishes, the URL Manager is notified.
10. If the crawl rate had been slowed, it is increased.

11. The URL Manager begins looking for a new batch (if it does not already have one) and the process repeats.

During this time, the LANC for each crawler is continuously being computed through gossip in a separate thread of execution.

3.3.2 Other Modes of Operation

In addition to cycle crawling, we implemented other modes of operation to cater for our evaluation requirements.

Persistent Proxy Coordinates In this mode the crawler is configured to find a batch and pass it to the Proxy Coordinate manager. The manager then computes the proxy coordinates for the batch. Once the median error falls below a threshold θ, the batch is assigned but the set of proxy coordinates is not dropped. The manager continues to compute the coordinates and assigns the batch again and again as soon as the outgoing assignment message queue falls below a certain size (we do this so that we do not overload the communications system or the receiving crawlers assignment queue). This mode was used to evaluate the long term behaviour of our Proxy Coordinates. Because of the large amount of requests that the crawler issues to web servers when operating in this way, we only use it for estimating coordinates for web servers which we control (running on PlanetLab).

Single Cycle Crawl As the name suggests single cycle crawling is when the crawler is configured to assign a single batch and then halt. This was used to fine tune the batch collecting process and find bugs.

Crawling with no Assignments In this mode we run the crawler but do not compute PCs or issue assignments. We simply download HTML pages, parse the content, add the out-links to the queue and repeat. We used this mode mainly to test the behaviour of the crawling component.

3.4 Discussion

Politeness When performing large crawls, we received multiple complaints about the behaviour of our system, including one formal complaint that was lodged against Imperial College. The Internet is the main source of income for many people and as such web masters do not take well to robots issuing hundreds of simultaneous requests to their servers. It is important to note that an environment such as PlanetLab which gives its users access to hundreds of high bandwidth servers can very easily be used for distributed denial of service attacks (even unintentionally). Behaviour such as this by software is very hard for web masters to counter because of the diversity of IPs on planet lab (the web master has to potentially block hundreds of IPs).

That said, it is very hard to achieve a 0 complaint rate when running any experiments which involves communicating with other peoples services because administrators are very paranoid and easily up-settable.

Originally our crawler referenced a GoogleMail email address in its UserAgent string, web masters were not too friendly in their communication to this address (because it did not really identify the crawler as being part of a University project). Following the advice of a web master, we referenced a web page hosted at DoC containing a description of the crawler (and later the blacklist request page) this was received well by web masters, the emails began to become more and more polite, because web masters now knew why their web sites were being crawled and also why bugs could occur.

The interactive blacklist was also well received by web masters, it allowed them to stop the crawler visiting their web site without having to physically ban IPs or change server configuration. Once a domain has been added to the blacklist, it is picked up by the crawler within 10 minutes. It was widely used by web masters

especially during longer crawls, and is something that we would re-implement if we were to build a large scale crawler again.

Configuring the Search The size of the assignment batch can be configured using two parameters: δ is the minimum number of distinct domains a batch must contain. And μ is the minimum number of URLs that must be under each domain. The crawlers search strategy can be adjusted by tuning the size of this batch. The parameter μ can roughly be seen as the depth to which we want to crawl each domain (although the URLs for any given domain could have originated from any page that the crawler has parsed). If we set a small μ, then each domain will be very lightly crawled, batches will be easier to find and the crawler will execute more cycles. If we set a large μ then the batches will be harder to find, but the time gained by assigning them will be considerable.

Proxy Coordinate Updates We would like to note that a single Proxy Coordinate is computed using download times obtained from multiple distinct URLs originating from the same web server. Because we are only downloading the HTML code of a web page and not the full set of images and other large media objects that many web sites now contain. The size of the web pages fall within a certain range, as such the variance in download times from a given web server is low enough for Pyxida to handle.

Using download times from multiple URLs gives our method the distinct advantage that it can be used in unison with an existing crawling system: All distributed crawlers will collect a large set of URLs for a smaller number of domain names[3]. In order to determine the best crawler to download them, they can simply issue a small number of downloads to adjacent crawlers, use the download times to compute the coordinates, and assign the whole batch of URLs to the best crawler. This means that a distributed crawler with the aim of downloading n URLs from a web server, will make n HTTP connections to the server with or without the Proxy Coordinate strategy. The only difference would be the increase in communication overhead required to perform the estimation.

Compatibility between Metrics Although it might seem strange to use RTT between gossip messages for our overlay coordinates and download times for our proxy coordinates, during our experimental work we found that the most of latency measurements for both metrics lay in the range between the hundreds and the thousands. The unit used for both RTT and download times is milliseconds. Although we did not find that both latencies followed the same distribution, this did not prevent us from computing low error proxy coordinates. And because the error in the Vivaldi algorithm is the squared difference between distance in the space and observed latency, a low error coordinate gives us an accurate estimate of the download time to a target.

Finally we would like to re-iterate that the distance between two crawlers in the NC space is the expected RTT for gossip messages (in milliseconds) (higher values of which tell us that the node is under high computational load), the distance between a crawler and a web server is the expected download time (in milliseconds). But the distance between two web servers in the space is meaningless and is purely a result of Vivaldi algorithm trying to find a rest state for the underlying spring model.

[3] because Internet link topology follows a Zipfian distribution (a large number of URLs point to a much smaller amount of domain names)[3]

Chapter 4

Implementation

In this chapter we will discuss the implementation of our system. We will begin with a brief look at some of the components which are used throughout our system such as data structures and ways of representing URLs. We will then look at the implementation of the Crawling component and finally that of the Network Coordinate component.

4.1 Common Objects

Before we begin looking at the components in more detail, we will briefly describe some of the common objects used throughout our implementation.

4.1.1 Data Structures

Because of the huge amount of data and multi-threading that a web crawler must handle, it was essential to use data structures that could handle concurrent read/writes and manage memory well. Java's *ConcurrentHashMap* and *ConcurrentLinkedQueue* objects provide implementations of a fast thread safe hashmap and queue, but they were not designed to handle very large datasets.
Ehcache[54] is a widely used Java distributed cache, it has multiple features including memory and disk stores, and can handle large data sets of key value pairs (where both keys and values can be any Java Serializable object). The maximum number of elements that Ehcache can store in memory and on disk can be configured as well as the eviction policy. We used it widely throughout our implementation to store the large amount of data seen throughout the crawl. We have implemented two wrappers around Ehcache Caches: bigMap and bigSet, which we will now describe.

bigMap

The bigMap object is a map data structure, it holds Strings as keys and buckets of Strings as values. As well as the operations that one would normally expect from such a data structure, it supports the *popRandomHeadValue()* action which selects a random key from the map, and returns and removes the first element from its bucket in FIFO order - this is used by our Queue.
To handle concurrency, the buckets are Java *ConcurrentLinkedQueue* objects with a maximum size (set at instantiation time); because Ehcache also handles concurrency, there was no need to synchronise the methods of this object.

bigSet

bigSet is a set of unique strings and supports the operations that one would expect (insert, contains, remove...). This object has no synchronised methods either as thread safety is handled by the underlying Cache.

4.1.2 CUrl Object

The CUrl object is a wrapper around the Java URL object and is the way in which URLs are represented throughout the system. It contains the origin of the URL (i.e. a Proxy estimation request, a URL that was parsed from previously downloaded HTML, or a target that has been assigned by another crawler in the system). If a host name has been resolved at some point during the operation of the system, its IP address will also be stored within this object.

4.1.3 Neighbour Manager

The Neighbour Manager is a simple object that maintains the crawlers knowledge of its peers, it has the following data structures:

- *upNeighbours* contains the set of neighbours that is known to be alive (responding to communication) at any given time. A neighbour is added to this set if the crawler has successfully communicated with it.

- *downNeigbors* contains the set of neighbours known to be down (not responding to communication) at a given time. A neighbour is added to this set if the crawler has observed a timeout or an error when communicating with it.

- *addr2coord* is a map containing neighbours as keys and their last seen coordinate and error as values.

4.2 Crawling Component

4.2.1 Downloader

Initially, we attempted to build our own HTTP client using the Java Net API. Although this worked well in the early stages of the project (when downloading from PlanetLab nodes which we controlled), when crawling the ”real” Internet we found ourselves constantly implementing additional features to deal with the diversity in the thousands of web servers encountered during a crawl. Implementing a reliable HTTP client which is conformant to RFCs is a huge task and because we rely on the client for our PC computation, we chose to use a third party client.
The Apache HTTP Client library is a widely used and up to date client. Our Downloader class is a wrapper around this library, it is responsible for all communication with web servers. For politeness issues, it identifies the crawler and provides a contact email address and URL in the UserAgent string.
A download is initiated by passing a CUrl object to the Downloader, the host name is resolved and the URL downloaded. Upon completion, the Downloader returns an instance of the IWebObject interface.

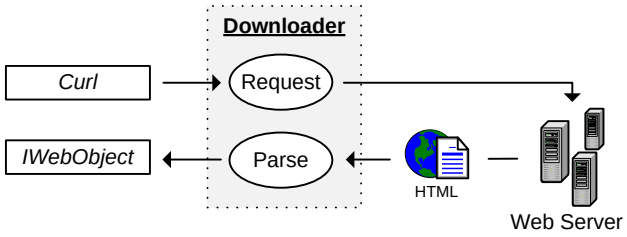


Figure 4.1: Shows the operation of the downloader.

IWebObject Interface

The IWebObject interface holds the useful information about the target. It can have three possible instances: (1) HTMLObject which represents a successfully downloaded target, (2) FailedObject which represents a target that failed to download, (3) UnknownObject which represents a successful download of content that is not supported (non HTML) - this could be images or videos for example.
The IWebObject interface holds the following information about the target:

- The URL of the target which is held in the form of a CUrl object.
- The size of the target in bytes, which is measured by counting the number of characters returned by the web server. Each character counts as a single Byte of data, which is the case for most ASCII encodings of plain text data.
- The download time, which is the time observed between the issue of a HTTP GET request and the receipt of the response.
- The status is the HTTP status code returned by the server, in the case where the downloader does not make contact with the web server, the status can be one of three integers: -1, which indicates a general communication failure, -2, which indicates an unresolvable domain name, and -3, which indicates an unsupported communication protocol.
- A set containing child links, which are obtained by parsing the HTML returned by the web server (this is only applicable for the HTMLObject instance).

The downloader is designed to be easily extendible, so in future the system could be extended to crawl ftp sites or images by creating new instances of the IWebObject interface.

HTML Parsing

In the early stages of developement, when we were crawling web servers which we controlled, we were relying on a simple HTML parser built using the Java regex libraries to extract anchor tags from HTML content. Unfortunately most HTML content present on the Internet does not conform to W3C standards, instead much of it is broken and malformed, but still contains valuable hyper-links which we want to extract. Building a robost HTML parser is a complex task, luckily there are people out there that spend a lot of time on them; we used HTMLCleaner, a light-weight HTML parser, it is elegant in it allows extraction of content from HTML based on XPaths. It also handles broken and malformed HTML well.
Once the links are extracted from the HTML, they are post-processed, using a set regular expressions. This is because anchor tags often contain relative links and malformed URIs (such as ”www.example.com” instead of ”http://www.example.com/”) which would fail future checks.
We encountered two further problems with HTML Parsing:

1. The HTML Parser was increasingly causing StackOverflow exceptions, this was because there were cases where it attempted to parse binary data. This was remedied by parsing the Content-Type field of the HTTP response and only parsing data if the content was plain text.
2. In much of the literature, the URL Manager is responsible for parsing HTML content, in practice we found that carrying around large strings in memory any longer than absolutely necessary was not a good idea and could lead to the Heap filling up. We decided to move the HTML parsing out of the URL Manager and into the Downloader: The HTML is parsed as soon as an HTMLObject is instantiated and immediately discarded thereafter, only the child links are retained in a list data structure within the HTMLObject class.

4.2.2 Crawler Control

The *CrawlerControl* is the point of entry for the crawling sub system, it instantiates the *QueueManager*, the *URLManager* and starts the pool of downloader threads. The threads are run using the Java *ExecutorService* which handles their asynchronous execution. Each running thread is also referenced in an array through which they can be issued start or stop commands.
Each thread runs on an infinite loop:

1. The thread polls the *QueueManager* for the next CUrl object to download (the *QueueManager* can also issue null targets, in which case the thread sleeps for 500 milliseconds and tries again - this feature can be used by the *URLManager* to slow down the crawl).

2. The thread passes the CUrl object to the *Downloader* which returns an instance of the *IWebObject* interface

3. The thread passes the *IWebObject* instance to the *URLManager* for further processing.

4.2.3 QueueManager

The QueueManager is responsible for managing the crawler's current work load. It is organised into three sub-queues:

- **Priority Queue** is a ConcurrentLinkedQueue of CUrl objects containing targets that have been assigned from the other crawlers in FIFO order.

- **Proxy Request Queue** is a ConcurrentLinkedQueue of CUrl objects containing proxy estimation requests in FIFO order.

- **Queue Cache** is a *bigMap*, this is where the URLs that have been obtained as a result of HTML parsing are kept (domain names are keys and buckets contain their associated URLs). The Queue Cache can hold a maximum of 15000 elements (5000 in memory and 10000 on disk), the maximum bucket size is set to 600 URLs.
  The cache eviction policy is LFU (least frequently used first), this is because we want to crawl domains which have a large amount of URLs: intuitively, an entry in the queue cache that is not often used (i.e. not often added to by the *URLManager*) would not have many URLs.
  Lastly, the Queue Cache is not accessed in FIFO order: when a request is served from this queue, a random key is selected from the map, the first URL for that domain is removed and returned (in FIFO order). This feature is to prevent the crawler from overloading a web server for which it has seen a large number of URLs in "one go". Without this randomization feature, our crawler could easily get stuck on pages with a large number of domains (such as spam farms or very large sites like Ebay and Amazon).

Probabilistic Queuing Downloader threads have no knowledge of the sub-queues, they poll the *QueueManager* through the *getNextTarget()* method. Which queue the request is actually served from is determined using a probabilistic approach:
The Priority Queue (as the name suggests) has priority over the other two queues because one of the main goals of the project is to investigate assignment strategies; it is essential that we crawl as many "assigned" URLs as possible. If it is not empty, there is a 0.8 probability that the *QueueManager* will serve a target from this queue.
The Proxy Request Queue probability of 0.1 because the total number of estimation requests issued by a crawler during an estimation phase is in the hundreds so the size of the Proxy Request Queue will generally be quite small.
If a request is not served from either of the above queues, it is served from the QueueCache, it is essential that we sometimes pick from this queue because we want to maintain a constant rate of new URLs coming in for the assignments.

There is also a probability that the *QueueManager* issues a wait command (or null target) to a thread, in which the thread will sleep for t milliseconds and try again. This feature can be used by the URLManager to slow down the crawl during the Proxy Coordinate computation phase - its default value is set to 0.

Logging Recording the evolution of the queue sizes was something that we were interested in doing as it gives a good indication of crawling activity. This is done by using a thread that dumps the sizes of the queues to file every 10 seconds. However, on demand measurement of the QueueCache size is very expensive (iterating through all keys and counting the size of each bucket) and a element that is stored on disk is much more costly to retrieve than one in memory (especially on PlanetLab), hence it is not done. Alternatively, we could maintain a counter on the put and get requests on the cache but we would also need to record evictions from the cache (which can happen any time) and count the number of elements in an evicted bucket. Although this is cheaper, it is still more expensive than not doing it at all and the size of the queue cache is not one of our experimental goals.

Thread Safety The QueueManager is the component in the system that sees the most concurrent accesses. each of the three sub queues has it's own put method. As all of the underlying data structures are synchronized[1], there was no need to use synchronized methods. The *getNextTarget()* method is not synchronized either for the same reason.

4.2.4 URL Manager

The *URLManager* is responsible for organising the large amounts of URLs seen during the crawl, deciding which ones should be assigned and when to assign them. It contains the following data structures:

- **URLSeen** is a *bigMap* that contains the full set of URLs that the crawler has encountered. The underlying cache can store a total of 15000 domain names (5,000 in memory and 10,000 on disk), after which it will evict entries on a LFU (least frequently used first) basis. The maximum number of URLs for each bucket is set to 600 which means that the maximum number of Urls that can be stored in URLSeen is $600 \times 15000 = 9000000$.
- **DomainReady** is a *bigMap* which contains domains (as keys) and buckets of URLs (as values) which are waiting to be assigned. The underlying Ehcache can hold 1000 elements in memory and 10000 on disk - although its size rarely exceeds the hundreds in practice.
- **Assigners** is a linked list containing instances of the IAssigner interface.

A single instance of the *URLManager* is shared amongst the downloader threads, it is accessed through a single synchronized method *processTarget()* which takes an instance of the IWebObject interface as an argument. When called, the *URLManager* does the following:

1. The origin of the URL is determined, if it is a result of downloading targets from parsed HTML it is not logged, if it is from an assignment from other crawlers it is logged to a file for future analysis, if it is from a Proxy Coordinate estimation request, it is passed to the communication manager.
2. If the target is valid HTML (an instance of the HTMLObject class) the child links (if any) are added to the URLSeen data structure.
3. Each child URL which is not currently contained within URLSeen is checked against the blacklist and added to the QueueCache data structure within the QueueManager.
4. The number of entries in the QueueCache bucket for each new URLs domain name is counted.
5. If the number of entries is over the assignment threshold, the domain is added to the DomainReady data structure and removed from the QueueCache (to prevent further crawling).

[1]The use of synchronized methods can be quite slow and should be avoided

The URL Manager contains one worker thread: the *checkReady* thread. It checks the size of the DomainReady data structure every 10 seconds, if it is over the domain assign threshold, the thread initiates the assignment process:

1. The DomainReady data structure is converted to a ConcurrentHashMap and emptied, this is so that it can be reused for the next cycle immediately.

2. The crawling can be slowed down (by setting the probability of issuing a wait command in the *QueueManager*), this is so that the Downloader threads do not interfere with the operation of the ProxyAssigner: we observed that running a large number of threads can increase the variance in the RTT samples used for the LANC computation leading to higher errors. However this is left as a configuration option as it might not be necessary if the number of crawling threads used is relatively low.

3. Each assigner is passed the Map containing the assignment.

Converting the DomainReady data structure can be quite costly and can take up to 1 second depending on how many URLs it contains. It is important note that the crawler threads will still be accessing the URLManager during this process. Handling thread safety was difficult (especially when the size of the data structures was large) and was acheived by using bigMap objects which handle concurrency and huge amounts of data well (thanks to the underlying Ehcache). Once this process is complete, the URLManager can carry on collecting domains for the next assignment cycle.

The URLManager waits until the Proxy Coordinate Manager has finished its assignment before issuing new assignments. This is because the Proxy Coordinate assigner takes time to compute the coordinates (whereas the Geographic and Random assigners issue assignments instantaneously) and we want the three assigners to issue roughly the same amount of URLs.

4.2.5 Assigners

In order to provide a quantitative evaluation of our assignment strategy relative to other published strategies, we have implemented both a geographic and a random assigner as well as our own proxy coordinate assigner. All three assigners implement the *IAssigner* interface, The relationship between the *URLManager* and the Assigners follows the observer design pattern: the assigners are added to a list within the *URLManager* at instantiation time, once the *URLManager* has enough URLs to assign, it passes the batch (as a key value set) to the assigners. The assigners iterate through the domains, picking a destination crawler based on their strategy and assign the chosen crawler the full set of URLs for that domain.

Geographic Assigner

The geographic assigner uses the GeoIP database [16] to make its assignments: each PlanetLab node holds a binary version of the database, the GeoIP Java client is then used to determine the geographic position of each crawler and unique domain name, the crawler that is closest to a given domain will be assigned the full set of URLs for that domain.

Random Assigner

For each domain in the batch a random neighbour from the *NeighbourManager* is picked using the Java *util.Random* library. All URLs for the domain are then assigned to that node.

Proxy Coordinate Assigner

The Proxy Coordinate assignment is handled by the *ProxyManager*, we describe its operation in Section 4.3.3.

4.2.6 DNS Resolver

Although implementing a multi-threaded DNS resolver would have been nice, it seemed too complex to get the timing right: at what point should a DNS resolution request be started? and who should be notified when the resolution is complete? Instead we implemented a simple lazy caching DNS resolver using the java.net API: The *DNSResolver* may be accessed by any component that sees CUrl objects. When a request is made, the resolver first checks its cache to see if it the domain name has been resolved before, if it has not, it is resolved and the IP is saved in the Cache for future requests.

CUrl objects also hold IP addresses, if at any point a domain name is resolved, its IP is stored in the object aswell. This limits calls to the Cache. Furthermore, when CUrls are transferred between crawlers (in assignments or estimation requests) the resolved IPs are also transferred for free.

4.2.7 Blacklist

There are two domain name blacklists used by the crawler, if a domain name is blacklisted, the crawler will not visit any URLs from that domain:

- The user driven blacklist is added to by web masters. We built a simple CGI web page running on the DoC web servers and referenced in the crawlers User Agent string. The web page contains a simple HTML form where web masters enter the hostnames or IPs that they would like to blacklist. Submitting the form updates a file on the web server, which is downloaded every 10 minutes by a thread and stored as a regular expression. Host names are checked against the regular expression when determining if they are blacklisted.

- Our crawler parses the blacklist fields of the robots.txt file. Any domain name that the crawler has downloaded a robots.txt for is stored in an Ehcache which is updated on a request basis: when the blacklist receives a request to check whether the crawler has been blacklisted through robots.txt, it first checks the cache to see if the file has already been downloaded, if it has not it downloads it and adds it to the cache. This is achieved using a small pool of Downloader threads and a queue. A domain name is assumed not to be blacklisted until proved otherwise (i.e. if the robots.txt file for the domain has not been downloaded yet or is non-existent then the domain is not blacklisted).

4.3 Network Coordinates Component

4.3.1 Communication Manager

Communication within the system is used for three things: (1) Network Coordinate Computation, (2) Proxy Coordinate Estimation Requests and (3) Issuing Crawling Assignments. Both proxy coordinate computation and overlay coordinate computation rely on randomly chosen neighbours for estimation requests, as such it does not pose a significant problem if a communication link fails, because the crawler can just chose another random neighbour. For the assignment communication, it is more important to reliably communicate a message, that said, when crawling thousands of web sites, it is not catastrophic if a few hundred URLs do not make it to their destination.

For these reasons, when designing the communications system, it was not seen as necessary to implement a high reliability system, such as JMS. Instead we extended the communication system present in Pyxida as it was relatively lightweight and was already being used for LANC computation.

The Communication Manager is responsible for maintaining the *upNeighbours* and *downNeighbours* data structures within the *NeighbourManager*: When communication timeouts or errors occur, the un-reachable crawler is added to the *downNeighbours* data structure. Similarly, when successful communication (either through attempted communication or the receipt of a message) is observed with another crawler, it is added to the *upNeighbours* data structure. The act of adding a node to *upNeighbours* causes it to be removed from

downNeighbours and vice versa.
When a crawler selects a random neighbour from the neighbour manager, there is a probability that it will choose a down neighbour (causing it to be added to *upNeighbours* if it responds). This method allows crawlers to communicate with other working crawlers whilst avoiding faulty ones.

Assignment Communication

The crawling assignments are passed to the *CommunicationManager* by the Assigners, the size of these assignments can vary depending on the crawlers configuration. Because they can potentially be quite large, an assignment to a given neighbour will be split into multiple sets that are no larger than 1000 Urls - this is to prevent the communication manager from chocking on very large messages containing tens of thousands of URLs. These sets are then pushed onto a queue, and removed every 5 seconds in FIFO order by a dispatcher thread.

4.3.2 Overlay Coordinate Computation

Crawlers use the return trip time of gossip messages to update their load aware network coordinates. The *CoordClient* object holds an instance of the Pyxida *NCClient* class which is responsible for the managing each crawlers coordinate. It contains a timer which every 10 seconds triggers the following update process:

1. The requesting crawler picks a random neighbour from the *NeighbourManager*.
2. It sends a *GossipRequest* message to the neighbour, records the sending time and waits for a response.
3. The arrival of a *GossipRequest* message triggers a callback on the receiving crawler: it replies with a *GossipResponse* message containing its coordinate and associated error to the requester.
4. The requester records the time of receipt of the message.
5. The requester then updates its own coordinate by passing the neighbours coordinate, the neighbours error, the observed round trip time of the message[2] to the *NCClient* class.

4.3.3 Proxy Coordinate Computation

The Proxy Manager is responsible for maintaining the crawler's set of proxy coordinates. It holds the *addr2coord* data structure which is a Java *ConcurrentHashMap* where keys are web server IP addresses, and values are *ProxyClient* objects.

ProxyClient Object

The *ProxyClient* object holds the coordinate and error for each web server as well as the set of pending URLs under its domain (as a FIFO queue). It is important to note that each *ProxyClient* is a wrapper around the Pyxida's NCClient library: in essence, each Proxy Coordinate exists in it's own Euclidean space, it is updated through latency measurements (which are download times) obtained from other crawlers in the overlay. Although each coordinate exists in its own space, two coordinates can be compared because each proxy coordinate is updated simultaneously with the same coordinates from estimators, so each space holds the full set of identical LANC coordinates for the overlay crawlers, but different ones for the Proxy Coordinates.

Creating Proxy Coordinates

The *ProxyManager* implements the *IAssigner* interface and is passed assignment Maps by the *URLManager*. When it is given an assignment, it populates the *addr2proxy* data structure by iterating over the map: Each host name is checked whether it is resolvable, if it is not, the domain and associated URLs are dropped. A *ProxyClient* object is then created for each web server and the URLs under its domain are added to the queue within its *ProxyClient* object.

[2] The RTT of the message is taken as the difference between the time of receipt and sending time

Updating Proxy Coordinates

Proxy Coordinate updates happen in three steps:

1. The crawler holding the coordinates (the Requestor) sends a **Estimation Request Message** (containing URLs) to a randomly chosen Estimator.
2. The Estimator sends an Acknowledgement message, stores the URLs in its queue and returns a response once they are crawled.
3. When the Requester receives the **Estimation Response Message** it updates its Proxy Coordinates with the observed download times.

Estimation Request Estimation request messages are issued every 10 seconds by an update thread in the *ProxyManager*. It loops over the Proxy Coordinate Map polling each queue for the next URL to estimate. These URLs are then added to a set, as they are added to the set, they are checked against the blacklist, any domain that is in the black list is removed from the Proxy Coordinate Map at this point[3]
The URL request batch is then sent to a randomly chosen estimator in the overlay. The estimator sends an acknowledgement and the process repeats.

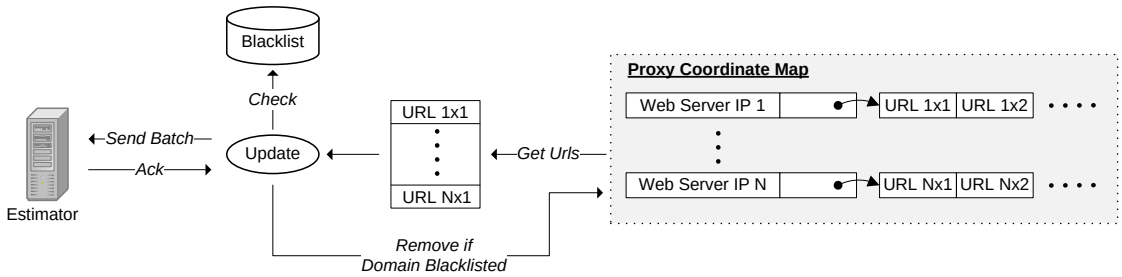


Figure 4.2: Depicts the sending of a Proxy Coordinate estimation request.

Receipt of Estimation Request When a Proxy Coordinate estimation request is received each CUrl object object is marked with the requester's address and a timestamp then added to the Proxy Request Queue within the *QueueManager*. This queue has priority over the main crawling queue (QueueCache), but lower priority over the assignment queue because the number of pending estimation requests are generally quite small relative to the number of assigned targets, especially during a large crawl.

The requests are removed from the queue and downloaded by the threads. After being downloaded, they are identified and passed back to the *CommunicationManager* where they are stored in a *ConcurrentHashMap*. This Map indexes the requesting crawlers address as keys, and the set of observed download times to the requested targets as values. There is a thread that checks the Map every 10 seconds, and dispatches any set of latencies that is older than 20 seconds. If a download arrives after its associated batch is sent, then a new batch is created and the process carries on.

In an earlier design, the URLs were not split up this way and a whole batch was handled by a single Downloader thread. This proved to be quite slow (because we were not taking advantage of our multi-threaded downloading) and if the thread failed, then the whole batch was lost.

[3]It is important to check the blacklist here because estimation of PCs requires continuously requesting pages from web servers from multiple vantage points. Because this can sometimes be demanding on the web server, web masters generally use the black list at this point.

The maximum age policy on waiting responses was implemented so that requests were guaranteed to receive a response within a given time. Without this feature, waiting batches might never be sent out if a download failed or if the communication with one of the web servers was taking an unreasonable amount of time.

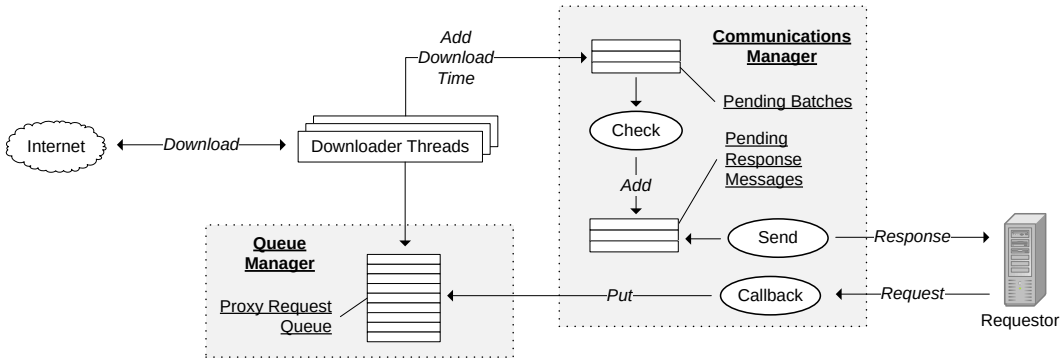


Figure 4.3: Depicts the receipt of a Proxy Coordinate estimation request.

Estimation Response The Estimation Response Message contains the set of observed download times as well as the estimators coordinate and coordinate error. The receipt of a response triggers a call back in the communications manager which passes the observed download times and estimator coordinate data to the *ProxyManager*. The *ProxyManager* then passes the coordinate, the error and each observed download time to the corresponding *ProxyClient* object within the *addr2coord* data structure. Each *ProxyClient* object then passes the latency and the estimator coordinate data to its Pyxida *NCClient* instance which updates the coordinate. Because estimation response callbacks can happen at any time, *addr2coord* had to be implemented as a *ConcurrentHashMap*.

At the end of this cycle, the median error of the set of Proxy Coordinates is checked against a threshold. If it falls below the threshold, the remaining URLs for each web server are assigned to the crawlers which are closest to it in the NC space, the *addr2coord* data structure is emptied and the *ProxyManager* notifies the *URLManager* that it has finished.

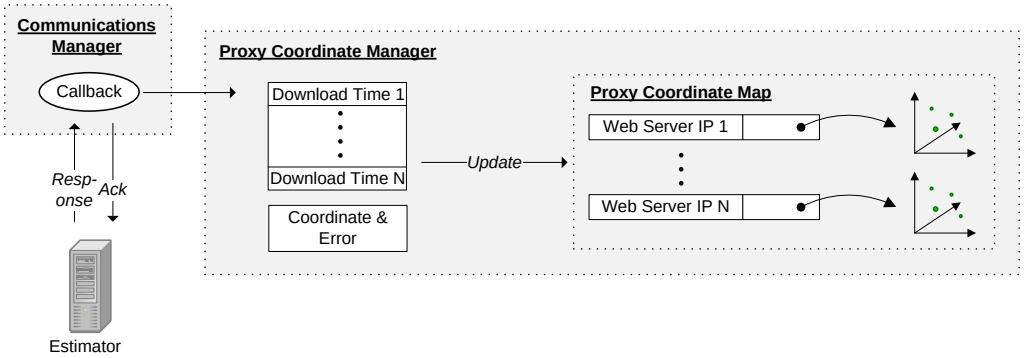


Figure 4.4: Depicts the receipt of a Proxy Coordinate estimation response by the requester

4.4 Testing and Distribution Methods

4.4.1 Software Testing

Throughout the course of the project, we implemented multiple tests to investigate the internal behaviour of our system. The most notable ones being used for testing the bigMap and bigSet implementations: We tested adding large numbers of URLs to the data structures, and checked the correctness when retrieving the URLs. We also implemented tests for the Downloader, to check timings and returned download statistics and also developed tests for the DNS server to check the correctness of the caching.

We used the different modes of operation of the software (Section 3.3) to test the crawling, URL management and assignment processing individually.

4.4.2 Web Servers

In order to test our crawler without disturbing web site owners, we installed web servers on 250 planet lab nodes: We used thttpd to host a Perl script running in CGI. The script would generate a random amount of HTML content containing a random number of hyper-links pointing to other PlanetLab web servers.
Because of the randomness in the content generated by this script, we can simulate a variety of different page sizes and download times.

4.4.3 Selecting PlanetLab nodes

PlanetLab nodes are always undergoing radical changes in load and do not observe 100% uptime, as such before we could deploy any software to the nodes we had to select the best ones. We developed a set of scripts which would concurrently issue the Linux *uptime* command (using muSSH[55] which can open a large number of SSH connections concurrently) to a large number of PlanetLab nodes. This would return their current computational load. We then select the machines with the lowest load from the set that responded to our command. This set was then partitionned into a number of independent sub sets.

By using multiple subsets we could run multiple identical experiments concurrently and thus be more resilient to failures in the PlanetLab nodes.

4.4.4 Distribution

To deploy and run our software on a large number of PlanetLab nodes simultaneously we developed a set of scripts to manage the distribution and execution of our binaries. We incorporated this functionality into our Ant build file so that we could move quickly from source code to a binary running on a target node. The build file contains a publish action which (when supplied with a list of target PlanetLab nodes) does the following actions:

1. The source code is compiled into a jar binary file.
2. A randomised list of seed URLs from a master seed file containing over 3 million unique domains is generated - this is so that each run of the crawler visits a different set of web sites, limiting regular visits to same web sites.
3. A file containing the list of the target PlanetLab nodes being considered is generated. This is so that each crawler knows which crawlers it should communicate with.
4. The compiled code and supporting files are compressed into an archive.
5. The archive is copied to a web server (generally the DoC web server) using RCP.
6. Each PlanetLab node is concurrently issued a command to download and extract the archive from the web server -this is done using muSSH.

After this process, the PlanetLab nodes have a ready to run executable. muSSH is used again to start or stop all binaries simultaneously. The process is quite fast and can publish the software to 70 nodes in around 10 minutes (assuming normal operation of the PlanetLab nodes).

4.4.5 Data Retrieval

Because the size of the log files generated by our software can be quite large (in the GigaBytes) we developed an efficient way to retrieve them. We wrote a script that compresses each nodes log files using muSSH. The archived logs are then transferred one by one using RCP.

4.4.6 Analysing Results

Once the archives have been downloaded, they are extracted and aggregated into a single set of logs for the whole set of Crawlers. The aggregated logs can then be processed by a set of scripts which use gnuplot to generate graphs.

4.5 Discussion

Thread Safety & Timing Thread safety and timing were two of the most difficult challenges to overcome when building our system. As you might have noticed, our system uses a wide array of timers and threads which run concurrently and change data structures continuously. Because our crawler runs independently and has no central point of control, it was crucial to implement a robust system. Synchronization in Java is very expensive and as such we tried to minimise the use of synchronized methods. This was achieved by using Java's *ConcurrentLinkedQueue* and *ConcurrentHashMap* Structures which are very robust and can handle large amounts of concurrency very well. Ehcache also proved invaluable to our implementation because of its ability to handle huge amounts of data whilst managing thread safety.

Downloaded Content Storage The aim of this investigation is to explore the ways of choosing optimal crawlers in assignment strategies and not to index downloaded content. Furthermore, disk writes on PlanetLab are very expensive and each node is limited to 5GB in total storage. For these reasons our crawler does not store the data it downloads, it discards it once it has parsed the HTML. This allows us to perform very long runs without worrying about the ways in which we will store the vast amount of HTML data seen throughout the crawl and thus makes our system less complex and more manageable.

Links In Section 4.2.3 we said that all of the hyper-links obtained as a result of HTML parsing were put onto the queue of the crawler which had obtained them. In the case of downloads from estimation requests, they are from web pages that the crawler in question is not responsible for (i.e. the crawler was asked by another crawler to download them) and those web pages might be pages that the crawler would not have otherwise seen. So we were faced with the question as to what to do with those links. Do we discard them, return them to the requester, or keep them? Clearly discarding them is not a good option because they might as well be kept by some crawler. We could pass them back to the requester, we know from [8, 14] that pages tend to reference other pages in the same geographic location so the requesting crawler could give them to which ever crawler ends up receiving the assignment, but in our assignment strategy geographic location is not as clear cut because a geographically close crawler might have very bad download speeds and so returning them to the requester might be just as good as the estimator keeping them.

There is a configuration option in our system, wherein estimators would return extracted links to the requesters, but we could not find a good enough reason for this to happen (Unless we only returned self-links). So in the interest of reducing communication overhead, we decided that estimators should keep parsed links for themselves.

Proxy Coordinates which Fail to Converge Sometimes a batch of PCs fail to converge within an acceptable amount of updates. This can be because of a high variance in the observed download times or unresponsive web servers. Obviously we need to set a maximum limit on the number of updates for a given set because it is not fair to issue a huge amount of requests to a web server just to estimate its coordinate. When this situation arises, web servers for which we have not been able to determine PCs can be assigned using the Geographic or Random assigners or just assigned at their current error levels.

Estimation Requests In a much earlier implementation, when a crawler received an estimation request it would initiate a set of new Dowloader threads to download the requested URLs. Despite this being less elegant than introducing them into the queue, we also noticed that starting a large amount of threads this way increased the variance in the round trip times of the gossip messages used in the LANC computation. This was because the thread that is responsible for updating the LANC suddenly found itself competing with new Downloader threads for CPU time. Similarly once the estimation request had been processed, the Downloader threads were terminated and the LANC thread suddenly had less competition.
By introducing the requests directly into the queue, we kept the number of threads alive at any given point in time consistent thus reducing the variance it the Gossip Message RTTs.

Chapter 5

Evaluation

In the evaluation of our system, we wanted to (1) highlight the ability of our assignment strategy to adapt to changes in the underlying crawlers, (2) compare our strategies performance (in terms of download time) to others, (3) show the overhead of maintaining Proxy Coordinates in terms of requests to web servers and computational load and (4) discuss our crawlers performance based on the criterea described in [3].

We will begin this chapter with an evaluation of the behaviour of the network coordinate system which underlies our assignment strategy, then we will compare the performance of our assignment strategy relative to other known strategies. Finally we will discuss the computational costs of our strategy and take a closer look at the behaviour of our system as a whole.

5.1 Network Coordinate Behaviour

In this section we will explore the behaviour of the network coordinates in response to (1) a decrease in bandwidth of a crawler, (2) an increase in computational load of a crawler and (3) an increase in response time of a web server. We will deploy our software on a small number of PlanetLab nodes and simulate the different phenomena. For the first two experiments we will benchmark our Proxy Coordinate assigner against a Geographic assigner so that we can highlight the adaptiveness of our system.

5.1.1 Effect of Decreases in Bandwidth on Assignments

In this experiment we explore how a decrease in a crawlers available bandwidth will affect its computed coordinate, and hence its assignment and the download speed of the system as a whole.
A decrease in bandwidth can occur if a crawlers network path suddenly becomes congested, or in the case of PlanetLab and other systems where bandwidth is shared among co-located virtual machines, when another process suddenly starts to consume large amounts of bandwidth.

For our experimental set-up we used 8 PlanetLab nodes to model the evolution of our system over a period of 140 minutes. The set of nodes used was composed of 6 European crawlers, and 2 North American crawlers. We used one of the European crawlers (which we will call the control) to manage a set of 17 proxy coordinates for PlanetLab web servers hosted in the US. The intuition behind choosing this geographic layout for the crawlers and web servers was so that we could more easily show the disparity between the geographic assigner and the proxy coordinate manager - because of the large distance between the web servers and the European crawlers, both assigners should favour the American crawlers.

After 33 minutes, we simulated a decrease in bandwidth on one of the American crawlers[1] by a factor of 5 and at 80 minutes we restored the bandwidth to its original value. The decrease in bandwidth was simulated by artificially increasing the observed download time within the *Downloader* component.

[1] We chose the American crawler which was being assigned the most targets by both assigners

The control crawler was configured with an assignment threshold of 0.3 but was configured not to release the proxy coordinates once an assignment had been made, instead to make assignments over and over again based on minimum NC distance; So as not to overload the assignment queue of the receiving crawler, we configured the control node as follows:

- The maximum number of URLs per assignment message was set to 500.
- A limit of 30 messages was set on the outgoing assignment message queue - the control would not assign new targets until this queue had less than 30 elements in it.
- The maximum number of URLs per domain (in an assignment batch) that could be assigned was set to 20.

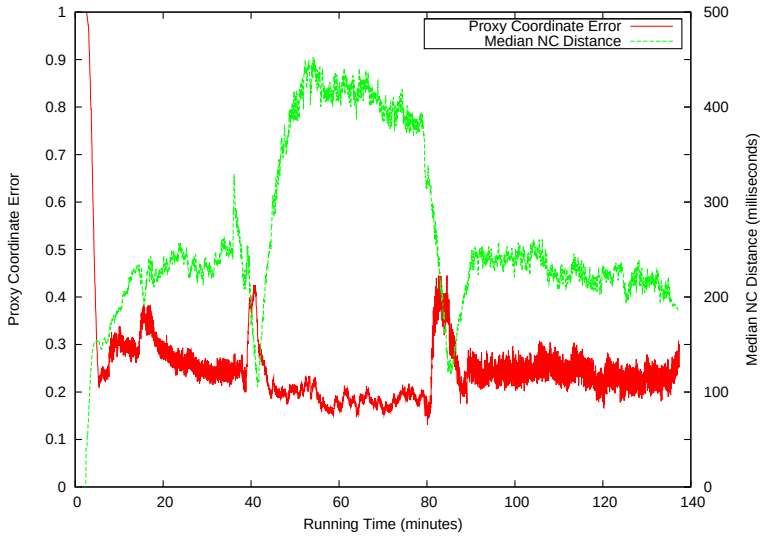


Figure 5.1: Shows the evolution of the median distance between the affected crawler and the proxy coordinates and the evolution of the median proxy coordinate error.

In Figure 5.1, we can see at 40 minutes, shortly after when we decrease the bandwidth, that the median distance between the crawler and the PCs increases sharply until settling between 400 and 450 milliseconds at 55 minutes; Concurrently, the median proxy coordinate error spikes to roughly 0.43 and then drops back to around 0.2 once the median distance has stabilised. A similar behaviour can be observed at around 80 minutes when the bandwidth of the affected crawler is returned to its original value.

During the period of low bandwidth, the affected crawler is downloading web pages at a much slower speed, thus its distance from the Proxy Coordinates increases. The Proxy Coordinate error spikes when the bandwidth is changed because there will a sharp variation in download times seen from the affected crawler and their position within the space will have to be adjusted. The reader should note that the affected crawlers coordinate will not change during the period of low bandwidth[2], instead the proxy coordinates will move further away from that crawler. We can correlate the NC behaviour seen in Figure 5.1 to the number of assignments made by both assigners to the affected crawler in Figure 5.2: Here we see that the number of URLs assigned by the Proxy Coordinate assigner stagnates at 40 minutes, 7 minutes after the bandwidth was decreased whilst that of the Geographic assigner remains the same. We can then observe a slight pick-up in assignments at 85 minutes after the bandwidth has been restored, and a much sharper pick-up at 115 minutes.

[2]Because a decrease in bandwidth will not affect the latency of the gossip messages

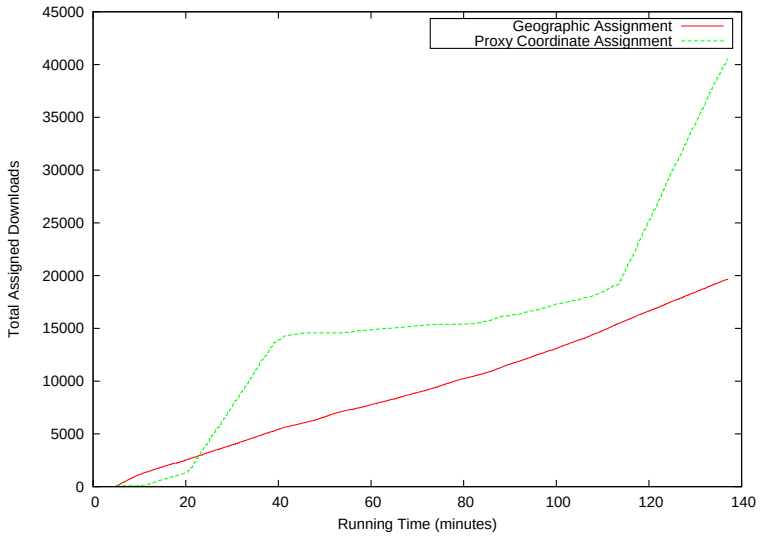


Figure 5.2: Shows the evolution of the total number of URLs assigned to the affected crawler by the Proxy Coordinate Assigner and the Geographic Assigner on the control crawler. This is plotted using data from the receiving end.

We are not too sure why the sharp pick-up in assignments only occurs at 115 minutes but could be due to slight variations in latencies and can be correlated with a slight change in median distance between the affected crawler and the PCs in Figure5.1, where we can see that the median distance between the affected crawler and the proxy coordinates drops slightly at minute 115 from 275 milliseconds to 250 milliseconds.

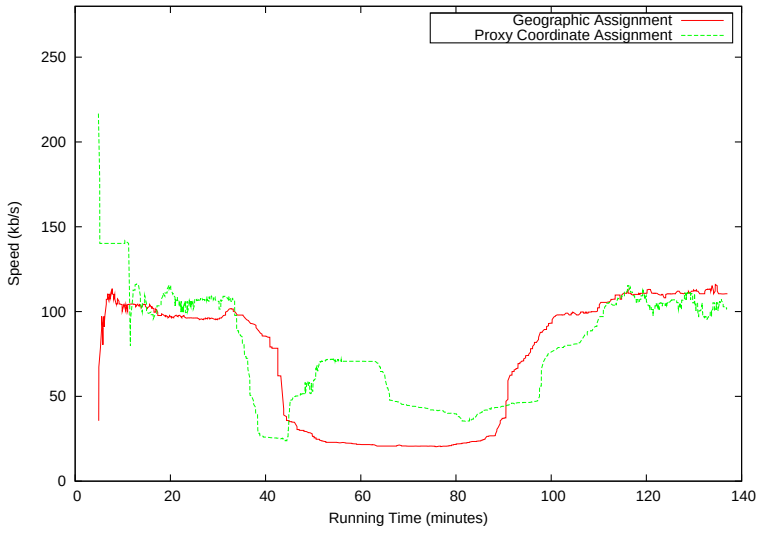


Figure 5.3: Shows the speeds of the Proxy Coordinate Assigner and the Geographic Assigner. For a given point in time, the speed is taken as the median speed of the 120 most recent downloaded URLs by the whole set of crawlers.

In Figure 5.3 we can see the effect of the decrease in bandwidth on the speed of the system as a whole: there is a significant decrease in speed for both assigners at roughly 35 minutes, the speed of the Proxy Coordinate assigner exceeds that of the Geographic Assigner at 45 minutes, this can be correlated with the stabilisation of the median proxy coordinate error at 45 minutes in Figure 5.1. This behaviour is mirrored at 87 minutes, after the crawlers bandwidth has been restored. The reason that we also see a drop in the proxy assigner speed during the low bandwidth period is because the Proxy Manager has re-assigned the targets which would have been assigned to the affected crawler, but the crawler which the targets where re-assigned to does not observe as fast download speeds as the affected crawler did before its period of low bandwidth. The Geographic assigner on the other hand has obviously not reassigned any targets and has taken the full hit of the bandwidth decrease.

5.1.2 Effect of Increases in Load on Assignments

Using a similar set-up to the last experiment, here we explored how an increase in load on a crawler will affect the rate at which it is assigned targets by the control node. We used 8 PlanetLab nodes to model the evolution of our system over a period of 100 minutes. We used the same set of nodes as we did the last experiment: 6 European crawlers and 2 North American crawlers. And we used one of the European crawlers to manage a set of 30 proxy coordinates for PlanetLab web servers hosted in the US.

We configured Pyxida with a *Window Size* of 20 and a *Smoothing Percentile* of 0.2 for both the Proxy Coordinates and the Overlay Coordinates, the reader is referred to Section 2.5.3 for an explanation of how these parameters affect the coordinate computation. The reason we chose to use a much more aggressive smoothing percentile is because increasing load on PlanetLab is very unpredictable and we needed to compensate for the high variance in the latency values and download times that will be seen during the period of high load.

After 36 minutes, we increased the load of one of the American crawlers[3] using a multi-threaded prime number finder[4], at 76 minutes we killed the process. The control crawler was configured similarly to the previous experiment to continuously make assignments at a rate which would not overload the receiving crawlers assignment queue. In Figure 5.1.2 we can see a sharp decrease in assignment rate at 38 minutes shortly after we started the load inducing process, the ProxyManager still assigns URLs to the affected crawler during the period of high load but its rate has considerably decreased. At 85 minutes (9 minutes after the load was restored) we can see a pick-up in the assignment rate. We can also see a slight increase in the Proxy Managers assignment rate at roughly 50 minutes, this is due to a spike in the affected nodes coordinate error at this time.

The reader should note that during the period of induced load, the affected crawlers coordinate will move further away from both the other crawlers coordinates (because high load causes higher RTTs for the Gossip Messages) and from the Proxy Coordinates (because higher loads causes decreases in download times). We can also see the effect of the induced load on the crawling speed of the system in Figure 5.5, where there is an overall decrease in the speed of both assigners during the period of high load, however the Proxy Coordinate assigner performs better overall.

The speed plot looks much more ”spiky” than that of Figure 5.3; a possible reason for this could be either that inducing high load on a PlanetLab node is highly unpredictable, and as such there will be a lot of variance in the download speeds seen, or that we are using more web servers in this experiment (30 instead of 16) and as such there is a higher variance in the download speeds.

Showing that an increase in load on a crawler would cause the Proxy Manager to decrease the rate at which it assigned its targets was much harder to show than in the bandwidth decrease experiment. We had to perform the experiment several times in order to get the parameters right (such as the smoothing

[3] the one which was being assigned the most targets by both assigners

[4] The prime-number finder is a Perl script that starts 150 threads each running in an infinite loop attempting to find prime numbers.

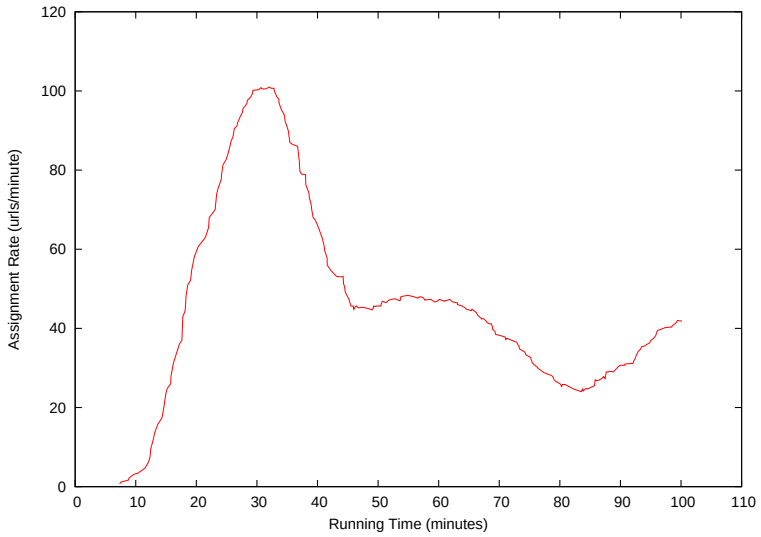


Figure 5.4: Shows the evolution of the rate of assignment from the Proxy Manager on the control crawler to the affected crawler. Although it is not shown here, the assignment rate of the Geographic assigner remains constant throughout the run of the experiment.

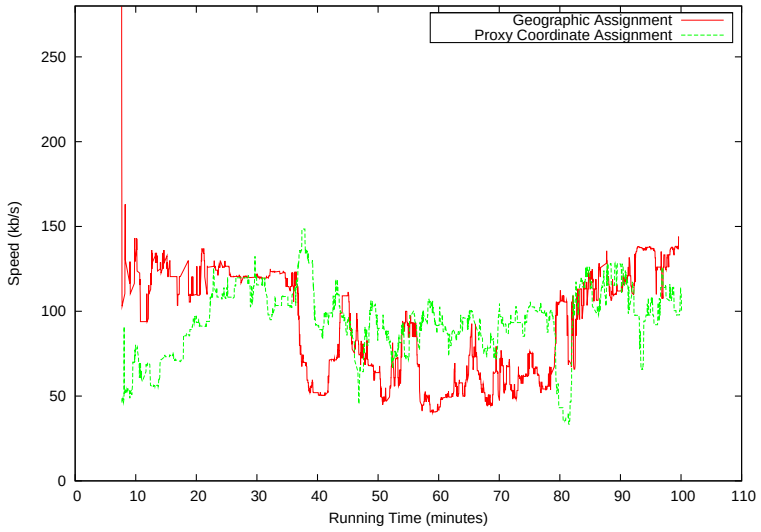


Figure 5.5: Shows the speed of the Proxy Coordinate Assigner and the Geographic Assigner. For a given point in time, the speed is taken as the median speed of the 120 most recent downloaded URLs by the whole set of crawlers.

percentile and the number of threads used in the prime number finder). We learnt that changing the load on PlanetLab in a ”predictable” way is very hard to do. In hindsight, maybe using a single-threaded script to influence load might have been more predictable.

Despite these complications, we believe that we have provided enough evidence that our NC system is adaptive to increases in load on one of the crawlers.

5.1.3 Effect of an Increase in Web Server Response Time

We will now take a look at changes in the Proxy Coordinates when the web server undergoes an increase in response time. To do this, we used a system of 7 PlanetLab nodes acting as estimators and one control node to record the evolution of the Proxy Coordinate over the course of the experiment, the control node did not assign URLs instead it just maintained a constant fix on the coordinate sending out an estimation request every 10 seconds. The Pyxida Proxy Coordinate *Smoothing Percentile* was set to 0.5 and the *Window Size* set to 16. we used a small Window so that the coordinate would adapt rapidly to changes and found that the value of the Smoothing Percentile did not really affect the outcome of the experiment.

We left the system for one hour to converge to a stable state, after which at 58 minutes we increased the response time on the server by 5 seconds using a CGI script - we used a Perl script that sleeps for 5 seconds when invoked. at 76 minutes, we reset the server response time to its original value.

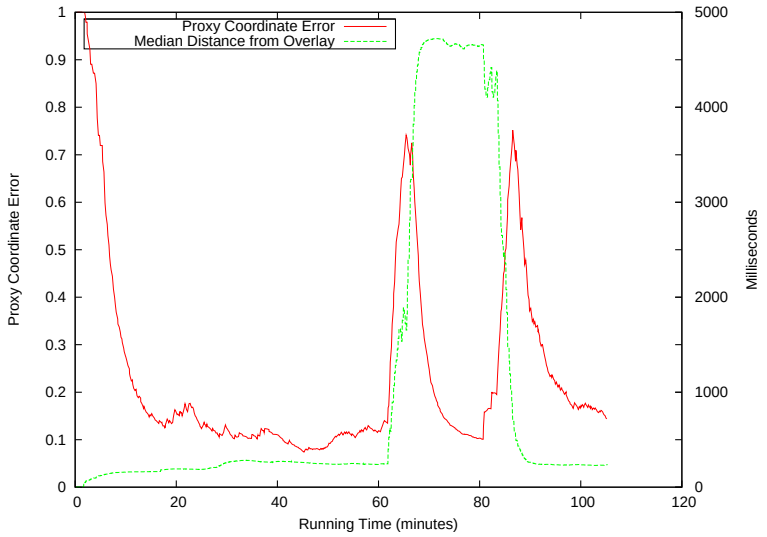


Figure 5.6: Shows the evolution of the median distance between the affected web server Proxy Coordinate and the Crawler Coordinates and the evolution of the Proxy Coordinate error.

The results of the experiment are shown in Figure 5.6, where we can see that at 62 minutes (4 minutes after we increased the response time) there is a dramatic increase in the distance between the web server and the crawlers, the error in the proxy coordinate also increases dramatically as the distance increases - the error increases because the Proxy Coordinate must adapt to the sudden increase in response time. Once the distance has stabilised, the coordinate converges to a low error state. We observe similar behaviour when we restore the server's response time to its original value.

This experiment shows how a web server will see its coordinate move further away in the NC space if its response time increases. Because all of the crawlers will see the same increase in response time, the web server's Proxy Coordinate moves further away from all crawlers. The aim of this experiment was to show how the web server response time affected its coordinate but this could be an interesting feature to integrate into a politeness policy: if the crawler system observed a web server moving away from all crawlers in the NC space, then it would know that the web server was probably overloaded and should stop crawling it.

5.2 Assignment Strategy Comparison

We will now evaluate the overall performance of our Proxy Coordinate assignment strategy relative to Geographic and Random strategies. The performance of the Random strategy in terms of download time is comparable to a hash based strategy because neither attempt to optimise network proximity.

We ran our crawler in single cycle mode on 7 independent sets of 10 planet lab nodes. Our aim was to allow the crawler to find a batch, compute Proxy Coordinates and assign the URLs using all three assigners; so that each assigner would assign approximately the same amount of URLs at the same time, we changed the URL manager so that the Proxy Coordinate assigner would trigger the geographic and Random assigners once it was ready to assign its batch. We used the single cycle mode to curb the memory usage of the crawlers so that the PlanetLab monitoring system was less likely to kill them. And we used a large amount of PlanetLab nodes (70 in total) so that our experiment would not be affected by failures of individual nodes. We ran this experiment twice, once with the Proxy Manager assignment error threshold set to 0.25, and once with it set to 0.15.

The download time CDFs for the various assigners are plotted in Figure 5.7. We can see that the both

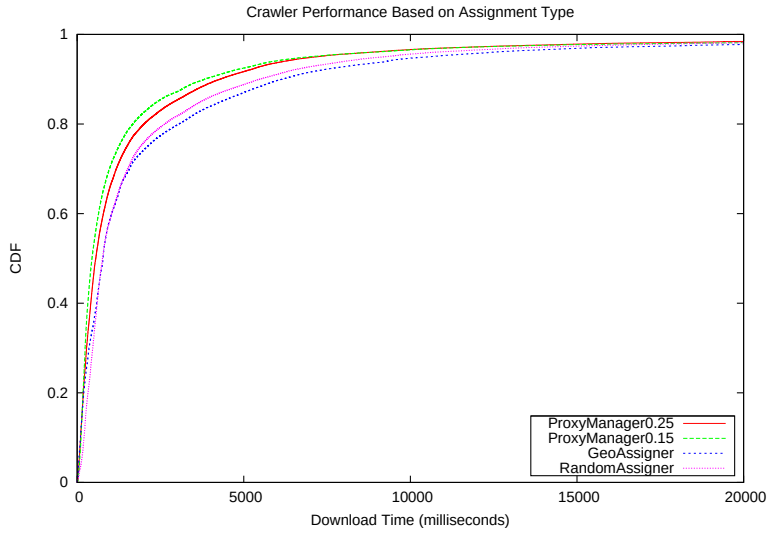


Figure 5.7: Shows the download time CDFs for the various assigners. ProxyManager0.25 (respectively ProxyManager0.15) is our proxy coordinate strategy with the assignment error threshold set at 25% (respectively 15%).

the ProxyManager0.25 and ProxyManager0.15 perform better than the Random and Geographic assigners. Interestingly, we found that the random assignment performs better that the geographic assignment, we believe that this is because of the un-predictability of PlanetLab nodes. The geographic assigner is constrained to choosing a crawler if it is geographically closest to a given target even if it is not the fastest, as PlanetLab nodes are very unpredictable, this can lead to very poor choices. The Random assigner should balance assignments more equitably and thus will assign an equal amount of targets to a bad node and a good node improving its overall download time.

We can see from Figure 5.8 that both of our proposed Proxy Coordinate Strategies perform better than the Random and Geographic assigner: for the ProxyManager0.25 there is a 90th percentile 22% decrease in download time over the RandomAssigner, and a 30% 90th percentile decrease in download time over the Geographic assigner. For the ProxyManager0.15 there is a 90th percentile 30% decrease in download time over the RandomAssigner, and a 38% 90th percentile decrease in download time over the Geographic assigner.

	90th Percentile	50th Percentile	Success	Failed	Total Download Size
ProxyManager0.15	3824.68	444.51	120661	4787	9617.67 Mb
ProxyManager0.25	4286.96	558.55	98722	5135	8827.54 Mb
RandomAssigner	5476.65	760.55	127683	7033	9633.58 Mb
GeoAssigner	6146.23	775.51	138563	9371	9998.86 Mb

Figure 5.8: Shows statistics for each assignment strategy.

5.3 Overhead of Maintaining Proxy Coordinates

In this section we will take a look at the cost of using Proxy Coordinates in terms of (1) number of requests issued to target web servers and (2) computational load on the crawler node.

5.3.1 Web Server Usage

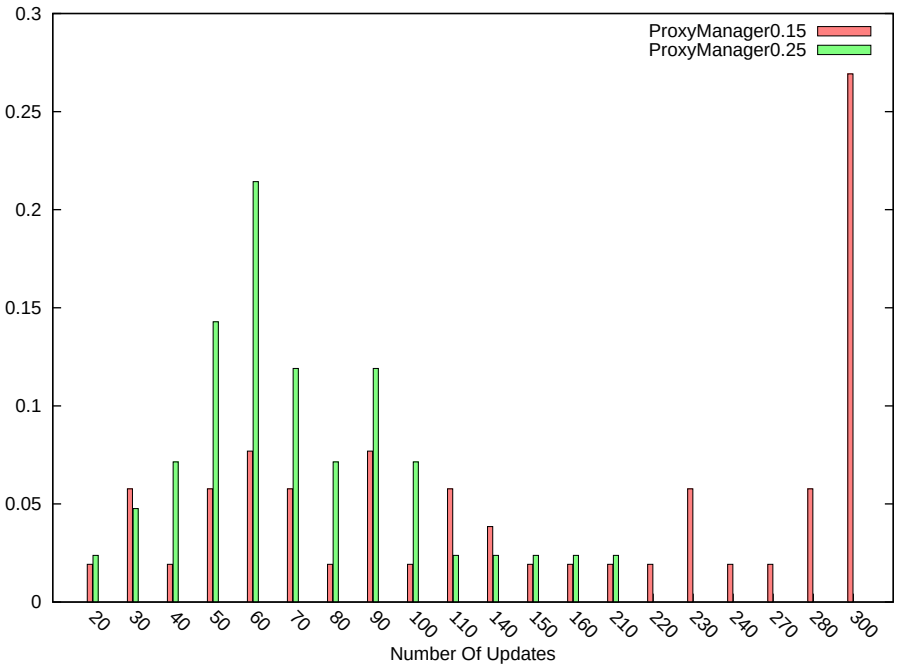


Figure 5.9: Shows the distribution of number of updates required to converge a set of proxy coordinates to an error level of 15% (ProxyManager0.15) and 25% (ProxyManager0.25), updates are normalised and rounded to the closest multiple of 10.

We plotted the distribution of the number of updates (where a single update is the result of an estimator downloading a URL from the web server) in Figure 5.9 that each crawler performed (from the experiment in Section 5.2) in order to converge its set of Proxy Coordinates. In the experiment, a threshold of 300 was set on the maximum number of updates that a crawler could make to a set of proxy coordinates. Thus in the case of ProxyManager0.15, 26% of the crawlers failed to converge the set of coordinates to an acceptable error within the given number of updates and those Proxy Coordinates were assigned at their current error level. The most probable explanation for this phenomena is a high variance in the observed download times from the web servers, this amount of variance may not hinder the proxy managers ability to converge a set of coordinates to an error of 0.25, but makes it harder for it to converge the coordinates to an error of 0.15.

The Proxy Manager was able to converge all of the Proxy Coordinates to an error of 25% or less, 90% of the coordinates required 108 updates or less and 50% of the coordinates required 64 updates or less. Converging the coordinates to an error of 15% was more difficult: 50% of them required 193 updates or less whereas 90% of them required 300 updates or less (remember that 300 is the upper bound on the number of updates and if a coordinate required that many updates, then it never reached the desired error level).

We can conclude that converging a coordinate to an error of 15% requires a lot more downloads from the target web server and as such can be less polite if the server is not able to handle that many requests, thus when using an assignment threshold of 15%, much more care must be taken as to what kind of web servers we are computing Proxy Coordinates for.

Finally, we would like to emphasise that the error of a Proxy Coordinate is taken as the squared difference between observed download time between a crawler and a web server and the distance between their respective coordinates in the NC space. Thus the error for a Proxy Coordinate is representative of our network coordinate system's ability to accurately predict the download time between a crawler and a web server.

5.3.2 Computational Cost

We will now take a look at the cost of maintaining Proxy Coordinates in terms of system load, we will look at a single cycle of the crawler so that we can show the difference in load on the PlanetLab nodes during the computation and after the assignment. In this experiment we used 7 crawlers running in single cycle mode to compute and assign a batch of 100 domains, we ran our experiment for one hour so that the crawlers would have time to compute the PCs, assign them and download them. The Proxy Manager assignment error threshold was set to 0.25.

Crawler	Time (seconds)	Median Proxy Error	Proxy Count	Update Messages
1	770	0.22	94	45
2	917	0.25	89	56
3	922	0.25	83	49
4	1054	0.24	88	66
5	1263	0.24	81	94
6	1266	0.23	88	61
7	1374	0.23	88	101

Figure 5.10: Shows (1) the Time taken to converge the coordinates to an error below the error threshold (0.25), (2) the median Proxy Coordinate error at assignment time, (3) the number of Proxies and (4) the number of Updates required. The average time for each node to converge its coordinates is 1081 seconds or 18 minutes and the mean number of updates required is 67 which falls within the median peek of ProxyManager0.25 in the update histogram of Figure 5.9.

Looking at the evolution of the load of the system in Figure 5.11, we can see that the median load is relatively high for the first 25 minutes of the experiment (between 1.5 and 4). After 25 minutes, the median load drops to below 1.5 and remains within this bound until the end of the experiment. This pattern can be correlated with the times taken for the crawlers to assign their Proxy Coordinates, In Table 5.10) the last crawler to assign its batch is Crawler 7 at 1374 seconds or 23 minutes, this coincides exactly with the point at which the median load drops to the low value.

We would like to note that the number of Proxy coordinates in each batch is less than the number that we requested (which was 100). This is because our URL parsing mechanism is not very robust and sometimes the system mistakenly identifies the domain name of a URL, erroneous domain names are picked up by the Proxy Manager when it attempts to resolve them and as such their URLs are not added to the set of Proxy Coordinates.

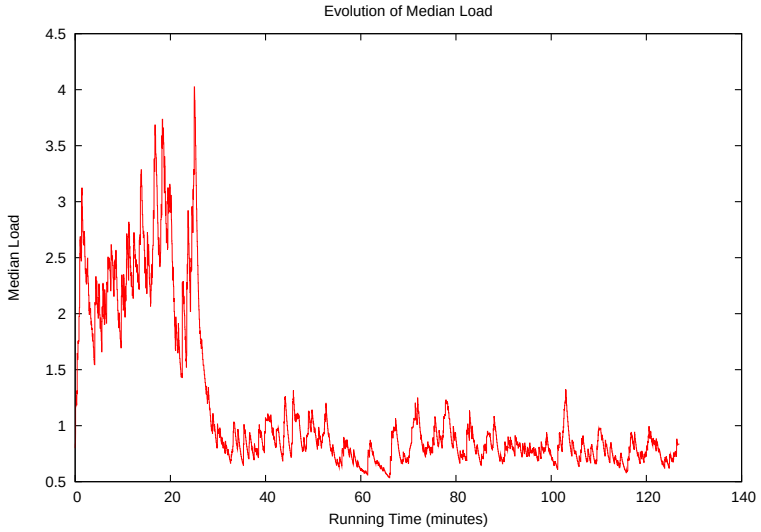


Figure 5.11: Shows the median load for the set of crawlers throughout the course of the experiment. The evolution of the load for an individual crawler is obtained by polling the linux uptime command for the 1 minute average load of that system.

Recall that in this experiment we only allowed the crawler to run for run cycle and then terminated, thus the above plot can be seen as the ”cost” of a cycle or the expected load increase of a system when computing a cycle. Each time Proxy Coordinates are being computed, the system will observe a similar load profile.

Load of a Linux System

The load of a Linux system is a measure the number of active processes waiting on the CPU at any time. High load averages usually mean that the system is being used heavily and the response time is correspondingly slow. It depends on the specification of system as to what is a high amount of load and what is a low amount of load; in our experience of PlanetLab, when the load of a node reaches 3 or 4, the node begins to become less responsive.

5.4 High Level View

In this section we will look at a run of our crawler from a high level perspective and attempt to evaluate our assignment strategy based on the metrics defined in Section 2.2.
The data in this experiment was obtained from a single 3 hour run of a system 14 crawlers, all of the results presented in this section came from the same experiment. The system was configured in the cycle mode of operation, each crawler was seeded with a set of 5000 randomly chosen URLs. In this run we only used the Proxy Coordinate assigner at an assignment threshold of 0.25. We did not use the Geographic Assigner or the Random Assigner so that we could have a more accurate representation of our assignment strategy. This is a more realistic experiment because in practice we would not run three different strategies concurrently.
The reader should note that we have ignored the downloads that each crawler must do in order to obtain a batch for assignment. This is because disk writes are expensive on PlanetLab and also we are not really interested in them because they are not being assigned.

5.4.1 Download Summary

First we will give a brief summary of what our crawler downloaded during the three hour period.

Total Runtime	184 minutess
Number of Crawlers	14
Unique Domain Names	836
Total Successful Web Server Connections	173588
Total Failed Connections	9868
Total Successful Downloads (HTTP 200 OK Connections)	164282
Total Download Size	13465.4 MB
Total Number of Hyperlinks Parsed	50279315
Average Links per Page	306

Figure 5.12: Shows some high level statistics of the crawl, these are taken as overall results from the system of 14 crawlers. The system successfully downloaded a total of 164282 web pages (only HTTP 200 status codes are counted here) as opposed to 173588 successful connections to web servers (which include other HTTP status codes as well as 200). Any situation where a crawler failed to make contact with a web server (i.e. through DNS resolution problems or general communication failures) is counted as a Failed connection. The crawler saw an average of 306 out-links per web page (this is obtained by dividing the total number of seen links by the number of successful downloads) we have noticed a trend in this number amongst all of our crawls: it generally falls in the range of 300 to 400 out-links per web page.

Html Status Code	**Count**
Communication Failure	9405
DNS Error	463
200 OK	164282
400 Bad Request	757
401 Unauthorized	59
403 Forbidden	594
404 Not Found	8353
410 Gone	10
500 Internal Server Error	1154
502 Bad Gateway	84
503 Service Unavailable	297
504 Gateway Timeout	35

Figure 5.13: Shows the break down of all of the web server connection attempts by the crawler. These are grouped by the HTTP status code which is returned by the target web server, status code 200 indicates a successful connection where HTML content was downloaded, status code 404 indicates the crawler requesting web pages which do not exist (possibly because of errors in our URL parsing), the reader is refered to [60] for a more detailed explanation of the other status codes. The entries Communication Failure and DNS Error are not HTTP status codes and represent situations where the crawler was unable to make contact with the target web server.

5.4.2 Assignment Strategy Evaluation

We will now briefly evaluate our assignment strategy using the criteria defined in [3].

Communication Overhead

Over the course of the Crawl, 1284 assignment messages were issued (with a maximum of 500 URLs each). 5360 successful Proxy Coordinate estimation requests were issued (estimation requests that received a response). The total number of distinct proxy estimation phases (or cycles) was 114. So on average, each Proxy Estimation Cycle required 47 updates to be issued.

Communication Overhead in [3] is defined as $E = \frac{l}{c}$, where c is the number of pages downloaded by the system as a whole and l is number of URLs that the crawlers have exchanged. In other assignment strategies such as the one in UbiCrawler[4] URLs are transferred individually, whereas in our implementation they are transferred in batches. Furthermore in our implementation there is an additional communication overhead associated with converging the Proxy Coordinates (estimation requests). Lastly, each crawler will require a certain amount of local crawling in order to find a batch that it will assign - the local crawling is ignored in this experiment. So it is not really representative to compare our communication overhead with that of others. But for the sake of completeness:
$E = \frac{l}{c} = 0.065$ We take $c = 183456$ as the total number attempted downloads, $l = 5360 \times 2 + 1284$ is the sum of the proxy estimation request messages, the responses (hence 5360×2 - because every request gets a response) and the number of assignment messages.

Coverage

The coverage of an assignment strategy is how many pages the crawling system has actually crawled relative to its target. It is defined as $C = \frac{c}{u}$ where c is the number of actually crawled pages and u the number of pages as a whole the crawler has to visit. A good assignment strategy should achieve a coverage of 1, e.g. every page that was assigned has been downloaded.
In this run, $u = 280684$ URLs were assigned in total and $c = 183456$ web server connections were made (failed attempts at communication are included here as well because the assigned URL was processed). Thus the coverage is $C = 0.65$, the reason that our crawler does not achieve a coverage of 1 is because the rate at which URLs are assigned is much greater than the rate at which URLs are downloaded (or removed from the queue by the downloader threads). In this experiment we used 100 downloader threads. A possible way of increasing the coverage would be to use more Downloader threads or decrease the size of the assignment batches (which was set to 100 domain names and 150 URLs per domain).
The reader should note that the coverage would decrease as the crawl progresses leading to a backlog in assigned URLs and the assignment queue becoming overloaded.

Balance

We are not aware of a common way to compute the balance of an assignment strategy, so we will make some brief observations about the number of downloaded URLs per crawler in the system. The standard deviation of the set is 7439.56, the minimum number of assigned URLs is 2125 and the maximum is 28374. As such, we can conclude that our assignment strategy is not very well balanced, however we would like to note that in a crawling system composed of crawlers with very different capabilities (such as our PlanetLab nodes), some crawlers will be much better suited than others to download URLs. As such some un-balance in the assignment is advantageous in an environment such as ours.

Overlap

Overlap is defined as $O = \frac{n-u}{u}$ where n is the total number of pages crawled, and u is the number of unique pages that the crawler as a whole was required to visit. $u < n$ happens if the same page has been erroneously fetched several times.
Because we ignored the amount of "local crawling" in this experiment and do not log full URL text because of disk space constraints, we are unable to give a true representation of the amount of overlap. The overlap in all assignments made from one crawler should be 0 because of the data structure where the assignment batch is held does not allow duplicate elements. But there is most likely a high amount of overlap in our system because when a crawler downloads a URL as result of an estimation request, it puts the extracted

hyper links back on its queue instead of returning them to the requester.
To see why this is the case, consider the following example: Crawler A is trying to compute a coordinate for the website Ebay.com and sends an estimation request to crawler B who has never encountered Ebay.com before. Crawler B downloads the web page and places the extracted links on its queue. Because Ebay.com contains a very large amount of unique web pages each with a significant amount of links back to Ebay.com, crawler B will very quickly see enough URLs from Ebay.com to include it in an assignment batch. Now both A and B will be assigning URLs from Ebay.com which will lead to a high amount of overlap.

Although we did not empirically evaluate the amount of overlap, we do expect it to be quite high in our implementation but could be improved by: (1) Returning Self-links obtained through the parsing of an estimation request to the requesting crawler (2) the monitoring of estimation requests (if a crawler receives a request from another crawler to estimate a URL under a domain, then it knows that the crawler is responsible for that domain) (3) Implementation of a common URLSeen data structure in the form of a distributed hash table.

5.5 Discussion

We have demonstrated our distributed crawlers ability to dynamically detect and compensate for a node that has seen a sudden decrease in bandwidth (In Section 5.1.1) and also for a node that has undergone a sudden increase in load (In Section 5.1.2). As such our assignment strategy is particularly well suited for a system of distributed crawlers comprised of heterogeneous hardware.

To our knowledge no other published strategy has demonstrated the ability to compensate for week points in the system in this way. It is worth mentioning though that UbiCrawler[4] provides a static way in which to compensate for nodes with different capacities (CPU, Memory or Bandwidth) in a hash based assignment strategy: by increasing the number of replicas for a given node (Section 2.2) in the system, their strategy will assign a larger portion of the Internet to that crawler. However the number of replicas for each crawler has to be configured before the crawl and as such is not adaptive to changes in the nodes behaviour, furthermore because their underlying strategy is hash based there is no way to optimise network proximity as such the overall performance of their strategy would be similar to our Random Assignment.

Although we are not aware of any assignment strategies which exhibits the same adaptive characteristics as our system, there are other strategies which use network latencies to determine their assignments. They can be grouped into two categories: (1) strategies which determine assignments off-line, before the crawl and (2) strategies which measure latencies ”on the fly” as new domain names are seen.

The strategies described by Rose et al[22] and Exposto et al[21] use latency metrics obtained off-line using the King[25] method and the Ping utility respectively. In both cases the measurements are then used to partition a set of target URLs among crawlers, these methods do not provide a way to deal with a previously unseen domain and as such are only suited to situations where there is a-priori knowledge of what will be crawled (such as target web servers and/or URLs).

In Section 2.2 we discussed two strategies which consider latency to a newly seen domain when determining their assignment: The crawler described by Loo et al[7] uses a technique called redirection: when a crawler receives an assignment, it uses ICMP Pings to determine its distance to the web server, if the latency is deemed to be too high, the target is redirected to another crawler. This is however used as a secondary measure to distribute work-load more efficiently in a hash based strategy and is not the primary mode of assignment in their system. Each crawler does not have absolute knowledge of other crawlers latencies (such as in our system), re-direction can also result in the target being assigned to a crawler which also observes high latency to the target in which case it is redirected again which is not efficient in terms of communication overhead.

Tongchim et al [13] use an estimation method based on ICMP Ping RTT in a system composed of two crawlers which is similar to ours in that when a crawler encounters a new domain, it measures its RTT to the target and requests for the other crawler to do the same, the crawler which observed the smallest RTT to the domain is then assigned all URLs under that domain. In their crawler once an optimal crawler has been found for a given domain, that crawler is kept as the designated crawler for that domain and as such their method is not adaptive - although we believe that their goal was not to perform long running crawls of the Internet. Furthermore, they only use this method in a system composed of two crawlers and as such have not demonstrated the scalability of their method.

In comparison with other assignment strategies in terms of download time, we found that using our Proxy Coordinate method at an assignment threshold of 0.15, we obtained a 38% 90th percentile decrease in download time over a Geographic Assignment and a 30% 90th percentile decrease in download time over a Random Assignment. The performance of the random assignment can be compared to that of a hash based assignment because in a hash based assignment there is no use of proximity or latency measurements. Although a hash based assignment does give advantages in terms of coverage, overlap, balance and communication overhead (see Section 2.2).

Although we did not empirically evaluate our strategy against an ICMP Ping based assignment strategy, we propose the following arguments as to why such a strategy is less suited than ours:

- ICMP Pings do give some indication of download speed, however they are not strongly correlated with throughput [23]. A possible reason for this is that the default size of a Ping packet is 64 bytes and is less likely to be affected by a router dropping packets than a web page whose size is in the hundreds of KiloBytes.
- Some firewalls block ICMP Pings and as such Pings may yield no response in cases where an HTTP connection would.
- ICMP Pings are sent through a low level connectionless channel and as such will not be affected by the the computational load of a crawler or a web server to the same extent HTTP connections are.
- From an engineering perspective, if we were to perform estimation using ICMP Pings to a target it would require the same amount of communication as estimation using HTTP downloads (substituting an ICMP ping for an HTTP connection). If we have enough URLs for the server being estimated, we might as well just download a URL and progress the crawl in addition to estimating the target. Furthermore, ICMP Pings generate more traffic between a crawler and a web server whereas ours generates the same amount that any other crawler would.
- Intuitively previously observed download times give a stronger indication of future download times than previously observed ICMP ping latencies.

On the subject of how our crawler performs on the criteria described in [3], communication overhead and balance seem to be less relevant when evaluating a system such as ours:
Absolute measurements of communication overhead should only be considered if it can be shown that reducing the communication overhead would result in faster crawling speeds - e.g the amount of bandwidth consumed by communication is hindering the downloading. To do this, we would not just look at the number of communication messages but at the sizes instead.

The importance of balance in the assignments is even more debatable because in a system comprised of heterogeneous hardware (such as PlanetLab), some crawlers might be able to handle a large work load where others might be less able, as such some un-balance in the job assignments for a system of heterogeneous crawlers can be advantageous.

The two criteria described in [3] that are applicable to us are overlap and coverage. Our crawler did not score highly on these due to implementation issues and ways in which they can be improved have been described in Section 5.4.2.

There are clear advantages in our assignment strategy over all other published strategies:

- We use download times directly to determine optimal crawler assignments. And we have shown that using our method results in a significant decreases in download times when compared to both geographic and random assignments.
- Our system has the ability to dynamically detect and compensate for week points in the system and as such can maximise the use of available resources.
- A Proxy Coordinate for a web server is computed by downloading the URLs under its domain. Although this requires additional communication between crawlers (for estimation requests and overlay coordinate computation), it does not require any additional communication with web servers (unlike Ping measurements).
- We represent both the crawlers and the web servers in a Euclidean space, where distances are representative of download times and load. This is very intuitive and gives all crawlers full knowledge of the overlay and Internet topology.
- The way in which we find URLs to assign (using batches of URLs under a domain whose size is over some threshold) works well with the Zipfian distribution of Internet link topology[3], which implies that a large number of URLs point to a much smaller amount of domain names. As such when a crawler wishes to download a large number of URLs from a domain, the cost of Proxy Coordinate estimation for that domain is negligible when compared to the amount of time saved in downloading such a large amount of URLs.

Chapter 6

Conclusions

Distributed crawling and assignment strategy research had previously focused on using the same underlying principals in different ways. To our knowledge, none of the research in this area had addressed the problem of optimising the use of heterogeneous resources. And more interestingly, none had considered using previously observed download times in assignment strategies. A probable reason for this is that the use of crawlers is mainly associated with large corporations which consider them trade secrets. However, due to the widespread use of the Internet and the increasing availability of large distributed systems to bed room developers. It is possible that more people and small organisations will find a need to download parts of the Internet. And because these people will not have access to massive data centres, they will have to make the best out of what they have.

We have used a radically different approach to that of previous systems by using Network Coordinates in our crawling design. And in doing so, not only have we shown how to build a dynamic and adaptable assignment strategy but have also shown that by considering previous download times one can greatly reduce future download times.

We have achieved this functionality simply and elegantly by using the dynamics of a network coordinate space. To our knowledge the use of network coordinates is not very widespread despite their ability to predict multiple types of network latencies. As such we believe that we have made a contribution by demonstrating an innovative use for them. Furthermore, we have done something quite innovative by showing that we can embed two very different kinds of latency measurements into the same network coordinate space.

We will now round off this report with a discussion of what we have learnt, some of the downfalls of our implementation and some of our ideas for further work.

Learning Outcomes

- **Building Complex Systems**: We learnt a huge deal about issues such as thread synchronization, managing huge data sets and communication. One of the most difficult things was getting all of the different threads working together for long periods of time. Even with the specialised objects that we used and which were designed to handle huge amounts of concurrency, there were still endless amounts of concurrency issues. We also learnt that even Java can get very serious memory leeks, and that a lot of care must be taken to destroy an object as soon as it is no longer needed.
- **Managing Complaints**: As we mentioned in Section 3.4, we received a lot of complaints throughout the course of the project. We learnt that when communicating with other peoples systems a great deal of care must be taken, especially when these systems are their primary source of revenue - as is frequently the case with Internet sites. Furthermore, we learnt that when performing experiments like ours it is very important to respond to all complaints and queries rapidly.

- **PlanetLab**: PlanetLab is a great tool, it gives its users a huge amount control over a lot of very powerful machines. However, PlanetLab nodes do have a tendency to be very unstable and unreliable, there are always failures occuring in the nodes, and we had to develop quite a few scripts to select good nodes and monitor execution. Furthermore, there are usage caps on disk storage, memory consumption and bandwidth, any process that exceeds a cap is terminated by the PlanetLab monitoring system.
  An important lesson that we took away from this experience is that if you need 10 PlanetLab nodes for an experiment then choose 50, split them into 5 sets of 10 and run the same experiment on all the sets. This greatly reduces the effect of failures and instability in the outcome of the experiments.

- **Distributed Systems**: This was the first time that we developed a large scale distributed system. We learnt that things that are very simple to do on a single machine, are very difficult to do cooperatively on a large number of machines. The most frustrating thing with long running, large scale systems is that they will generally work fine for the first couple of hours, but as soon as you start running them for a long time or attempt to scale them up, there is always something that goes wrong. Finding bugs can take a very long time and because in most cases, bugs only occur after a long series of events which are very hard to reproduce.

Downfalls

- **Communication**: The communication component of our system is largely based around that in Pyxida. Although it works well for small amounts of communication, it has been a bottleneck in our system, especially when we started to run much larger crawls towards the end of the project. The main choking point is when using it to transmit large amounts of URLs in assignment messages. In hindsight, we should have used a much more robust communication framework such as JMS and/or have used compression when transmitting large serialized objects.

- **Memory Consumption**: Memory consumption was a very difficult problem to overcome in the implementation of our system. We spent a lot of time fixing memory leeks and thinking up ways of economising memory. We got the crawler to a point where it could run for a few hours without it being a problem, but as the run time of the crawler progresses it has a tendency to use up a lot of memory and gets terminated by the PlanetLab monitoring System.

- **Politeness**: Although the user managed blacklist was quite successful and well received by web masters and we parsed the blacklist portion of the robots.txt. It was not enough for our crawler to be considered polite. During long crawls, we caused a large number of complaints by web masters, including some very aggressive ones. If we had to re-design the politeness policy of our crawler, we would parse more fields of the robots.txt file such as the exclusion rules and minimum interval between two requests (even though we tried limiting successive requests by shuffling our queue).

Further Work

- **Using Proxy Coordinates for Politeness**: In Section 5.1.3 we saw how an increase in web server response time moved its coordinate further away in the NC space. Because a polite crawler should stop crawling a server if it sees a drastic increase in its response time, a crawler using network coordinates could stop crawling a web server if it saw the Proxy Coordinate suddenly move away from all crawlers in the space.

- **Using the Vivaldi Height Vector to Achieve Balance in Assignments**: In an early design we envisaged embedding the size of each crawlers queue into the network coordinate space. This could be achieved using the Vivaldi height vector (Section 2.3.2) to model the size of the queue. A crawler with a very large number of URLs in its queue would see its height increase, moving it away from the other crawlers, and thus reducing its assignments. Once the crawler had worked its way through its queue, it would move closer to the other crawlers and its assignments would increase. This would

be very useful and could be used to achieve a much better balance, and because there would not be a case where there is a crawler that had a huge backlog of URLs, it would also improve coverage.

- **Message Routing**: In Section 2.3.4, we discussed $\hat{\theta}$-Routing and its applicability for routing messages in NC spaces. We could envisage using this technique in a much larger system of crawlers where nodes do not have total knowledge of all of their neighbours. Crawlers could compute Proxy Coordinates and then forward the URLs using $\hat{\theta}$-Routing to its nearest crawler neighbour. Techniques such as this might greatly reduce communication overhead in a much larger scale system.

- **Exploiting Geographic Distance Between Web Servers**: A question which we did not have time to answer in our investigation was: if a crawler observed high download speeds to a web server, would it also observe high download speeds to another web server that is very close geographically to it? The intuition behind this is that when two Internet hosts have a Geographic distance of 0, it is very likely that they are hosted by the same ISP and are on the same network. So maybe the crawler which is best suited to download from one is also best suited to download from the other. If this is the case then this could be used as a heuristic during the crawl and would reduce the amount of Proxy Coordinate computation that we would have to do.

- **Broadcasting Download Times to Compute Proxy Coordinates**: As we mentioned throughout this report, our crawler works in cycles to compute Proxy Coordinates and assign the URLs. Firstly, we think that it would be good idea to cache the coordinates instead of discarding them in the event that the web server is seen again.
  Secondly, we can envisage a much more complex implementation wherein each crawler perpetually computes Proxy Coordinates. For this to work, we would have to implement some kind of download time broadcasting system where crawlers transmit download times observed from web servers containing large numbers of URLs, there would then be a negotiation protocol between the crawlers to determine who would become responsible for computing the Proxy Coordinates. The system could then adjust assignments using the changes in the NC space and would more fully leverage the adaptability properties of the Proxy Coordinates that we have shown in this report.

Bibliography

[1] *Netcraft Ltd*. http://www.netcraft.com/

[2] *Google reveals scope of Web-crawling task*. http://news.cnet.com/8301-1023_3-9999814-93.html

[3] Junghoo Cho, Hector Garcia-Molina. *Parallel Crawlers*. WWW2002, May 7-11, 2002, Honolulu, Hawaii, USA.

[4] Paolo Boldi, Bruno Codenotti, Massimo Santini, Sebastiano Vigna. *UbiCrawler: A Scalable Fully Distributed Web Crawler*. Software: Practice and Experience. Volume 34 Issue 8, Pages 711 - 726. 24 Mar 2004

[5] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. *Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web*. In Proc. of the 29th Annual ACM Symposium on Theory of Computing, pages 654663, El Paso, Texas, 1997

[6] David Karger, Tom Leighton, Danny Lewin, and Alex Sherman. *Web caching with consistent hashing*. In Proc. of 8th International WorldWide Web Conference, Toronto, Canada, 1999

[7] Boon Thau Loo, Owen Cooper, Sailesh Krishnamurthy. *Distributed Web Crawling over DHTs*. ScholarlyCommons@Penn, 2004.

[8] Weizheng Gao, Hyun Chul Lee, Yingbo Miao. *Geographically Focused Collaborative Crawling*. Proceedings of the 15th international conference on World Wide Web, Search engine engineering, 287 - 296, 2006.

[9] *Unix Ping Software*. http://ftp.arl.mil/ mike/ping.html

[10] V.Jacobson. *Traceroute Utility*. ftp://ftp.ee.lbl.gov/traceroute.tar.gz, 1989.

[11] Vladislav Shkapenyuk Torsten Suel. *Design and Implementation of a High-Performance Distributed Web Crawler* Proceedings of the 18th International Conference on Data Engineering, 357, 2002.

[12] Kasom Koht-arsa and Surasak Sanguanpong. *High Performance Large Scale Web Spider Architechture*. in Proc. Internataional Symposium on Communications and Information Technology, Oct. 2002.

[13] Shisanu Tongchim, Prapass Srichaivattana, Canasai Kruengkrai, Virach Sornlertlamvanich, Hitoshi Isahara. *Collaborative Web Crawler over High-speed Research Network*. Proceedings of The First International Conference on Knowledge, Information and Creativity Support Systems, pp.302-308, Ayutthaya, Thailand, August 1-4, 2006.

[14] José Exposto, Joaquim Macedo, Antonió Pina, Albano Alves, José Rufino. *Geographic Partition for Distributed Web Crawling*. Workshop On Geographic Information Retrieval, Proceedings of the 2005 workshop on Geographic information retrieval, 55 - 60, 2005.

[15] RFC 1876. *A Means for Expressing Location Information in the Domain Name System*.

[16] Maxmind LLC. *GeoIP*. http://www.maxmind.com/

[17] *hostip.info*. http://hostip.info/

[18] *CAIDA. NetGeo - The Internet Geographic Database*. http://www.caida.org/

[19] Ying Zhang, Thomas C. Schmidt, Matthias Whlisch. *On the Correlation of Geographic and Network Proximity at Internet Edges and its Implications for Mobile Unicast and Multicast Routing*. The Second International Conference on Systems. ICONS 2007.

[20] Charu C. Aggarwal, Fatima Al-Garawi, Philip S. Yu. *Intelligent Crawling on the World Wide Web with Arbitrary Predicates*. In Proc. 10th Intl. World Wide Web Conference, 2001.

[21] José Exposto, Joaquim Macedo, Antonió Pina, Albano Alves, José Rufino. *Multi-objective Web partitioning for efficient distributed Web crawling*.

[22] Ian Rose, Rohan Murty, Peter Pietzuch, Jonathan Ledlie, Mema Roussopoulos, and Matt Welsh. *Cobra: Content-based Filtering and Aggregation of Blogs and RSS Feeds*. Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI'07), Cambridge, MA, April 2007.

[23] Nitin H. Vaidya, Saad Biaz. *Is the Round-trip Time Correlated with the Number of Packets in Flight?*. IMC03, October 2729, 2003, Miami Beach, Florida, USA. 2003 ACM.

[24] Schloegel, K., Karypis, G., Kumar, V. *A New Algorithm for Multi-objective Graph Partitioning*. European Conference on Parallel Processing. (1999) 322331

[25] Krishna P. Gummadi, Stefan Saroiu, Steven D. Gribble. *King: Estimating Latency Between Arbitrary Internet End Hosts*. SIGCOMM Internet Measurement Workshop 2002.

[26] Raj Lin *A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks* 1989 ACM Computer Communication Review.

[27] L. Subramanian, V. Padmanabhan, and R. Katz. *Geographic Properties of Internet Routing*. In Proceedings of the 2002 USENIX Annual Technical Conference, pages 243-259, Berkeley, CA, USA, June 2002. USENIX Association. http://www.usenix.org/events/usenix02/full_papers/subramanian/subramanian_html/node21.html

[28] T.S. Eugene Ng, Hui Zhang. *Predicting Internet Network Distance with Coordinates-Based Approaches*. INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE.

[29] Marcelo Pias, Jon Crowcroft, Steve Wilbur1, Tim Harris, Saleem Bhatti. *Lighthouses for Scalable Distributed Location*. Springer Berlin / Heidelberg, 278-291, 2007.

[30] T. S. Eugene Ng, Hui Zhang. *A Network Positioning System for the Internet*. USENIX Association, 2004.

[31] Manuel Costa, Miguel Castro, Antony Rowstron, Peter Key. *PIC: Practical Internet Coordinates for Distance Estimation*. Distributed Computing Systems, Proceedings. 24th International Conference on Distributed Systems , 178-187, 2004.

[32] J. A. Nelder and R. Mead. *A simplex method for function minimization*. Computer Journal, 7:308313, 1965.

[33] Frank Dabek, Russ Cox, Frans Kaashoek, Robert Morris. *Vivaldi: A Decentralized Network Coordinate System*. ACM SIGCOMM Computer Communication Review. Volume 34, Issue 4 (October 2004).

[34] P. Pietzuch, J. Ledlie, M. Mitzenmacher, M. Seltzer. *Network-Aware Overlays with Network Coordinates*. Lisboa, Portugal : Proceedings of the 1st Workshop on Dynamic Distributed Systems (IWDDS06), July, 2006.

[35] *PlanetLab*. http://planet-lab.org

[36] L. Corwin and R. Szczarba. *Calculus in Vector Spaces: Second Edition*. Marcel Dekker Inc., 1994.

[37] Nicholas Ball, Peter Pietzuch. *Distributed Content Delivery using Load-Aware Network Coordinates*. Proceedings of the 3rd International Workshop on Real Overlays & Distributed System (ROADS'08), Madrid, Spain, Dec 2008.

[38] Peter Pietzuch, Jonathan Ledlie, Michael Mitzenmacher, Margo Seltzer. *Network-Aware Overlays with Network Coordinates*. Proceedings of the 1st Workshop on Dynamic Distributed Systems (IWDDS'06), Lisboa, Portugal, July 2006.

[39] Jonathan Ledlie, Michael Mitzenmacher, Margo Seltzer, Peter Pietzuch. *Wired Geometric Routing*. 6th International Workshop on Peer-to-Peer Systems (IPTPS'07), Bellevue, WA, February 2007.

[40] Y. Hassin and D. Peleg. *Sparse Communication Networks and Efficient Routing in the Plane*. In PODC, July 2000.

[41] J.M. Keil and C.A. Gutwin. *Classes of graphs which approximate the complete euclidean graph*. Discrete Computational Geometry, 7:1328, 1992.

[42] Peter Pietzuch, Jonathan Ledlie, Margo Seltzer. *Stable and Accurate Network Coordinates*. Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS'06), Lisboa, Portugal. 2006.

[43] Han Zheng, Eng Keong Lua, Marcelo Pias, Timothy G. Griffin. *Internet Routing Policies and Round-Trip-Times*. Springer Berlin / Heidelberg, Passive and Active Network Measurement, 2005.

[44] Peter Pietzuch, Jonathan Ledlie, Margo Seltzer. *Proxy Network Coordinates*. Technical Report 2008/4, Department of Computing, Imperial College London, United Kingdom, February 2008.

[45] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[46] Marc Najork and Janet L. Wiener. *Breadth-first search crawling yields high-quality pages*. In Proc. of 10th International World Wide Web Conference, Hong Kong, China, 2001.

[47] Allan Heydon, Marc Najork. *Mercator: A Scalable, Extensible Web Crawler*. WWW, Springer Netherlands, 219-229, 2004.

[48] Sergey Brin and Lawrence Page. *The anatomy of a large-scale hypertextual Web search engine*. In Proceedings of the Seventh International World Wide Web Conference, pages 107–117, April 1998.

[49] Hsin-Tsang Lee, Derek Leonard, Xiaoming Wang, Dmitri Loguinov *IRLbot: scaling to 6 billion pages and beyond*. Proceeding of the 17th international conference on World Wide Web, 427-436, 2008.

[50] Koster, M. (1996). A standard for robot exclusion. http://www.robotstxt.org/wc/exclusion.html

[51] *Berkeley Internet Name Domain*. https://www.isc.org/software/bind

[52] *Advanced, easy to use, asynchronous-capable DNS client library and utilities*. http://www.chiark.greenend.org.uk/ ian/adns/

[53] *Mozilla Open Directory Project*. http://www.dmoz.org/

[54] *Ehcache*. http://ehcache.sourceforge.net/

[55] *muSSH*. http://sourceforge.net/projects/mussh/

[56] *The Jakarta Commons HttpClient*. http://hc.apache.org/httpclient-3.x/

[57] *Pyxida: An Open Source Network Coordinate Library and Application*. http://pyxida.sourceforge.net/

[58] *Apache Ant*. http://ant.apache.org/

[59] *HTML Cleaner*. http://htmlcleaner.sourceforge.net/

[60] *RFC 2616*. Fielding, et al. http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html