A Distributed Web Cache using Load-Aware Network Coordinates

Nicholas Ball
Department of Computing
Imperial College, London

Supervisor: Peter Pietzuch
Second Marker: Alexander L. Wolf

June 17, 2008
Abstract

Worldwide internet usage has increased by 290% over the last eight years. This increase has put considerable amounts of strain on web servers which are showing symptoms such as; high latencies, dropping/refusing/resetting connections and returning errors to clients. A decent distributed web cache could put an end to these problems. Web caches reduce the load on web servers by duplicating and distributing frequently accessed content, reducing the number of requests the origin web server handles. Many existing web caches offer a variety of benefits, however none seem to strike the right balance between locality and load awareness, two very important properties.

This investigation aims to develop a scalable distributed web cache that does strike this balance. This will be achieved with a promising new mechanism, network coordinates. Network coordinates places each participating machine into a virtual coordinate space, where distances can be interpreted as latencies. The mechanism therefore inherently offers large amounts of locality based information in a very simple format with low-costs. However, one downside of network coordinates is that they do not incorporate any load awareness, an essential property for a successful distributed web cache.

This investigation will also show how network coordinates can be extended to be load-aware. The proposed distributed web cache will use these transformed network coordinates, as well as other enhancements, to give a more sophisticated locality and load-aware system. Such a system will be able to handle both today’s and tomorrow’s internet demands.
I would like to thank my supervisor, Peter Pietzuch, for all the support he has given me throughout the year. Not only was his advice invaluable, but his enthusiasm and genuine interest in this project was admirable and motivating.

I would also like to thank my second marker, Alexander L. Wolf, for providing me with some key pointers on the direction of the project and its potential problems.

Finally, I would like to thank my girlfriend, for putting up with me working late into the night over the last month or so. I hope she can forgive me for not seeing her as much as I would have liked.
# Contents

1 Introduction ........................................................................... 1  
  1.1 Contributions ................................................................. 4  

2 Background ........................................................................... 5  
  2.1 Distributed Web Caches ...................................................... 6  
    2.1.1 Squid ................................................................. 6  
    2.1.2 CoralCDN ............................................................ 7  
    2.1.3 OASIS ................................................................. 8  
    2.1.4 Meridian .............................................................. 10  
  2.2 Network Coordinates ....................................................... 10  
    2.2.1 Network Overlays .................................................... 10  
    2.2.2 Network Coordinates ............................................. 12  
    2.2.3 Vivaldi ............................................................... 13  
    2.2.4 Accuracy and Stability ........................................... 15  
    2.2.5 Routing Schemes .................................................. 16  
  2.3 Web Caching ................................................................. 18  
    2.3.1 How a Cache Works ............................................... 18  
    2.3.2 Replacement Strategies ......................................... 19  
  2.4 PlanetLab ................................................................. 19  
  2.5 Languages and Supporting Applications ......................... 20  
    2.5.1 Java ................................................................. 20  
    2.5.2 Python ............................................................. 20  
    2.5.3 SMV ................................................................. 21  
  2.6 Libraries ................................................................. 21  
    2.6.1 Pyxida Library .................................................... 21  
    2.6.2 HttpComponents Library .................................... 22  
    2.6.3 JUnit ............................................................... 22  
    2.6.4 Prefuse Library .................................................. 22  

3 Overall Architecture ........................................................... 23  
  3.1 Cache ............................................................................ 23  
  3.2 Client ............................................................................ 24  
  3.3 Interaction .................................................................... 25  

4 Design .................................................................................. 27  
  4.1 Multiple Designs ............................................................ 27  
    4.1.1 Oracle System ..................................................... 27  
    4.1.2 Distributed System .............................................. 27  
  4.2 High-Level Managers ..................................................... 28
8.3 Challenges .............................................. 79

9 Conclusion & Further Work ............................... 81
  9.1 Further Work ........................................ 82
  9.1.1 Refinements ...................................... 82
  9.1.2 New Directions ................................... 82
  9.2 SMV Model for the Channel Locking Mechanism .......... 83
  9.3 CoTop Recordings for Disk-Memory Experiment ........... 84
    9.3.1 Cache Node - Disk Test ........................ 84
    9.3.2 Cache Node - Memory Test ...................... 85
    9.3.3 Client Node - Both Tests ....................... 85
  9.4 Linux Distribution Mirror XML File .................. 85
  9.5 Visualization Module Screenshots ..................... 86
Chapter 1

Introduction

Internet congestion has evolved to be a growing concern amongst all users [3]. Figure 1.1 shows that worldwide internet usage has increased by 290% over the last eight years. Moreover, internet users and contributors (web servers) are being placed in more geographically dispersed locations such as the Middle East and Africa. The former with a 1176.8% usage increase and latter 1030.2%. Due to this increase, more and more web servers are becoming overloaded and as a result are generating high latencies, dropping/refusing/resetting connections and returning errors to clients.

![Figure 1.1: Worldwide internet usage growth between the years 2000 and 2008 [1].](image)

Essentially, there are two main problems at hand. Firstly, the increase of internet usage has brought long-term overloading of web server, as the number of daily requests cannot be handled. Short-term high-load can also affect web servers, such as when serving sudden 'hot' or highly popular content. This effect of a sudden surge of incoming requests has been dubbed the 'Slashdot' effect [2]. Secondly, the increased dispersion of users and content adds to the congestion as content must travel further distances effecting large parts of the network along its route back to the requester.

Web caching is essential to the growth and scalability of the World-Wide Web as it mitigates overloading and congestion by duplicating and distributing frequently used content across a set
of caches. Two very important properties for the success of a distributed web cache today are; 

*locality awareness* and *load awareness*.

Systems are *locality-aware* if they keep data sent, local. This will involve taking into account where the request is coming from and returning the content from a cache that is nearby. Proximity is usually in terms of latency but may be for example geographically based. Nevertheless, all these schemes try to keep data local, mitigating the effects of transferring data long distances and obeying the locality principle [20]. Locality-awareness offers a number of benefits, such as:

- Offering lower latencies.
- Decreasing network saturation as data is kept local.
- Much better reliability due to fewer links and routers being traversed.
- Offering higher bandwidths.

For a system to be *load-aware*, there needs to be some selection which reduces the overloading of nodes. This may be the done by distributing requests uniformly across all caches or with an even better scheme that avoids specific overloaded caches. Load may be represented in different ways such as computational or network load. Load-awareness ensures that clients get their content from a cache that is not overloaded and therefore most probably faster.

All existing distributed web caching systems have their benefits and pitfalls (Section 2.1). Old web caching techniques, such as hierarchical based systems (e.g. Squid, Section 2.1.1) offer basic caching benefits but fail to be either locality or load aware. They also offer these benefits at high costs such as additional delays, distribution of redundant content copies, the introduction of bottlenecks and single points of failure.

Newer web caching techniques (Section 2.1.2 - 2.1.4) have tried and succeeded to some extent to be both locality-aware and load-aware. However, such detection mechanisms have had to be encoded into the system and cannot cope with dynamic behaviour as well as we would like. In addition, newer web caching techniques have failed to incorporate some beneficial aspects of a hierarchical based system. For example, newer techniques allow all caches to retrieve content from the origin if needed. Hierarchical based systems restrict this operation to only the parent caches that are near the internet gateway. Such a restriction intuitively minimizes bandwidth congestion.

The aim of this project is to design (Section 4), implement (Section 6) and evaluate (Section 8) a scalable distributed web cache that maps clients to an appropriate web cache using a promising new mechanism, network coordinates (Section 2.2). Network coordinates gives each machine (also known as a node) a coordinate within a virtual n-dimensional space, based on repeated round-trip time (RTT) measurements to other nodes. The nodes can then be arranged so that their distances to one another are defined as latencies in some time unit (usually in milliseconds).

Network coordinates are potentially very powerful for applications such as distributed web caches as they embed locality information within their structure. This opens up a set of well-defined geometric solutions to problems that were not so intuitive beforehand. This now means that caches and clients can be arranged in a virtual space, giving clients the opportunity to locate their most appropriate web cache (one with the lowest latency and therefore generally closest in terms of routing paths) with a simple nearest neighbour lookup. Equally, caches can
ask their neighbours if they have content cached, reducing total request times and still keeping data local. The use of network coordinates generates what is called a proactive network-aware overlay (Section 2.2.1). Proactive overlays tend to be more efficient as measurements are done periodically. Reactive overlays are however more accurate as measurements are done for each request.

Although network coordinates appear to be a great solution to many applications, it is important to stress they are not perfectly accurate. Increasing the number of dimensions in the virtual space does increase accuracy of node placement but with higher overheads. Still, they are no match in terms of accuracy to systems such as OASIS (Section 2.1.3) and Meridian (Section 2.1.4) which does on-demand probing (reactive measuring). That being said, for a distributed web cache, the most accurate answer is not required as HTTP requests are generally short-lived and are generated in large bundles making the overheads of having perfect accuracy too high. However, network coordinates can be used to repeatedly find a good answer, increasing its efficiency on every request due to fixed overheads.

Network coordinates also have another problem associated with them. As already described, network coordinates provide a large amount of locality based information. However, locality-aware systems will map clients in dense areas such as cities or even countries with intense internet usage to similar caches, most probably overloading them. Equally, caches placed in highly unpopulated areas would be under using their resources. It is for these reasons locality-awareness should always be accompanied by load-awareness in a distributed web cache.

An extension to this project is to do exactly this. It would be hard to justify network coordinates as a suitable mechanism for distributed web caches if no load balancing was taking place. The aim of the extension is to try and embed a load metric into the already existing network coordinate mechanism. This will not only render network coordinates more promising in this application but could be used in many others.

So far, network coordinate latency measurements have been taken with PING requests (ICMP ‘echo request’). PINGs are sent over connection-less links and estimate the basic round-trip time (RTT) between two nodes with the use of packets, stating any packet loss. However, application-level transmissions (such as with the use of TCP) are connection oriented and reliable, resending any packets that are lost. The Internet Standard treats packet loss and congestion as synonyms. When data is sent through routers which cannot be stored or transmitted it is lost. This is due to the average sum of inputs exceeding its output rate (network congestion), resulting in packet loss.

If we were to replace the standard PING round-trip time measurements with an application-level measurement, we would not only get the effects of network load but computational load also. If we sent some data over a TCP connection to another node on our system and an application then received this data and sent it back, the RTT calculated would be dependant on the computational load of both nodes. This is because some process (the application) needs to be scheduled on the CPU in order to actually receive and send the data. This new scheme will now give us new meaning to the distances between nodes, as they are now application-level latencies and therefore a much more realistic and useful measurement for selecting the best node to communicate with.

After such transformations, network coordinates should be an ideal and ground-breaking mech-
anism for applications such as distributed web caching.

*Content location* and *replication* are yet another two very crucial properties that most traditional web caches have. *Content location* enables caches to locate data within the system (not locally cached) very quickly, improving the system’s global cache hit rate. *Content replication* alleviates the load on a cache by copying content across to other caches, so to give users additional points of access.

One potential problem with a network coordinate based system is that we no longer have these two properties. We could in theory build a content location mechanism on top of the system, but we would then be disregarding all locality and load awareness information if pursued. The proposed system will therefore incorporate these properties without compromising its locality or load awareness.

Due to the developed distributed web cache now having locality/load awareness and content location/duplication properties at low costs, it should be able to beat today’s and tomorrow’s internet demands, overcoming the problems caused by growing internet congestion.

### 1.1 Contributions

An outline of the contributions of this project can be seen below. The project aims to deliver these and justify them with the evaluation of the system built.

- Extension of network coordinates to incorporate load-awareness properties.
- Development of a fully working scalable distributed web cache.
- Web cache is modular in design and extensible.
- Web cache is both locality and load-aware, due to the extended network coordinates.
- Web cache has a suitable content location and replication scheme.
- Development of a visualization module in order to see the virtual coordinate space.
- Development of testing modules such as a client request generator.
- Evaluation of the extended load-aware coordinates.
- Evaluation of the distributed web cache against a number of different strategies.
Chapter 2

Background

Reviewing and discussing all past work is beyond the scope of this project. Rather, this project aims to discuss the key sections that are relevant to the understanding and justification of the proposed designs and extensions.

A thorough investigation into already existing systems (Section 2.1) will help us devise a design for our distributed web cache. Starting with a traditional distributed web cache and moving onto some state-of-the-art implemented systems. This will help appreciate the foundations of this area of study, let us see what we are competing with and recognize what we can evolve and improve on.

Following this, we will then investigate Network Coordinates further (Section 2.2); a promising mechanism for an application such as a distributed web cache. We will try to place network coordinates into an overlay network category (Section 2.2.1) to better understand their properties. There are two main types of network coordinates that we will briefly discuss (Section 2.2.2 & 2.2.2).

An example of a simulated-based algorithm (Section 2.2.3) will be investigated as well as the accuracy and stability of network coordinates (Section 2.2.4). There are a number of routing schemes out there but we will be investigating one more relevant to network coordinates (Section 2.2.5). Routing is an important feature for collaboration within overlay networks.

As distributed web caches require some sort of mechanism to locate data, we will discuss what those may be and how they depend on the overlay network chosen (Section 5). Once the system has been designed and built, it will need to be tested and evaluated. This will be done in both closed environments such as a local-area network and in a renowned testing platform called Planet Lab (Section 2.4). Planet Lab is a research network that supports the development of new network services. There are a number of pitfalls and myths which will be clarified at a later stage.

An open source network coordinate library called Pyxida (Section 2.6.1) will then be reviewed as it implements the core component within our distributed web cache system. Finally, the languages, supporting applications (Section 2.5) and libraries (Section 2.6) will be briefly discussed, giving an overview of what was used to design, implement, test and evaluate the system.
2.1 Distributed Web Caches

Web caching is the action of copying and migrating data to web cache servers in an attempt to give users other points of access and reduce the load on the origin web server. However, it is only useful if two or more users request the same piece of information from the same web cache. Intuitively, this is because the web cache needs to ‘save’ the data on the first request (i.e. retrieve from the original source) and only on the second request can it start to return the cached item, alleviating the origin web server. However other techniques can be used in a distributed web caching system such as asking other neighbouring caches if they have the data cached.

A distributed web cache is simply a group of web caches that communicate amongst themselves in order to distribute content or route client’s requests to a more appropriate cache. When working with distributed web caches, a single web cache is also known as a node. A node may carry more functionality than an ordinary web cache, such as its ability to act as a server and client in order to communicate and collaborate with other nodes throughout the network. For a matter of reference, we will initially introduce a traditional distributed web cache called Squid [4] which has a hierarchical structure.

2.1.1 Squid

Squid [4] is a hierarchical web proxy cache which was built from the Harvest project [5]. It is implemented as a set of independent caching servers organized in a logical parent/child and sibling hierarchical manner. They share their contents by using a simple multicast query mechanism with the help of the Internet Cache Protocol [6].

Caches closest to the Internet Gateways (i.e. backbone transit entry-points) act as parents to caches topologically further away. When a child cache has a cache ‘miss’, the query is passed onto its parent. The parent then either has the content cached (in which case, it is passed down the hierarchy) or it goes to retrieve it. This way bandwidth is preserved on the backbone transit links as only parents retrieve data. It also means content is disseminated to lower levels in the hierarchy, giving all children higher ‘hit’ counts and ultimately reducing load on participating web servers.

Squid uses siblings (caches at the same level in the hierarchy) to distribute server load. When a cache has a local ‘miss’, a multicast query is sent to its parents and siblings. Siblings that receive this request do not then propagate it onward.

Squid has a number of disadvantages. Firstly, caches do not include the content of their descendants, ‘cousins’ or cousin’s descendants. This is because queries can only be sent up the hierarchy and not down. This means that caches could have a high false ‘miss’ count. Secondly, fetched objects may pass through a number of intermediate caches on their way back to the original requester. This can potentially give high latencies and a higher hop count. Thirdly, multicasting messages increases load on all caches. Fourthly, in order for the system to have optimal performance, knowledge of the topology is required in order to place caches correctly within the hierarchy. Lastly, due to such high loads because of multicasting, ICP packets could be dropped, giving some requests huge latency penalties.

The following systems are considered the state-of-the-art and have achieved good results in respect to request response times, data throughput and accuracy of chosen node.
2.1.2 CoralCDN

CoralCDN [7] is a peer-to-peer content distribution network which was built to alleviate web servers from huge demands and what is known as the 'Slashdot' effect [2]. With the help of cooperating web cache nodes, CoralCDN minimizes the load on the original web server as well as avoiding the creation of hot spots. CoralCDN is an application of a system called Coral, which is what gives it these properties.

Coral is a latency-optimized hierarchical indexing infrastructure based on what is called a distributed sloppy hash table (DSHT). DSHTs are slightly different to distributed hash table (DHT), which are used in many traditional peer-to-peer distributed systems [8, 9]. Conventional DHTs map single keys to single values in a random evenly distributed fashion with no regard to locality or hot-spot prevention. DSHT’s are; locality-aware, prevent hot-spots, tree saturation and provide a weaker consistency than DHTs do (giving them the added sloppy adjective in their name).

DSHTs are designed to store multiple values under one key (soft-state key/value pairs). This permits systems such as CoralCDN to do a number of useful lookups such as finding a set of web cache proxies that are storing a particular web object or locating a client’s nearby nodes to minimize latency.

Coral nodes are assigned 160-bit IDs (generated by hashing their IPs) and keys are assigned IDs in the same identifier space. A measurement of closeness is defined as the distance between a node’s ID and a key. Given a key and value, the value is placed in the identifier space close to key and when searching for a stored key/value pair; the search gets sequentially closer to the key. However in order to place a key/value pair in the identifier space, there needs to be some mechanism for nodes to route to a particular key. Each node in a DSHT is therefore required to have a routing table.

The node’s routing tables have a property; that for any key \( k \), a node’s routing table \( R \) allows it to find a node closer to \( k \), unless \( R \) is already the closest node. This is accomplished by using the XOR metric (based on Kademlia [10]) with IDs. When using the XOR metric, longer matching ID prefixes give a smaller binary number and therefore making them numerically closer. This XOR metric allows each node at every hop to calculate which next node is numerically closer to the key it is trying to search for. Therefore when a request is made there is a sequence of nodes visited with monotonically decreasing distances to the key we are searching for. As all routing tables are logarithmic in size in respect to the total number of nodes (i.e. \( O(\log N) \)), \( N = \text{Total Number of Nodes in DSHT} \), so is the hop count in reaching a key.

Tree saturation is a common problem with DHT based structures as it is often the case that nodes near a frequently accessed key are prone to high levels of traffic. However, hot-spots and tree saturation are avoided in DSHTs by a clever sloppy storage and retrieval technique. When a key/value pair is stored in the identifier space, it is not necessarily stored at the closest node to the keys location, but instead it may be placed close to it. This is achieved by using a variation on the routing technique described before, which is to route to the node near the key and terminating the search prior to reaching the closest node. The key/value is then stored at this early found node, relocating data away from the congested zone. However in practice Coral has a much more complex method with a forward and reverse phase in order to keep the central hot-spot nodes fresh and propagate the data further outward only when needed.
Retrieval of the key/value is done by routing towards the key and returning a value when the search terminates. If the value is returned prior to reaching the closest node then this is the works of the sloppy storage procedure.

The locality-awareness of Coral is achieved with each coral node belonging to several DSHTs called clusters. These clusters are specified by a maximum RTT called a diameter. The system has a fixed number of diameters known as levels. Each node is within one DSHT at each level. Groups of nodes that have a RTT below a threshold can join the appropriate level-i cluster. Coral then queries nodes in the higher-levels before the lower-levels (i.e. faster clusters before slower ones). This intuitively reduces the request latency by prioritizing nearby nodes.

Coral’s design is both locality-aware and load-aware. However locality-awareness has not been embedded into the structure of the scheme. Instead, Coral uses this knowledge to prioritize local nodes giving an increase in performance. Coral’s unique DSHTs provide neat load balancing features and disseminate content away from locations with large loads. However, it is clear from results that its notion of multi-level local awareness has given CoralCDN a good increase in performance for the client.

![Figure 2.1](image.png)

Figure 2.1: End to End client latency for requests of Coralized URLS, comparing the effects of single-level vs. Multi-level clusters. Top graph includes all nodes and bottom graph includes nodes in Asia. Taken from [7].

Figure 2.1 shows the end-to-end client latency with single and multi-level clustering, and quite clearly shows the benefits of such a scheme. The graph on the left shows the results of their experiments on all nodes whereas the right shows only the nodes in Asia, emphasising the performance benefits of clustering.

### 2.1.3 OASIS

OASIS (Overlay-based Anycast Service InfraStructure) [11] is a distributed anycast system. When a service is registered with a group of servers, client can ask for a good server for it to contact. OASIS selects a ‘good’ server based on network locality, liveness and apparently, load.
Other techniques probe clients, helping it decide which server for it to contact. This probing can be done repeatedly in the background or what is called on-demand probing (with every request). One shortcoming of systems such as Meridian [12] is that on-demand probing adds extra latency to each request, as well as potentially triggering intrusion-detection alerts on the client. However, on-demand probing does increase accuracy of the decision and the extra latency is minimal in proportion to savings if the requested content is relatively large in size or the communication link is long-lived.

Before OASIS is deployed, a predefined set of different portions of the internet (based on their IP prefixes) are mapped onto geographic coordinates of their nearest known landmark. Probing is then only done to each prefix [13] (not each client), lowering the total number of probes made throughout the system. The measurements obtained by the probing are then used to map the prefixes onto a location. These new mapped locations are then used to give the system its locality awareness. In order to maintain its locality information, OASIS requires a type of stable coordinate system. Intuitively, geographic coordinates were chosen as they did not have the criticisms associated that network coordinates have [14, 15, 16].

Advantages of using geographic coordinates are that they are stable and allow for infrequent probing of prefixes. However, geographic coordinates may give incorrect semantics to routing distances, as routing paths may take a diversion due to link closures etc...

As mentioned before, OASIS probes to find the closest service (replica) to a prefix. The RTT between a prefix and its closest replica is also used as a heuristic in deciding when to re-probe the prefix for a better replica. This gives OASIS a type of absolute error predictor, as re-probing prefixes with higher RTT increases accuracy. It is also important to note that OASIS uses Meridian as its background mechanism for probing.

The OASIS system consists of core nodes that help clients select the most appropriate replicas. Nodes can however have multiple roles such as being core nodes as well as replicas. When a client makes a request for a service, a core node maps the client’s IP address to what is called a network bucket. The network bucket places closely related IP addresses into netblocks (range of IP addresses). The network bucket is then mapped onto a geographic location as a coordinate. If this mapping is done successfully then the ‘best’ service replica is returned. If however this process fails then a random replica is returned. OASIS has mechanisms for mapping such IP addresses to network buckets (bucketing tables) as well as to map buckets to geographical locations (proximity tables).

The results indicate that OASIS has low RTTs, high throughput, low DNS resolution time, and high end-to-end performance with low latency. Although hard to directly compare to other techniques, OASIS is quite competitive. With lower overhead (due to no on-demand probing) than the leading technique Meridian but similar accuracy in decisions made.

However, although locality-awareness is a key building block in the mechanism, load-awareness is not. Developers of OASIS have additional algorithms to support load-awareness but believe that there is no need for it to be perfect. They say a reasonable good node, in respect to load, is acceptable. In today’s world, I believe load-awareness is an essential property within distributed web caching systems.

OASIS was developed on the assumption that geographic location mappings are accurate enough
for locality knowledge, and has proven just that. For applications such as web caching, decisions may be too accurate (with respect to locality) as file sizes may not be large enough to outweigh the costs of finding such a good answer. It also means that if there is good locality awareness and poor load balancing, clusters of clients within highly dense areas (such as cities) will be accessing the same servers even though they will be overloaded.

2.1.4 Meridian

Meridian [12] is yet another peer-to-peer overlay network scheme that aims to do locality-aware node selection. Meridian does not have locality embedded into its structure, but uses on-demand probing to estimate distances of nodes. In order to find the nearest node to a client, Meridian uses a set of PINGSs to step forward logarithmically closer to the target.

Each node has a fixed number of links to other nodes and organizes its peers into rings of exponentially increasing size. Heuristics are used to calculate ring membership in order to maximize each node’s performance within the network. When a query is made by a node, the message hops closer to the target. At each hop; a measurement is made to calculate the latency to the target, requests are then sent out to all neighbours in the target’s ring and half of the nodes in the adjacent rings to also probe the target. The neighbour with the shortest distance is then selected. This process then repeats recursively, minimizing the latency at each step exponentially.

Clearly the number of nodes within each ring is a trade-off between accuracy and overhead. The more nodes within a ring, the more information a node has to make better routing decisions however at higher overhead costs due to more probing.

Both Meridian and OASIS were designed to be incredibly precise. It is for this reason they are often used in applications that require frequent contact, outweighing the cost of finding the best node. For applications such as web caching, finding the best answer is not necessary.

Network coordinates offer a low-cost mechanism with relatively good levels of accuracy (but not perfect). They aim to achieve aggregate savings by repeatedly finding a good answer to queries quickly. Network coordinates also embed network locality information into their structure making it very simple to find nearby neighbours.

2.2 Network Coordinates

2.2.1 Network Overlays

For a decentralized distributed web caching system to work there must be a number of web caches (nodes) that communicate and collaborate. The structure and design of the network in which these nodes reside is what is called a network overlay. The strategy for deciding what links to each node to keep, (adopted by each node) leads to the creation of a network overlay. It is called a network overlay because it is built on top of an already existing network, in our case the internet.

A network overlay is a strategy which each node implements in order to choose the other nodes to adopt as neighbours. The various network overlay schemes can be categorized into three ma-
jor groups [17] and have evolved: network-oblivious overlays, proximity-aware overlays and network-aware overlays, respectively.

Network-Oblivious Overlays

Network-oblivious overlays create links to neighbouring nodes based on identifiers in a logical space.

The most common is the structured distributed hash-table (DHT), whereby each node is assigned a random logical identifier. Nodes keep links with their numeric neighbours in the logical space as well as keeping a few long distance nodes to reduce the overall hop count when routing.

As the neighbouring nodes are chosen at random it is very probable for neighbours to be physically very far apart, making routing inefficient. This is especially true if much of the traffic is local as there is no notion of proximity or locality-awareness. However, the advantage of such a scheme is that the load is evenly distributed throughout the network.

Some very well known applications of structured distributed hash-table (DHTs) are Chord [8] and CAN [9].

Proximity-Aware Overlays

Proximity-aware overlays do not base their structure on node proximity knowledge but use it whenever a node has a choice between linking with a number of nodes. Proximity-aware DHTs such as Pastry [18] and Tapestry [19] prefer to include nodes that are physically close in their routing tables, improving network efficiency. However they are still based on a logical identifier space with nodes randomly placed, and efficiency is only included when a choice is present.

Network-Aware Overlays

Network-aware overlays move away from logical identifier spaces and create a network structure that is based on the physical distance of nodes. They exploit a fundamental property to increase the quality of service for local traffic. This property is called the locality principle [20], which states that network traffic that is local should stay local. Amongst local traffic, quality of service features such as latency, bandwidth and reliability improve due to a lower hop count.

Such overlays require there to be some type of measurement being done between nodes to calculate these distances. There are two types of overlays that deal with this dynamism; reactive and proactive overlays.

Reactive overlays initiate measurements between nodes whenever a message is routed. Proactive overlays decouple measurement overhead from overlay usage by periodically measuring in the background. Clearly if the overlay usage is high then reactive overlays will have a larger overhead but with fresher measurements. Proactive overlays run the risk of working on stale measurements as the network structure may have changed since the last measurement. However, they in turn do not suffer from high overhead under high usage.
A good example of such a reactive network-aware overlay is one that has been mentioned before, Meridian (Section 2.1.4).

Due to the embedding of latencies into a coordinate space and the periodic measurement scheme, using Network coordinates yields a proactive network-aware overlay.

### 2.2.2 Network Coordinates

Network coordinates give each node a synthetic coordinate (of \(n\)-dimension, typically \(n = 2\) or 3) based on the latency between other nodes. Each node keeps track of its own coordinate updating its position in the latency space. This is done by obtaining RTT measurements to a subset of other hosts within the overlay network. The use of network coordinates yields network-awareness properties as it will position nodes that have lower latencies between them closer to one another, obeying the locality principle [20].

![Figure 2.2: A latency space on PlanetLab. Taken from [17].](image)

Figure 2.2 shows a 3-dimensional latency space constructed with Vivaldi (Section 2.2.3) on 115 North American PlanetLab nodes. As you can see, clusters are being formed around geographical regions within North America, showing how latencies give a good estimation of geographic locality. However, this figure is actually showing network routing locality information, a much more useful metric.

Network coordinates give a number of advantages to any application using them [17]:

Firstly, measurement overhead is greatly reduced as a node does not need to measure its distance to other nodes to be able to approximate its distance (and therefore latency) to them. Once given a coordinate, nodes have access to a great deal of locality based knowledge. As described beforehand (Section 1), locality has a huge number of advantages.

Secondly, what once required a specialized solution now can be accomplished with a whole array of well-defined algorithms within the field of geometry and coordinate systems. Problems
such as looking for the best node to communicate with, are now solved by a simple nearest neighbour query.

Thirdly, network coordinates can be used to unify both wired and wireless networks. Wireless networks are based on a geographical communication model, restricting nodes to communicating with neighbours only. Network coordinates have similar properties and therefore allows wired networks to use algorithms developed for wireless networks [21].

Lastly, network coordinates can cope and adapt to network changes. Nodes update their own coordinate and re-position themselves iteratively. Other schemes such as GNP [15] do not have this property, as certain nodes remain fixed within the virtual space.

It is quite easy to see how a distributed web cache could benefit from the power of network coordinates. Mapping clients to their nearest cache would simply need for the clients network coordinate to be calculated and doing a nearest neighbour lookup.

There are a number of methods for measuring latencies to calculate network coordinates. These mechanisms can be divided into two categories; Landmark-based and Simulation-based calculations.

**Landmark-based**

Landmark-based schemes use a fixed number of landmark nodes for other nodes to calculate their distance from. For example, GNP [15] makes its nodes calculate their distance to multiple landmark nodes throughout the network. A clear disadvantage to this scheme is that landmark nodes could become a bottleneck as all nodes must contact the same landmark nodes, potentially overloading them. Another problem is that the accuracy of such systems tends to rely on the choice of landmarks.

Landmark-based systems also have the problem of having single points of failure. Because of this, research in landmark-based systems evolved and systems such as Lighthouses [22] were developed. The Lighthouses scheme does not provide a set of fixed landmarks. Instead, each node creates its own local space from a set of random nodes called lighthouses and then uses a shared transition matrix to convert local spaces into global ones.

**Simulation-based**

Simulation-based schemes tend to be preferred as they are more decentralized. Simulation-based schemes calculate coordinates by modelling nodes within a physical system. Examples of such systems are Vivaldi [14] and Big Bang Simulation [23].

**2.2.3 Vivaldi**

Vivaldi [14] being the most popular simulation-based calculator, uses spring-relaxation to attract and repel nodes according to their network distance measurements (latency). It gives you the ability to predict RTT without prior communication.

Vivaldi does not provide a coordinate space with exact RTT predictions, as there are a number of inaccuracies (Section 2.2.4). The Vivaldi algorithm instead tries to find coordinates that
minimize the error of predictions.

Let $x_i$ be the coordinate assigned to $i$ and $x_j$ to $j$. Vivaldi characterizes the errors in the coordinates using a squared-error function shown below, where $\|x_i - x_j\|$ is the distance between coordinates of nodes $i$ and $j$ in the coordinate space.

$$E = \sum_i \sum_j (L_{ij} - \|x_i - x_j\|)^2$$ (2.1)

Equation 2.1 is what Vivaldi tries to minimize throughout the system. Minimizing the squared error function is equivalent to minimizing the energy in a spring network. The error function is therefore suitable for use in a physical mass-spring system.

Due to Vivaldi’s choice of $E$, we simulate our network as a set of springs between each pair of nodes. The current length of the spring is the distance between the nodes within the coordinate space. The rest length (relaxed length) of the spring is set to the known RTT and the potential energy of a spring is proportional to the square of the difference of its rest length and current length. The sum of all the energy from all springs in the network is exactly what we wish to minimize, $E$.

---

**Algorithm 1** A simplified distributed Vivaldi algorithm, with a constant time step $\delta$. Taken from [14].

```plaintext
// Node $i$ has measured node $j$ to be rtt ms away.
// and node $j$ says it has coordinates $x_j$

simple_vivaldi(\texttt{rtt}, \texttt{x}_j)

  // Compute error of this sample.(1)
  e = \texttt{rtt} - \|\texttt{x}_i - \texttt{x}_j\|

  // Find the direction of the force the error is causing.(2)
  \texttt{dir} = \texttt{u}(\texttt{x}_i - \texttt{x}_j)

  // The force vector is proportional to the error.(3)
  \texttt{f} = \texttt{dir} \times \texttt{e}

  // Move a small step in the direction of the force.(4)
  \texttt{x}_i = \texttt{x}_i + \delta \times \texttt{dir}
```

Algorithm 1 shows a simplified version of the decentralized Vivaldi algorithm. The unit vector $\texttt{u}(\texttt{x}_i - \texttt{x}_j)$ gives the direction of the force on node $i$. Whenever a new measurement is received from a remote node, the RTT ($\texttt{rtt}$) and the remote node’s coordinate ($\texttt{x}_j$) is fed into the algorithm. The algorithm is a decentralized one which shows how each node updates its own coordinate. As RTT samples arrive, Vivaldi calculates the error of the sample pushes its coordinate for a short period of time, minimizing the error in respect to the other remote node. Clearly as a node communicates continuously with other multiple remote nodes, its coordinate converges.

**Line 1** of the algorithm simply shows the error to the remote node being calculated. The error calculation is then used to calculate in which direction the node will move; **line 2** is finding the direction of the vector and **line 3** the force of it. **Line 4** shows the nodes coordinate being recalculated, taking into account the direction and force of the vector. It moves for a small
constant time step $\delta$.

If we fed measurements from all the remote hosts within the network into Vivaldi, the calculation would most probably be very slow. If however we only calculate our coordinates based on a handful of remote hosts, measurement complexity reduces to $O(\log n)$; becoming much more feasible.

A stability issue arises when nodes only contact their immediate neighbours. What tends to happen is that coordinates have a twisting effect (also known as folding) due to not having any anchoring with further nodes in the system. Results show that if nodes are in direct contact with other further nodes, they then obtain a higher coordinate accuracy at a larger scale. There are many optimizations to the Vivaldi algorithm, such as an addition of a height vector into the already defined Euclidean space to model the time a packet takes to travel the internet access link.

Results show that when compared to GNP [15] (landmark-based system), Vivaldi has a competitive prediction error. However, both systems are hard to compare and are good for different applications.

### 2.2.4 Accuracy and Stability

Applications of network coordinate have been performing well in simulated environments such as Planet Lab (Section 2.4). However in practice, they seem to fall short of producing stable and accurate coordinates. Issues such as unstable latency measurements taken from the same two nodes and repeated violation of the triangle inequality property causing oscillation, all contribute to their instability. These issues will now be described in more detail.

#### Churn

Churn occurs when coordinates are not stable due to high dynamism within the system. Although churn was already noticeable within simulated experiments, it has proven to be more of a problem in practice [24] with systems such as Azureus [25]. Many techniques can be used to stabilize them such as performing measurements multiple times and retrieving a local minimum before updating local coordinates. Such a scheme is what is used in the Pyxida Library (Section 2.6.1).

#### Triangle Inequality Violations

The triangle inequality property states that for any triangle, the length of a given side must be less than the sum of the other two sides but still greater than (or equal to) the difference of the other two sides. When the latencies between three nodes cannot form a triangle, they are said to have violated the triangle inequality property. As latencies throughout the internet do not always obey this, embedding them into a Euclidean space will give some inaccuracy [26]. Results show that triangle violations very much exist, with around 18% triangle violations between 399 PlanetLab nodes [24]. However, recent research shows that violations can be overcome and stable network coordinates can still be produced [27, 24].
Drift

Although coordinates might reflect the true relative latency distances between nodes, absolute coordinates might not. Over long periods of time, network coordinates drift from their origin with relative distances amongst nodes remaining constant [27]. Drift has a number of disadvantages, one being that it limits the amount of time a coordinate will remain a useful locator. This does not directly influence the systems measuring but may be a problem if applications of network coordinates were to use such locator coordinates. For example, the issue may occur if using a routing technique dependant on locator coordinates (Section 2.2.5). A proposed solution is to incorporate a gravitational pull towards the origin. Results of this technique show that it successfully stops drifting without harming the systems accuracy [24].

Latency Variance

Inevitably, latency measurements of the same two nodes over a short time span, gives a wide variation of results. This can pose a problem with systems that are very dynamic, with nodes entering and exiting frequently.

2.2.5 Routing Schemes

Within a distributed web cache, there needs to be a strategy for nodes throughout the overlay network to communicate and exchange messages. This strategy must try to minimize the hop count as well as the delay stretch, and is known as a routing algorithm.

Routing is essential to numerous issues when developing such a system. For example, if a client is attempting to find the closest cache, then the client needs to do a nearest neighbour search.

The most hop count efficient technique would to intuitively have links with every node in the network creating a fully connected overlay network. This means only one hop would be needed for a message to be sent anywhere. However, this system would not be very scalable as each node would require $n$ links.

A number of different routing strategies exist, however when dealing with network coordinates it almost seems sensible to use a kind of sense of direction in the routing strategy. A strategy called $\theta$-routing [28], takes advantage of techniques already developed for network geometry in the context of wireless networks [29, 30]. The routing techniques used in such wireless networks, use neighbouring nodes and a sense of direction in order to provide an efficient algorithm. Wireless networks are however restricted to only communicating with other nodes within their locality, which we are not. This means we can reduce the hop count when routing by taking some long distance shortcuts. In order not to over complicate the scheme, we will describe $\theta$-routing working within a 2-dimensional space.

$\theta$-routing gives each node has what is called a $\theta$-graph spanner. The nodes simply subdivide the area around it into $\frac{2\pi}{\theta}$ sectors of angle $\theta$. If a remote node lies within a sector, the remote node is entered into the local nodes routing table. It then forwards any messages by sending them to the neighbouring node within the sector of the targets location. Advantages are that routing tables are small and paths taken to send messages have a low delay stretch as nearest neighbours are organized in relation to latency measurements. Although in theory it is a nice
idea, the routing has a linear hop count proportional to the diameter of the network. This is because we are not taking into account any long-distance nodes within a direction, making a message pass through all nodes linearly to get to a target location.

The improved technique recently developed is called the scaled $\theta$-routing scheme ($\hat{\theta}$-routing) [31], which makes the assumptions that firstly, all nodes have a coordinate within a Euclidean space and secondly, that nodes do not have a maximum communication range. It is these assumptions that make it applicable to our wired overlay network with coordinates and not for a wireless network.

$\hat{\theta}$-routing takes the originally defined $\theta$-graph spanner and adds exponentially increasing rings around it to also take into account long distance nodes. The routing tables are now divided up into $\frac{2\pi}{\theta}$ sectors and $r$ rings. The areas that are crossed by the sectors and rings are called zones. Remote neighbours are now placed within each zone of the local nodes routing table, giving it the ability to skip intermediate nodes that are very nearby within the coordinate space. Not only does this strategy reduce the hop count to a logarithmic complexity but it also means that the network coordinate twisting (or folding) problem is implicitly solved (assuming the links are also used for retrieving measurements). However, the drawback is that the routing tables have a substantial increase in size.

![Figure 2.3](image.png)

Figure 2.3: Shows how $\hat{\theta}$-routing subdivides a coordinate space into sectors and rings. Also shows how a node routes to a destination by calculating the zone of the destination and greedily forwarding the message to the furthest hop. Taken from [31].

Figure 2.3 shows just how this subdivision of a coordinate space is done and how a message is greedily routed to a destination.

The technique is also approved for dimensions that are higher than 2 or 3. However, the assignment of nodes to zones poses a problem with such a high number of dimensions. We will not go into this and simply take the fact that its possible and is very similar to that already explained.

Now that we have explained the routing technique, it is important to note how a node acquires and maintains knowledge of its nearest neighbours. The mechanism used to do this is one taken
from a proximity-aware overlay called Pastry [18]. Rather than exchanging routing table knowledge with a remote node, the local node emits a query for its own coordinate to the remote node. All nodes on the path of this query then add their routing tables to the message, which then arrives at the target node. The target node now has all the routing information added by others, including at least one nearby neighbour. This nearby neighbour has a routing table that is similar to what the local nodes new routing table should be. It is in this way a node acquires knowledge about its vicinity for its routing table. Results show that the number of sectors has a strong correlation with the efficiency of routing. It also shows that finding the true nearest neighbour to a node is almost entirely dependent on the accuracy of the network coordinates within the system.

Figure 2.4: For 2-dimensional routing with 'perfect' routing tables, far reaching rings and fewer anglers result in a zigzag behaviour across the coordinate space. Taken from [31].

Figure 2.4 shows a heat map evaluation of the routing efficiency. It is done by giving nodes 'perfect' routing knowledge and then measuring the hop counts and delay stretch of queries made across the network. As you can see there is a strong correlation with both the number of sector and rings to the hop count. However the maps also show that if the number of rings is increased, the delay stretch can also increase. This is because with long edges (large r), messages tend to zigzag across the network instead of taking a more direct route. Intuitively the more rings that are added into the graph, the larger their circumference will be, giving them a choice of a wider range of nodes to choose from as their neighbour. This is what creates this zigzagging effect when routing.

2.3 Web Caching

The most obvious component of the system is the actual web cache itself. Although not the core of this investigation; how a web cache actually goes about caching data, the kind of data we should expect to find when using web caches and as a result the best replacement strategies are all important factors when designing a distributed web cache.

2.3.1 How a Cache Works

Web browsers placed on client machines are what generate HTTP GET requests for internet objects such as HTML pages, images, large image files etc... Depending on which has a fresh copy of the object, the requests are then picked up by either the local web browsers cache, web
caches or by the origin web server. When a web cache receives a GET request, there are three possibilities;

1. The object that is being requested is uncachable.
2. The object is not found in the cache (cache miss).
3. The object is found in the cache (cache hit).

In the first case, the request is forwarded to the origin web server. The second case provides the cache with a choice to either; ask other caches for the object or simply retrieve the content from the origin server itself. In the final case, the object found is tested for freshness. If fresh then the object is simply returned to the client, otherwise a conditional GET (cGET) request can be made to collaborating web caches. There are two main types of cGET requests; an If-Modified-Since which has a timestamp of the last known modification, and an If-None-Match request with an ETag representing a server chosen identifier of the object contents. The cGET request can be serviced by either another web cache or the origin web server. A web cache that receives the cGET request and does not have a fresh copy of the object tends to forward the request to the origin web server. The response to the request is either the entire object or a not-modified message if the content hasn’t been changed [32].

The freshness of a particular object is recognized by an expiration policy within the web cache. This is generally based on the time-to-live field generated by the origin web server or calculated by the web cache itself based on the last modification time. The object is stated stale when the time-to-live field is out of date.

2.3.2 Replacement Strategies

Web caches have a limited amount of resources and cannot cache every single object requested. That is why a cache typically has a replacement strategy to deal with such constraints. However some strategies work better than others in different applications, which is why web caches seek to use the strategy that minimizes the average miss count. Studies showing that the relative frequency with which web pages are requested follows a Zipf like distribution [33]. Due to these implications, a comparison was made between four replacement algorithms; Perfect-LFU, LRU, GD_Size and In-Cache-LFU.

Perfect-LFU is based on the LFU (Least Frequently Used) scheme but keeps a count on a cached object even after it has been eliminated. In-Cache-LFU erases all knowledge of the object once removed, incurring less overhead.

Although the comparison showed that GD_Size and Perfect-LFU algorithms performed the best in terms of hit ratio, they have significantly more overhead than simpler schemes such as LRU (Least Recently Used) or In-Cache-LFU. Out of these two simple schemes, LRU performs the best and therefore should be used.

2.4 PlanetLab

PlanetLab [34] is a global research network comprising of approximately 840 nodes at 420 sites that supports the development of new network services. It is simultaneously used by a
number of researchers for a number of experiments ranging from short-term to continuously running services.

PlanetLab however does have a number of disadvantages to it [35];

Firstly, it is said that results obtained from experiments set up in PlanetLab are not reproducible as it does not offer users a controlled environment. Instead, network services are subjected to real-world conditions with delay, churn and node failures. Your application is also placed in what is called a slice. Multiple slices are running on a single node, potentially slowing down your application.

Secondly, the network between PlanetLab sites does not represent the Internet. This is because a number of the sites within Planet Lab are dominated by global research and education network (GREN) which connects high speed access links across the world. This means that sites connected to one another do not reflect ordinary commercial links.

This may become a problem when modelling clients for our testing phase as clients clearly do not have access to such internet speeds and bandwidth. However as a platform to deploy the distributed web cache for long-term usage, PlanetLab may be appropriate due to such links between nodes.

There are also a number of factors that against popular belief are no longer true about PlanetLab. Such as PlanetLab being too heavily loaded as well as not being able to guarantee resources.

A study shows that by having good practice whilst implementing a distributed system for PlanetLab, problems such as measuring accurate latencies can be overcome [35].

2.5 Languages and Supporting Applications

2.5.1 Java

Java 1.5 is the version of choice as it is installed on most systems, making testing less of a hassle. Versioning in Java when using object serialization is very important as two different versions running on separate systems communicating can give you serialization errors. Unfortunately, it was later decided to install our own Java distribution on test nodes, to ensure no serialization problems occurred whilst testing. However, no recently implemented features in Java 1.6 were needed apart from generics which are also in Java 1.5.

Java offers a huge number of advantages for communication based applications such as a distributed web cache. Although it was necessary to use Java (because of Pyxidia being written in it), it would have been the language of choice. It is simple to understand and has useful features for communication such as object serialization and high-level communication libraries such as the HttpComponents library (Section 2.6.2).

Java also allows for unit testing (Section 2.6.3) on mock object which will be useful when testing.

2.5.2 Python

Python is a very powerful high-level programming language which has an immense standard library. The language itself emphasizes productivity and code readability. The reason for using
such a language in my project is due to three reasons;

Firstly, the API for the PlanetLab management system can easily be accessed with a simple Python program, rendering it very useful for the addition and removal of nodes on a slice.

Secondly, with the use of a python application called *vxargs* and shell scripts, commands can be made to multiple hosts in parallel, simplifying the process of deployment onto arbitrary number of nodes.

Thirdly and lastly, data saved by the log server requires analysis and interpretation. Python has great high-level libraries for parsing text files very quickly, storing and producing useful statistical results.

2.5.3 SMV

SMV (Symbolic Model Verifier) is a model checking program. Model checking is a method for formally verifying finite-state concurrent systems. Temporal logic formulas are used to express the model specification and efficient symbolic algorithms are then used to check if the specification holds by traversing the model. Model checking is very useful to show properties such as liveness and safety.

SMV is simple to understand and makes it very easy to encode simple communication protocol models.

2.6 Libraries

2.6.1 Pyxida Library

The Pyxida library [36] is an open source Java based library for network coordinates that integrates a number of ideas taken from previous studies [27, 24]. It will predict the latency between nodes in a distributed system with accurate and stable coordinates. The network coordinate functionality is given by the use of the Vivaldi algorithm (Section 2.2.3) which is a simulation-based technique.

The Pyxida library is good starting point for anyone trying to develop an application with simulation-based network coordinates. It allows you to run an already built application, to initially understand how the library works as well as use the core parts of the library to incorporate coordinates into your own applications.

The core library supplies a class called *NCCClient*, which is should be instantiated on each node and fed RTT samples from all other nodes. Most of the work is done behind the scenes but parameters will probably need to be tweaked in order to get the right balance between stability and dynamism.

When Pyxida calculates the local coordinate, a RTT sample is taken from each host the library knows about. The sample is then fed into the Vivaldi algorithm to minimize the error and place our coordinate into a low-energy state.

Pyxida also has a latency filter built into the system which helps improve accuracy and stability of the network coordinates (Section 2.2.4). Every sample entered into the Pyxida library is stored in sets, with each set corresponding to each remote host. The total number of samples stored for each host is depicted by the *window size*. The *minimum window size* is the number of measurements needed before we can use this host within our calculations for our own
coordinate. The maximum window size is the maximum allowed number of samples held for each host. When our own coordinate is calculated, a smoothed RTT sample is taken from each remote host’s set of samples. This single smoothed sample is a percentile of the samples stored.

Clearly, when altering the parameters there is a clear trade-off between dynamism and stability. For example, the higher the window size the more stable the coordinates will be.

In the library, the percentile parameter is set by default to 0.5 (i.e. median) and is set with protected access rights. This made me have to alter the original library source code and recompile it so that this can be altered. The window sizes are however set to public, allowing changes to be made wherever necessary.

2.6.2 HttpComponents Library

The HttpComponents library [37] (HttpCore) is written in Java and has implemented all of the most fundamental aspects of the HTTP protocol. It allows for a fully functional client and server side HTTP services to be built with minimal effort.

The library has proven to be invaluable in my implementation and deals with much of the core functionality of the system.

2.6.3 JUnit

JUnit [38] is an open-source testing framework written in Java. JUnit is by far the most popular of the testing suites and allows for quick-testing as well as test-driven development. For our particular application, most of our work will be communication based. This can be supported in JUnit with the use of mock object testing. Whereby you create mock objects for a test case and run whatever commands you wish, testing the state of the object after.

Such a testing suite will be very helpful in testing correctness of certain functionality and code snippets.

2.6.4 Prefuse Library

The Prefuse library [39] is a toolkit to create rich interactive data visualizations. It can be used to visualize data in a variety of formats but is mainly used for displaying nodes and links. The toolkit is in active development and has a very lively forum as a place of reference. It will be used to visualize the nodes deployed, updating their positions based on their coordinates. This would give some kind of dynamic view of the position of nodes within the virtual space.

However as mentioned before, network coordinates have height as part of their coordinate. Height gives nodes the opportunity to escape when being pushed from all angles. Unfortunately, Prefuse only works in 2D and the added height parameter in the coordinates makes it tricky to see the full picture. However, as height should not be considered an additional dimension but a fixed latency cost added to the distance, it could be represented by a colour on the 2D map.

This visualization tool kit can therefore be used to display and update our nodes showing all information including its position, links and nearest neighbours (Section 7.5).
Chapter 3

Overall Architecture

To investigate the potential of network coordinates in a distributed web cache, the entire system must be designed and built to be later analysed. Such a system must include a number of components for it to be complete and usable in a multi-user environment.

![Figure 3.1: The main components in a distributed web cache environment.](image)

The system involves three different sub-systems interacting as shown in Figure 3.1, namely the client, the cache and the origin web server. Intuitively, the cache receives requests and handles them by any appropriate method; such as checking its internal cache for the content and fetching it from the origin server if not found. The client is what generates the requests and forwards them onto an appropriate cache. My proposition is to use network coordinates to help find this appropriate cache. Lastly, the origin web server is where the original content is held and where the cache should retrieve and duplicate from.

3.1 Cache

The overall design of the application running on the cache machine is important to get right from the start of the project, so that to be able to render a modular design. The machines running the caching system will have three main components all interacting, which can be seen in Figure 3.2. The three component are actually part of one application but are shown segmented for easier understanding.

The **Cache Manager** component is what takes care of the storing and retrieving of cached content from a medium such as disk. It is what defines the caching strategy used when for example disk space is low and an item must be replaced. There needs to be a clear interface for the addition, checking and retrieving of content within the system.

The **Proxy Server** is the point of entry for HTTP requests. The server then handles these requests with an appropriate method. This method will involve interacting with the cache manager to check to see if the content is cached locally. The proxy server may initiate a new
connection to the origin web server to retrieve the content, caching it whilst returning it to the requester. The server may even decide to ask other caches about the content and therefore may interact with the Cache Node Manager.

The **Cache Node Manager** is where the implementation of the network coordinates is done. Its main task is to keep perfecting its position in the virtual space defined based on RTT measurements to other nodes. Different types of the manager may require it to do additional tasks, such as to keep a set of nearest neighbours up to date and to somehow gain knowledge of new nodes close by in the virtual space. This will be covered in more depth in the design (Chapter 4) and implementation (Chapter 6) chapters.

### 3.2 Client

The client machine is where requests are generated and forwarded to the most appropriate cache. In order for such requests to be captured and handled appropriately the client machine will need to have an application running. Figure 3.3 shows the three main sub-systems on a client machine, the Client Requester being a separate application from the Proxy Server and Client Node Manager which are bundled into one.

The user on the system will be generating requests with a **Client Requester**, which may be their favourite web browser. This application will need to be configured to go through its local proxy server which is easily done with any new or old browser. The local proxy server will be running on the local machine on a different port to that of normal HTTP servers.
The **Proxy Server** on the client machine is what captures these requests made and simply forwards them to an appropriate web cache. The web cache chosen is defined by interacting with the Client Node Manager.

The **Client Node Manager** has almost identical behaviour to the Cache Node Manager. All RTT measurements and network coordinate maintenance is done here. The Client Node Manager must also keep track of its closest cache neighbour. This is done so that the Proxy Server can forward requests to the close by cache.

### 3.3 Interaction

As already described, the different components within the architectural sub-systems interact heavily.

This interaction can be seen clearer in Figure 3.4 where the arrows show interaction between the sub-systems and components within. Note that the node managers in the diagram are also interacting with other nodes within the system not shown on the diagram.

The different sub-systems can interact in two ways. Firstly, they can interact through their proxy servers. This is how HTTP requests are forwarded and handled. Secondly, they can interact through their node managers. This is where RTT measurements are done and any other actions required for the maintenance of the node within the virtual space.
The components on a single sub-system also interact. For example, the client browser will send its requests to the local proxy and the local proxy will then forward that request on to the closest cache proxy based on the information taken from the local client node manager. Equally, the proxy server on the cache machine will receive requests and will want to check if such items have already been cached, interacting with the cache manager. Or it might decide to ask other cache nodes, picking up this information from the local cache node manager. Its last possibility is to get the content from the origin web server, as shown in the figure.
Chapter 4

Design

4.1 Multiple Designs

Before developing a distributed web cache, I decided it would be wiser to design and implement a centralized version of the system to not only get the lower-level communication right before moving on to a more ambitious system but to also be able to compare performance between the two. The design is essentially the same, however the need for the system to be modular and to be able to extend modules became an important aspect of the design. The differences in the design of the centralized version (Oracle system) and the distributed version can be considered more as policy decision, and can be seen below.

4.1.1 Oracle System

The oracle system is a centralized version whereby all new caches and clients register onto the system via a central oracle. This oracle keeps links up and running to all other nodes, keeping track of their position in the virtual space.

When a cache registers with the oracle, it requests a random set of online hosts. Links to each of these hosts in the set are then established. This is done so that the newly added cache can generate RTT samples between a handful of other hosts, stabilized and placing the host accurately in the virtual space.

When a client registers with the oracle, the same process occurs. However, the client also repeatedly asks the oracle who its nearest cache neighbour is. This information is then taken by the forwarding proxy server which then forwards any subsequent requests to this nearest cache.

This is a simple scheme which will allow for the basic communication protocols to be developed and tested. There are of course a number of disadvantages to this scheme which will be discussed later in the evaluation section (Section 8).

4.1.2 Distributed System

The distributed system will require no central process. Instead a node connects to a bootstrap node which can be any node already in the system. After a few RTT samples are generated and the newly added node is positioned in the system (potentially badly), it sends a routing
message to its bootstrap node. The message then gets routed towards the newly added node, visiting multiple nodes on its path. Each node visited, appends its routing table to the message. Once the message gets delivered back to the newly added node, it merges all the routing tables into its own and sets up links with any newly discovered hosts. These links enable it to calculate its position in the system more accurately as well as gain information about its nearest neighbours. The process is then repeated continuously with another randomly selected host in its routing table.

Intuitively this system requires no centralized node, as all that is needed is for all nodes to know at least one other node in the system. From this one node, knowledge of the system and nearest neighbours can be gained very easily.

The same routing procedure is done with clients. This gives it vital information about who its closest cache neighbour is, which can then be used by the Proxy Server. Each request the Proxy Server receives, then gets forwarded to the closest cache.

4.2 High-Level Managers

Each sub-system should have a general manager capable of executing high-level actions easily. However, each manager may have slight differences and therefore must extend a clearly defined general manager. The simplified UML diagram in Figure 4.1 shows how the hierarchy of managers and how they extend one another to carry domain-specify functionality.

Instead of breaking-up the caching mechanism into a separate module, it would be simpler to extend the generalized manager SystemManager into two types of managers; the CacheManager and the ClientManager. The CacheManager should hold all the needed implementation for the addition, checking and retrieving of local cached content. The ClientManager has no such functionality but rather offers a set of access methods to a nearest neighbour variable. The difference between all the concrete sub-classes will be how they spend their thread time. This might be by repeatedly asking around for neighbours or asking the oracle for the closest cache.

It is these managers that are started to get the sub-system up and running and they themselves hold and start-up the node manager and the proxy server. However in the oracle system, the OracleManager will have no proxy server as its sole responsibility is to keep other caches and clients updated with the links in the system and not to handle requests itself.

4.3 Node Managers

Much of the core of the project is within the node managers. As briefly described beforehand, the node manager will take care of the machines local coordinate within the virtual space. The RTT samples returned by each connection to a remote host will be fed synchronously into the Pyxida library. The Pyxida library (Section 2.6.1) offers a clean-cut interface to a simulation-based implementation of network coordinate generation. It uses the Vivaldi algorithm (Section 2.2.3) to calculate its coordinates and is very simple to use and as well as extend.

Due to the centralized and decentralized systems being developed in the same package (as well as potentially other versions), the node manager must be extensible. Different concrete sub-
Figure 4.1: The abstraction between the different types of managers at the highest level in the system.
classes of the SystemManager will in-turn instantiate their respective concrete node managers. However, much of the functionality of these node managers is shared. The NodeManager should be the abstract class which contains such shared functionality. A brief definition of these shared tasks are as follows:

- Maintain a set of data structures that hold link information.
- Retrieving a particular link based on some unique identifier and test to see if a link exists.
- Finding the number of a links of a particular type of remote host.
- Returning local specific information such as our local coordinate, local host ID, host type etc...
- Link state change procedures.
- Adding RTT samples into the Pyxida Library.

The abstraction of the NodeManager can be seen in Figure 4.2, where the concrete subclasses that extend it are; the DistributedNodeManager for the decentralized version and the MinionNodeManager and OracleNodeManager for the centralized version.

Figure 4.2: The abstraction between the different types of node managers in the system.

### 4.3.1 Links

A Link is an abstract view of a communication link to a remote host. The position of a node in the virtual space is found by repeatedly calculating application-level RTTs to all the links it is connected with. These RTT samples are obtained by having an open Link to a remote host. The node managers should hold these links in three main data structures; **upLinks**, **pendingLinks** and **extraLinks**. These data structures in effect define the state of a Link.

Intuitively a Link can be set up in two ways. Firstly, a node may initiate contact with another node due to the possession of its host name. This type of link is called a Sender. Conversely, a
link may be initiated by another node making contact, this type of link would be generated by some kind of listening server (Receiver) and is called a ReceiveWorker. The two types of links are almost identical apart from their initiation, meaning that most of the functionality will be held in an abstract class called Link. This abstraction can be seen in Figure 4.3. Whenever a node manager is created, a Receiver should be created. The Receiver listens for any incoming connections to the node manager, spawning off ReceiveWorkers for each one.

Figure 4.3: The different types of links that need implementing.

Each link to a remote host should have one connection and only one thread to handle this connection. Although unorthodox, this will cut down the number of threads within the application. As the communication between links will be little (due to periodic measurements only happening at least every 4 minutes), it would be a waste to have threads waiting for messages from hosts. This will cut down the number of threads for hosts by half.

However, some protocol must then exist in order to know which host should be receiving and which sending. Such a protocol must be safe (only one host communicates at one time) and have the liveness property (if a host wishes to communicate, it eventually will). The proposed design to achieve this is to have a communicative locking mechanism on the channel which guarantees the properties required.

The Link must also do repeated RTT measurements with the remote host linked with and the samples fed into a method defined by the high-level manager. As well as executing the RTT measurement action, the Link will also have to execute a number of different actions given to it. These actions should all implement a clean cut interface called LinkAction (Figure 4.4) which forces any such action to define an execute method for all Links to understand and execute. It also enables other LinkActions to be defined easily for other purposes such as testing.

Figure 4.4: The LinkAction interface.

Although Links can understand actions, they do not have a high-level communication medium by which to send messages. This can be achieved with Java easily, which has object serialization. Object serialization is used to send Java objects across data links and works by flattening objects before sending and expanding them back to their original form when reaching the other
side. Of course both sides must have the object implementation and the objects and its content must be *serializable*.

![Diagram](image)

**Figure 4.5**: The *RequestMessage* abstract class and the *ReplyMessage* concrete class.

An appropriate design for link messages would be to have one abstract class (*RequestMessage*) which defines an abstract method *handle*. Having such a method simplifies the process for the recipient as running *handle* is all that must be done. If the requesting message wished for data to be sent back to the sender, then a *ReplyMessage* is created on the recipients side. *ReplyMessages* can hold any object type that is *serializable*.

You might think this would be a problem as the object received in such a message needs to be type-casted to be useful. However, as it is a *LinkAction* calling the shots, it will know what messages (and its contents) to expect, eradicating object type-casting problems. Both classes can be seen in Figure 4.5.

### 4.3.2 Oracle Node Manager

The oracle node manager (*OracleNodeManager*) is the centralized component in the Oracle system. Its main job is to keep links with all nodes that enter into the system.

Initially, when a node enters the system, it will contact the oracle node manager and asks for a random set of other nodes within the system. This will stabilize the newly added nodes coordinates. Once stabilized, if the node is a client, it will repeatedly ask the oracle node for the closest cache to it.

The two main functions for a oracle is therefore listed as:

- Find the closest node based on a given coordinate and all links.
- Retrieve a random set of links that are up and running.

### 4.3.3 Minion Node Manager

The minion node manager is part of the oracle system. The minion node manager has little use for its thread time, and would be better if not used at all. However, the *MinionNodeManager* does have to set up an explicit link with the oracle system, which is never to be broken. If the link does break for some reason, the *MinionNodeManager* should try to initiate contact with the Oracle repeatedly.

This node manager can be used by any system working with the oracle (client or cache). This saves code duplication, as the node managers are identical.
4.3.4 Distributed Node Manager

The additional functionality held by the DistributedNodeManager should be the implementation and access methods for a routing table (RT). The node manager should have clearly defined access methods to the routing table, allowing:

- Insertion of a host into the routing table.
- Retrieval of a host based on a given coordinate and the routing table.
- Removal of a host from the routing table.
- Retrieval of the nearest neighbour of a particular type as well as a set of nearest neighbours.
- The continuous updating of the routing table.

Clearly, such methods defined are closely linked with the actual routing scheme chosen. The routing scheme we will be using in this application is one already discussed (Section 2.2.5). The most basic scheme (θ-routing) is to have a single ring divided into sectors. However, the benefits of extending this scheme to handle multiple exponentially increasing sized rings (θ-routing) clearly outweigh the drawback of having more links to handle. The proposed design for such a structure is a 2-dimensional array, with sizes dependant on the number of sectors and the number of rings. Each array position is a single zone in the routing table.

The process of adding a link into the routing table can now be defined as a mapping from; the coordinate of the host being added (as well as our local coordinate) to two array indexes. This mapping can be done with simple geometric calculations. These indexes can then be used to place the link into the routing table. Equally when we need to route a message towards a particular coordinate, we will run the same methods to find which host in the routing table to pass the message onto.

The act of routing and initiating a routing procedure will be handled by the execution of a LinkAction, which results in the sending of a RequestMessage.

4.4 Proxy Servers

The system being built will require at least two types of proxy servers. One for the client; forwarding requests to an appropriate cache and one for the cache; handling requests in some way. Both versions will however require a server which listens for requests and spawns off the appropriate handlers, depending on the scheme. This gives a great opportunity to use the abstract factory design pattern.

Figure 4.6 shows part of this pattern. The defined concrete classes; CacheHandlerFactory and ForwardHandlerFactory should both spawn off their respective handlers, namely; CacheProxyHandler and ForwardProxyHandler. Both of these handlers extend the abstract class ProxyHandler. This abstraction can be seen in Figure 4.7. For a better view of the abstract
Figure 4.6: The ProxyServer concrete class holding an abstract ProxyHandlerFactory which has concrete sub-classes CacheHandlerFactory and ForwardHandlerFactory.

factory design pattern, have a look at Figure 4.8, where a simplified UML diagram has been drawn to show the interaction between the components of the system.

Figure 4.7: The ProxyHandler abstract class (implementing Runnable) with two concrete sub-classes CacheProxyHandler and ForwardProxyHandler.

The abstract class ProxyHandlerFactory should have an abstract method called makeHandler which returns a handler of abstract type, ProxyHandler. This simplifies the proxy server’s job by running makeHandler on each request, generating the appropriate handler.

The abstract handler class ProxyHandler will also have an abstract method called run. When implemented by a specialized handler, the run method will actually be overriding the method from the Runnable class. This is done because each request needs to be a separate thread, allowing concurrent requests to be handled. The run method will be initiated/started by the proxy server, once the object is created by the factory. This then handles the request in whichever method the handler has been programmed to do so.

The proxy server itself must also be a thread of its own, as it must always be listening for incoming requests. However, in a service such as a proxy server, requests can build up very quickly potentially slowing down the server to a halt. This is why it is wiser to implement a ThreadPool which holds a set of threads, making sure the number of them does not exceed a
limit.

There are many variations that could be implemented on top of the basic CacheProxyHandler technique. Clearly one is necessary as the basic technique offers no collaboration between caches, most probably resulting in a very high false miss rate. These variations and their respective advantages and disadvantages are discussed in Section 5 further.

### 4.5 Cache Manager

As briefly mention in section 4.2, the caching functionality will be placed in the abstract CacheManager class. Any high-level manager wishing to have such functionality must extend it. The cache replacement strategy we will be implementing will be the LRU (Least Recently Used) strategy which is very simple and performs well in web caching (Section 2.3.2).

Before discussing the actions required by the cache manager, it is important to state that each piece of content stored should be encapsulated as an object (CacheContent). This object should have simple access methods for retrieving the relevant data from disk as well as store it. It should also contain general information about the content such as length in bytes and the content encoding type.

The CacheManager is where all these CacheContent objects will be stored. The functions required by the manager are as follows:

- Insertion of content objects into storage.
- Retrieval of content objects from storage.
- Explicit method for resetting of the content object’s position in the time line (to obey the LRU strategy).

Although basic, its important to get it correct as it is the foundations of the caching systems.

### 4.6 Component Interaction

The final design of the interaction amongst the abstract components can be seen in Figure 4.8. Here we can clearly see how multiple links belong to a node manager and how the design factory is used to create handlers for the proxy server. Figure 4.9 shows a concrete version of this interaction for the distributed web caching system.
Figure 4.8: A simplified UML diagram showing the abstract interaction between the different components in the system.
Figure 4.9: A simplified UML diagram showing the concrete interaction between components of the distributed cache manager. Note that the basic \textit{CacheProxyHandler} class was used in this diagram. It could just have well been the \textit{NeighbourHelpCacheProxyHandler} (the Cache-Centric technique) and its respective factory.
Chapter 5

Content Location & Replication

Overlay networks such as ones that have DHTs, offer an inbuilt mechanism to store and retrieve content as they are based on a key/value mapping. It allows nodes to know on what particular web cache within the network a piece of information is stored.

With network coordinates there is no such notion of this. Storing and retrieving is done on a single web cache and collaboration amongst nodes is only done to help place another node into the network coordinate space.

However network coordinates have a number of advantages that can be used in building better content distribution and retrieval mechanisms. For example, if a web cache within the network were to have a local cache miss, then it could ask its immediate neighbours. This is a variation of a simple flooding technique used in lots of distributed applications, but different in that the query is only sent to caches within the vicinity and not propagated further. Another approach would be to have some sort of random walk technique through the network, asking caches on the way. If a cache has what we are looking for then the content is sent back, otherwise the message is passed on in a random direction. However this technique will probably prove to be highly inefficient as it would disturb and potentially overload countless nodes throughout the network as well as adding delays in retrieving content for the client.

The simplest and poorest of approaches is to have nodes only look for content locally and therefore not assisting each other in the sharing of content at all. This approach was initially implemented (CacheProxyHandler) to test communication code first.

Later on during development, it became apparent that the ordinary CacheProxyHandler was a bad technique. As we have so much locality information provided by the node manager it seemed a waste to not develop something that actually collaborated with other cache nodes.

I came up with two design options to extend the existing technique and evaluated their pros and cons. I call these techniques; Cache-Centric Neighbours and Client-Centric Neighbours handling.

5.1 Cache-Centric Technique

In most traditional distributed web caches such as hierarchical ones (Section 2.1.1), requests are first tested locally and passed on if a local cache miss occurs. The passing on of
the request gives another opportunity for the content to be returned by a cache node. This will inherently increase the system's global hit count as less false misses will occur. However, a request passed on will contain a latency penalty for both: passing on the request to different caches and returning the content through the previous caches.

With the use of network coordinates and the forwarding of a request to neighbouring cache nodes, these latencies would be minimal. This is because the metric used for proximity is in fact latency. Network coordinates also give us the opportunity to pass on the request to all surrounding cache nodes in parallel. Secondary cache nodes (caches to see a client's request second) could then also do the same parallel request and so on... flooding the entire overlay network with requests. This would be highly inefficient and add too much latency to a request. This is because, each level of cache nodes reached adds an extra cache to the path on which the content must travel to return to the user. Instead a more sensible scheme would be to do a single-level flood, where only a cache node’s direct neighbours (secondary cache nodes) get asked for the content.

The requests sent from a client node to a cache node and ones from a cache node to another cache node must be identified differently. This is because the cache needs to know when it is allowed to forward the request to its neighbours and when it must reply to the requester no matter what the outcome of the local cache test.

Figure 5.1: Shows a coordinate configuration in which the Cache-Centric Neighbours technique is used. The situation is when there is a global cache miss.

Figure 5.1 shows how a client's closest cache neighbour is picked to handle the requests. For each request, the selected cache (primary cache) checks for the content locally and; assuming a cache miss and knowing that the request was sent from a client, asks its direct cache neighbours (secondary caches) within a given distance from itself.

The secondary caches that receive such a request will know that the request was forwarded by a cache and therefore will only return the content if it is cached locally. Otherwise an appropriate response is returned letting the primary cache know of the absence of the content. Figure 5.2 shows the decision process adopted by the closest cache node (primary cache) chosen by the client node when sent a request.
Figure 5.2: Shows the decision process that is gone through by the primary chosen cache node when sent a request from the client using the Cache-Centric technique.

The advantage to such a system is that multiple caches can be checked very quickly and in parallel. Clearly when doing such parallel checks, a limit for the amount of waiting time must be set. The first cache that returns the content is chosen as the selected secondary cache. When the selected secondary cache returns the content it must however go through the primary cache before it gets sent to the client. This gives the primary cache the opportunity to copy the contents into its own local cache store. This is a side effect of such a scheme and is a huge advantage as content replication is important. Replication of content from cache to cache is important as it gives users more points of access to popular data. Once locally cached, the primary cache can then deliver the content to another user on the next request without having to ask its neighbours, improving response rates.

Our hope is that if our application-level RTT measurements succeed in encapsulating load into the coordinates, an overloaded cache will move away from the congested zone. The load on the cache may be due to a number of reasons, however if it is due to servicing hot/popular data, content replication will have a huge effect on the performance of the system. This is because once the overloaded cache has moved away, another cache will inevitably take its place. When a
client then asks this new cache for the popular data, it will forward the request to its neighbours. The once overloaded cache then has the opportunity to replicate the popular data onto the new cache. The roles would then most likely switch as the new cache overloads and the old cache recovers. This would mean that caches would in effect work a round-robin style load-balancing technique which is very desirable.

There are however disadvantages to the cache-centric scheme, but should be considered minor. Firstly, there is potentially a small latency penalty for asking neighbouring caches for the content. This however is counter balanced by the fact that you are increasing your chances of retrieving the content much quicker than if you were to get it from the origin. Also, the latency penalty is minimal as the caches are nearby in the latency coordinate space.

Secondly, although it may be considered as an advantage due to content duplication, the content must go through another cache node. This can potentially slow down the rate of which the content is returned to the user. However, this rate will most definitely be quicker than what you would get if retrieving from the origin, subsequently making the technique worth wild to use.

Thirdly and lastly, a disadvantage shared by both techniques is the overhead of sending multiple requests to different cache nodes. In order to make it parallel, each request will have to be running on a separate thread which will use up more of the systems potentially scarce resources. However, these requests should be incredibly short-lived as they exist as long as it takes to send an HTTP request to a node very nearby and retrieve its response.

5.2 Client-Centric Technique

The problem with the Cache-Centric technique is that the secondary caches being asked are not the clients neighbours but instead they are the primary caches neighbours. Although some may be in both’s neighbours list, many will most probably not be. This inspired me to evaluate another technique, one I call the Client-Centric Neighbours technique.

The process is basically the same. Single-level flooding occurs to check caches for content locally and content is sent back if found. However, instead of a chosen cache (primary cache) doing the parallel checking, the client does it themselves. Two types of requests are made:

- **Local-Check-Only**, where the receiving cache only returns content if it is locally cached.
- **General-Request**, where the content is returned either from the local cache or origin web server.

The client then spawns off *Local-Check-Only* requests to all its cache neighbours within a distance limit, apart from its single closest cache neighbour which gets a General-Request (all requests sent in parallel). The result is that all the actual neighbours of the client are checked for the content and only one gets the content if not found. So the client is guaranteed to receive the content and potentially very quickly. Figure 5.3 shows how the client is now used to decide which caches to ask.

The advantage to this scheme is that clients get to ask caches that are truly closest. This will reduce the latency if a cache node does in fact have the content locally cached. However, there are a number of disadvantages.

Firstly, if the content is found by a neighbour then the closest cache might still go get the
Figure 5.3: Shows a coordinate configuration in which the Client-Centric Neighbours technique is used. The example shows when all caches within the distance limit have had a local cache miss and therefore the closest cache having to get the content from the origin web server.

content from the origin web server. This could be fixed by sending all neighbours a *Local-Check-Only* request first and then sending the *General-Request* only if no one has the content. The problem with this is that there are two round-trips occurring, which could significantly add to the total end-to-end request time.

Secondly, unlike the Cache-Centric technique, no content duplication is being done between cache nodes and therefore subsequent requests made for the same content by other clients nearby will again result in the retrieval of the content from the origin web server. This simply is not a good enough technique as one of the main aims of the project is to alleviate the origin web server from subsequent requests being made.

Thirdly and lastly, as mentioned in the cache-centric technique, the spawning of multiple threads inevitably uses up more valuable memory.

It is for these reasons that I have chosen to design and implement the *Cache-Centric* technique as it offers many more advantages than just content location. The main advantage being its ability to replicate data when needed, essential to a successful distributed web cache.
Chapter 6

Implementation

The implementation of the distributed web cache followed on quite nicely from the design. As much of the abstraction was thought about early on in the project, fitting the pieces together was a simple task. What remained however was the coding of the actual functionality. Although relatively straight forward, there were a number of concurrency issues due to the large amount of threads running.

In order to test the Pyxida Library and to see how network coordinates actually behaved, I decided to start the implementation with the node manager and link classes. This included all of the low-level communication code required to keep a link up as well as send high-level messages across a link.

6.1 Abstract Node Manager

The node manager is a what holds and offers access methods to all the links within the system. Its three main data structures (upHosts, pendingHosts and extraHosts) hold the links within the system and tell us what state these links are in.

- **Pending** - Trying to connect.
- **Up** - Link is up and running.
- **Extra** - Link is up, we are not interested in having up but remote host is.

The Links themselves can update their state and therefore move them from one set into another. This is a problem because a large amount of links and the node manager, will be accessing these sets concurrently. It is for this reason that I had to synchronize access to all three lists and provide clearly defined access methods, ensuring no concurrency issues.

The synchronized lock was implemented by defining a protected final object called lock. Whenever any access is needed to any of the lists, it must be synchronized with this lock.

The synchronized access methods do as they are named. For example, the state change method, movePendingToUp requires the block of code to be synchronized with the lock object and the Link object moved from one set to another, changing its state from pending to up.
A difficult obstacle that I had to overcome was one that becomes apparent when two hosts try to get in contact with each other at the same time. For obvious reasons we do not want to have two links to the same host. This is taken care with the synchronized methods that change a link’s state. Before changing a link’s state and moving into the upHosts set, the host identifier (hostname with port) is checked against all other hosts in the set. If the link already exists in the upHosts set then the other link is closed down.

The node manager also defines the node types (NodeType) which depict what type of node we are. The final set of defined choices are CACHE, CLIENT, ORACLE and CLIENTREQUESTER. The node type value is needed so that two nodes connecting can tell what type each other is.

The methods for interaction between the instantiated NCClient Pyxida Library object and the sets of Links is defined in the node manager. When a node manager is instantiated, a NCClient object is instantiated and kept as a private global variable (called ncc). The node manager then has a synchronized access method called updateCoordinate which takes in the required parameters needed to add a RTT sample into the library. Each Link that has performed a RTT measurement calls this method to place its measurement into the ncc object. The ncc object then separates the samples into sets defined by a remote host identifier, also passed in as a parameter. The ncc object can then be used to retrieve the current local coordinate which is returned as a Coordinate object.

The Coordinate object is defined by the Pyxida Library. It sole function is to hold coordinates and offer a load of useful methods. For example, a distance calculation method is provided that calculates the distance between two Coordinate objects.

6.1.1 Receiver

Whenever a node manager is instantiated, a single thread is set-up to run the Receiver. The Receiver takes care of any incoming requests, turning socket connections into Links. This is achieved by the creation of a ReceiveWorker object.

When a ReceiveWorker object has finished setting up the socket, the first action that is executed is one initiated to identify the remote host (GetIDAction). The GetIDAction object is executed by the ReceiverWorker which sends the remote host a IDMsg. When the IDMsg is handled by the remote host, its local details such as NodeType are sent back to the requesting host. The ReceiverWorker then continues by explicitly handling a message received by the remote host (which will be another IDMsg but this time from the remote host to the local host).

Once both hosts have finished identifying themselves, they run a method defined in the Link class, established.

6.1.2 Sender

Equally, whenever the local host wants to communicate with a host that we do not have a link with, it can instantiate a Sender object. The sender constructor takes in a host name and a port to which to connect to.

When a Sender is run, a socket is set up with the remote host. As the receiver will execute a GetIDAction first, we must handle any messages received. Once handled, we then execute the
action ourselves to gain the necessary information about the remote host.

Like the ReceiveWorker, the Sender also execute the established method in the Link class after learning about the remote host.

6.1.3 Links

The core of the communication link is stored and managed through the Link class. Once a socket is set up, both the ReceiveWorker and Sender will run the established method which officially changes the Link’s state from pending to up by moving its placement within the data structures defined in the node manager.

The communication link then goes into an infinite loop. In this loop, we are continuously checking to see if its time to calculate a RTT measurement or if the link’s action queue contains any actions to execute. The link’s action queue is a synchronized list that holds objects of type LinkAction. Whenever a higher-level manager or node manager wishes to execute an action on a particular link, a concrete class of type LinkAction is instantiated and placed in the link’s action queue. The queue then repeatedly gets checked by the Link for actions and executes such actions if it has managed to acquire a lock on the channel. This is quite a neat way of defining a communication link as high-level actions can then easily be defined and executed.

Due to the decision to only have one socket open to a remote host and only having a single thread to take care of this socket, some sort of protocol for locking the channel is necessary. This ensures that communication is only made in one direction at one time. For example, take two hosts; A and B with an established link. Some times host A will be listening and handling requests whilst host B is generating messages and sometimes B will be listening and A will be generating the messages. As mentioned in the design section (Section 4.3.1) the communication protocol implemented must be;

- **Safe** - only one host should be communicating at one time.
- **Live** - if a host has the desire to communicate then it will eventually communicate.
- **Deadlock free** - follows on from the liveness, making sure that both hosts do not stop communicating waiting on each other.

This was a relatively difficult obstacle to overcome as it is quite hard to get right. The way it was done was to implement a channel lock. Every execution of the loop makes the host wait for some time period for a lock request. If the other host decides it needs to communicate then it will perform a lock acquire action. If after the lock acquire action, confirmation is sent from the other host, then the channel lock is given to the lock requester. However, if both hosts perform the lock acquire action simultaneously then a localTurn parameter depicts which host gets the channel lock.

How a host attempts to acquire a lock can be seen in Program 1, where an integer with value 1 is sent to the remote host. The local host then waits indefinitely until the remote host either responds with a request of his own (responding with 1), resulting in a lock conflict and false being returned or responding with a confirmation (responding with 2), resulting in true being returned and the lock acquired. What a host does when waiting for a request can be seen in Program 2. The local host is made to wait for a certain time period and then reads any objects.
Program 1 The `acquireChannelLock` method used to acquire a channel lock.

```java
private boolean acquireChannelLock() {
    out.writeObject(new Integer(1));
    int result = (Integer)in.readObject();
    return (result == 2);
}
```

Program 2 The `channelLockRequest` method used to give the remote host a chance to acquire the channel lock.

```java
// The TIMEOUT of the socket is set to the amount
// of time to wait for a lock request from the
// remote host.
// ** getSocket().setSoTimeout(TIME TO WAIT) **

private void channelLockRequest() throws IOException {
    int result = (Integer)in.readObject();
    if (result == 1) {
        out.writeObject(new Integer(2));
    }
}
```

in the input stream. If a 1 is waiting for it or is received whilst waiting then a 2 is sent back and the method exits successfully. If however, nothing is received, the timer will eventually finish and a `SocketTimeoutException` is thrown.

The `localTurn` parameter of one host is always the inverse of the remote host’s. This is achieved by giving the initial `Sender` first priority (`localTurn = true`) and the `ReceiveWorker` second (`localTurn = false`). Each parameter then gets inverted locally every time a conflict occurs, keeping them inversely synchronized. Originally I had implemented for both hosts to back off for a random period of time whenever a lock conflict occurred. However, this wasted too much valuable communication time and was therefore dropped.

6.1.4 Link Actions

The design section showed how `LinkActions` are defined, enabling `Links` to execute them simply and elegantly. `LinkActions` are only created by high-level managers and on occasion, by links themselves (such as when performing `MeasurementActions`). A number of actions need to be implemented for both systems (oracle and distributed).

The action which are defined and used by both systems are as follows:

- **GetIDAction** - Used in the initial stages of setting up a link to get the remote host’s information. The action sends out a `IDMsg`, which when handled by the remote host, sends back its information.

- **SendMsgAction** - A simple action used to send a single `RequestMessage` to the remote host.
• **MeasurementAction** - This is used to perform an application-level RTT measurement. When it is time for a measurement to be performed, the Link executes this action. The action involves a number of steps, covered in Section 6.1.4.

### Measurement Action

One of the most important pieces of code in this project lies within the `MeasurementAction` and `RTTMsg` classes. This is where the actual application-level RTT measurement is done, which attempts to embed a load metric into our network coordinates. When a channel lock is acquired, the Link executes all the requests in the action queue and then performs a measurement action if enough time has elapsed since the last measurement.

Figure 6.1: Shows host A and B performing a 2-way application-level round-trip time measurement, generating one RTT sample for each host.

Each link periodically executes the `MeasurementAction` which entails a number of steps. The aim of the action is to perform a 2-way application-level round-trip time measurement. A 2-way measurement enables both hosts participating in the measurement to get a RTT sample. The concept behind the 2-way measurement can be seen in Figure 6.1 where the data is sent across the link 3 times, generating 2 RTT samples.

Once all data structures are initialized and host A knows that host B is ready and listening, host A starts its local timer and sends a certain amount of precomputed random data. The precomputed data is actually generated by the node manager at start up for all Links to use. The amount of data to be sent is also defined in the node manager and is easily changed with the alteration of a public static field. Once host B has received all of the data it also starts its local timer and sends the data back to host A. When host A has finished receiving this data, it stops its local timer and then continues by sending the data back to host B again. Finally,
on receipt of this data, host B also stops its local timer. Both local timers are then divided by two by their respective hosts to get the 1-way round-trip time.

However in practice, the measurement action is much more detailed. In Table 6.1.4 you can see the process being done with a little more detail. To give you an example of the fickleness of the measurements, initializing data structures during the measurement gave wild results. Therefore, initialization needed to be done before the measurement process. This was ensured with the remote host sending a single ReplyMessage once ready. The ReplyMessage synchronizes the hosts as once sent, the remote host is immediately ready to read incoming data without having to initialize anything.

The measurement process also requires the exchange of each others local coordinate and coordinate’s error. This is needed to add a RTT sample into the NCClient object (defined by the Pyxida Library).

Another important point to note is how Host B in Table 6.1.4 ends the process by resetting its timer for the next measurement. It is reset to a time slightly earlier than Host A’s. This means that measurements will be done in reverse fashion the following time round, and reversed again the following... This ensures that both sides of a link initiate the RTT measurement equal number of times, making the process more symmetrical.

Note that the timers are actually implemented with calls to System.currentTimeMillis, which is turned out to be much more accurate than if done with Java Timers, due to the extra level of indirection.

6.2 Oracle & Minion Node Manager

The oracle system is the simplest of schemes. As discussed in section 4.1.1, the oracle node manager keeps track of all the nodes within the system. The node manager then gets asked periodically for the nearest neighbour to a particular node. The node manager must then look for this neighbour and return it.

The OracleNodeManager starts by setting up its Receiver and waits for others MinionNodeManagers to connect to it. The additionally defined methods for the oracle node manager are defined below.

- **findClosestHost** - looks through all links, finding the closest host to a particular coordinate. The distance is measured by a method defined in the Coordinate class. This will be called by the Clients running the MinionNodeManagers.

- **getRandomSet** - returns a random set of $\log_2(n)$ hosts. This is called by both the Caches and Clients running the MinionNodeManager at the very start of the connection with the oracle.

Optimizations can be easily made to the scheme, such as updating a list of nearest neighbours for each node, so repetitive searching is not needed each time queried. Another technique would be to have a data structure similar to the routing table in the distributed system. We can then discard half of the links automatically as direction is embedded in the table.
<table>
<thead>
<tr>
<th>Host A</th>
<th>Host B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Initialize data structures and local variables</td>
<td>1. -</td>
</tr>
<tr>
<td>1.2 Send <code>CoordinateMsg</code></td>
<td>1.2 Send local coordinate</td>
</tr>
<tr>
<td>1.3 Send <code>ErrorLevelMsg</code></td>
<td>1.3 Send local coordinate’s error</td>
</tr>
</tbody>
</table>
| 1.4 Send `RTTMsg`                           | 1.4 Initializing data structures and local variables  
|                                             | 1.4 Send `ReplyMessage`                     |
| 1.5 Start Local Timer                       | 1.5 Start Local Timer                       |
| 1.6 Send random data                        | 1.6 Send random data back                   |
| 1.7 Stop Local Timer                        | 1.7 Stop Local Timer                        |
| 1.8 Send random data back                   | 1.8 -                                       |
| 1.9 -                                       | 1.9 Send `CoordinateMsg`                    |
| 1.10 Send local coordinate                  | 1.10 Send `ErrorLevelMsg`                   |
| 1.11 Send local coordinate’s error          | 1.11 -                                      |
| 2 Calculate 1-way RTT                       | 2 Calculate 1-way RTT                       |
| 2.1 Add sample to `NCClient` object         | 2.1 Add sample to `NCClient` object         |
| 2.1.1 Update timing for future measurement  | 2.1.1 Update timing for future measurement  |
|                                             | 2.1.1 (slightly earlier than remote host)   |

Table 6.1: A detailed view of the 2-way RTT process explained with 2 hosts; A and B. Note that time is interpreted as if reading ordinary text; left to right followed by the next line.
The MinionNodeManager does very little with its thread time other than set up new links. However it has the oracle’s link defined explicitly as a global variable. It also has a method defined to place an action in the oracle link’s action queue. This was done so that the high-level manager can communicate directly with the oracle and not have to search for the Link object each time.

6.2.1 LinkActions

The actions that are relevant to a link in the oracle system were designed to be executed by the oracles minions. The actions are:

- **RandomSetAction**
- **NearestNeighbourAction**

For a Cache, the first (and only) action done (once the oracle’s link is established) is to send a RandomSetAction. This is achieved with the MinionCacheManager (High-level manager) by putting the action in the oracle link’s action queue through the method defined in the MinionNodeManager.

The RandomSetAction sends a RandomSetMsg to the remote host. As the RandomSetMsg only applies to an OracleNodeManager, the manager handling the message remotely is checked. Following this check the remote host will then execute the getRandomSet method, which return a random set of hosts to the local host. Once received locally, the RandomSetAction follows on by setting up links with all of the hosts returned by the oracle.

This procedure is done in order to ensure that each host is reasonably connected to the network. This makes for more stable and accurate coordinates. The cache running the MinionNodeManager then does nothing apart from its periodic measurements to each host.

A Client is almost identical to a Cache. The initial set up also retrieves a random set of nodes. This again is to ensure the Clients coordinate is stable and accurate. However, the action is placed in the oracle link’s action queue by the MinionClientManager (High-level manager). This manager is slightly different to that of the caches as it also periodically places a NearestNeighbourAction in the oracle link’s action queue.

The NearestNeighbourAction is responsible for sending a NearestNeighbourMsg with the local coordinate as a parameter. On receipt of the message, the oracle executes its findClosestHost method. The result is returned back to the local host (Client) which updates a global variable (nearest cache node) in the high-level manager (MinionClientManager) for all components to use.

6.3 Distributed Node Manager

The distributed node manager (DistributedNodeManager) has a much more complex method for learning about neighbours. It uses a routing scheme which has been discussed in depth in sections 2.2.5 and 4.3.4. The routing table itself is is defined as a 2-dimensional array, with its two indexed representing the ring and sector a host is in.
The distributed node manager defines the retrieval of these two indexes as two separate private functions; \texttt{whatRing} and \texttt{whatSector}.

When deciding what \texttt{ring} a host should go in, the distance to that host is calculated. This distance is the \textit{complete distance}; the basic trigonometric 2D distance with the addition of both host’s height. Height is beneficial as it gives a node the ability to escape into another dimension, increasing distance drastically. Using the complete distance is important because we now have application-level RTT measurements being performed, embedding load into the distance. The height of a node is just one way load can represent itself.

Once the \textit{complete distance} has been calculated we can then decide which ring it belongs to. Rings are organized in exponentially increasing sizes. Iterating through the rings starting with ring 0, we test if the distance calculated is smaller than the current ring size. When the test is \texttt{true}, the ring number is returned. If no ring is found (host is too far away) \texttt{−1} is returned.

When deciding what \texttt{sector} a host should be placed in, a number of operations need to be done. Intuitively, the internal array size depicts the number of sector. Each array index is a sector within a circular circumference around the local host. Each sector is made up of \(360/\text{numofsectors}\) degrees, and each quarter of the circle is made up of \(\text{numofsectors}/4\) sectors (\texttt{SECTORS_PER_QUARTER}). With this information we can easily place a host in a sector.

We first calculate the amount of displacement on the \textit{x}-axis (\(x\text{Dist} = (\text{localxcoordinate} - \text{remotexcoordinate})\)) and the \textit{y}-axis (\(y\text{Dist}\)). With this information alone we can place the host into a quarter. Looking at Program 3 you can see that we divide up where the host can be in the circle by looking at whether the calculated values are position or negative. If the \(x\text{Dist}\) value is \texttt{positive} (\texttt{negative}), then the host must be in the \texttt{EAST} (\texttt{WEST}) of the circle. Equally if the \(y\text{Dist}\) value is \texttt{positive} (\texttt{negative}), then the host must be in the \texttt{NORTH} (\texttt{SOUTH}) of the circle.

The next step in the program is to initialize the \texttt{sector} variable with the first sector’s index of the quarter (Sector 0 starts NORTH, increasing as you go clockwise). Once done, the final step is to calculate how many more sectors to move along from the beginning of the quarter. This is done with basic trigonometry, by calculating the angle between the quarter mark and the vector which is between the local host and the remote host. With the ordinary distance calculated and the \(x\text{Dist}\) and \(y\text{Dist}\) values we can calculate the angle with the inverse \texttt{sine} function.

Figure 6.2 shows where each of these values are in geometric terms. The calculation for angle \(A\) for example, is \(\text{arcsin}(\text{abs}(x\text{Dist})/\text{td})\). The angle is then finally divided by the total number of degrees in a sector. The quotient of the value is what should be added to the initialized \texttt{sector} value, giving the final index of the sector. The final configuration of the nodes and their respective sectors (assuming 8 sectors) can be seen in Figure 6.3.

The reason for making the process so simple is so that a host’s placement can be calculated very quickly. The same two step process used to place a node within the routing table is also used to remove a host from the routing table or to find the \textit{next hop} when routing a message towards a coordinate.

Routing table membership is also recorded at the Link’s side for quick and easy testing. The
Program 3 Part of the whatSector method, which decides what sector a host should be in based on its coordinate.

```
// Ordinary distance calculated.
double td = Math.sqrt(Math.pow(Math.abs(xDist), 2) + Math.pow(Math.abs(yDist), 2));

// Right side of circle (East)
if (xDist >= 0) {
    // Top side of circle (NorthEast)
    if (yDist >= 0) {
        sector = 0;
        angle = Math.asin(Math.abs(xDist) / td);
        // Bottom side of circle (SouthEast)
    } else {
        sector = SECTORS_PER_QUARTER;
        angle = Math.asin(Math.abs(yDist) / td);
    }
    // Left side of circle (West)
} else {
    // Top side of circle (NorthWest)
    if (yDist >= 0) {
        sector = 3 * SECTORS_PER_QUARTER;
        angle = Math.asin(Math.abs(xDist) / td);
        // Bottom side of circle (SouthWest)
    } else {
        sector = 2 * SECTORS_PER_QUARTER;
        angle = Math.asin(Math.abs(yDist) / td);
    }
}

// Add any additional angles
sector += (int)Math.floor(angle / DEGREES_PER_SECTOR);
return sector;
```
Figure 6.2: Shows the calculations that are done to place a node into a sector in the routing table.

Figure 6.3: Shows the final placement of nodes within the sectors after the calculations. Sectors 0, 3, 5 and 6 respectively.
distributed node manager periodically sorts through the routing table keeping it updated. This is done by looking through all links and checking for three conditions and updating the table with its changes. The three condition being checked are:

- If the host is new to the routing table.
- If the host has moved to another location in the routing table.
- If the host was in the routing table but has now moved too far away or has been replaced.

### 6.3.1 Link Actions

The distributed system has two more actions (along with those used with the abstract node manager) that are related to it. Both are used by all nodes in the distributed environment and provide different functionality. A summary of the two actions is as follows:

- **RouteAction** - Used by the distributed node manager (*DistributedNodeManager*) and given to a random link periodically. The action itself sends out a routing message which then eventually gets routed back to the original host with all the routing tables of the hosts that it went through. This action is discussed in more detail in the following section (Section 6.3.1).

- **LinkInterestAction** - Whenever a link is of no interest to us any more, for example when the host is too far away to be included in our routing table, a LinkInterestAction is executed on that link. This expresses our interest or disinterest in having the link open. This is done with the sending of a *InterestMsg* which sets a boolean value stored in the remote host’s Link object. We may change our mind, telling a remote host we are now interested. If both hosts are not interested in having the link open, the link is safely closed. Both remote hosts must know of each others disinterest before the closure of the link as caches will try to reconnect to other caches if a link is unexpectedly closed. This is not the case with caches and clients as we expect clients to come and go often.

#### Route Action

The route action is a fundamental piece of code to the distributed system. With the routing table and **RouteAction**, nodes can learn about other hosts within the system. This is achieved by the *DistributedNodeManager* periodically placing **RouteAction** objects into randomly chosen Link’s action queue.

When a **RouteAction** is executed by a Link, a **RouteMsg** (with the local host name and the local coordinate of the sending host) is sent out to the remote host. When the remote host handles this message the hosts local host name is compared with the host name in the message.

If the host names are the same then the message has arrived to its destination. No more routing is required and therefore all routing tables added to the message are merged with our own routing table.

If however the host names are different, the message has not reached its final destination. In that case we take the coordinate supplied with the message and find the next host to forward the message to.

The **next hop** function is similar to that used to place a link within the routing table but with a few differences. For example, it is likely that the **whatRing** method does not give us any ring as
the coordinate position we are routing to may be too far away. If this is the case we try to take the host that is the furthest away, minimizing the hop count. Equally, whatSector may result in returning a position in which no host exists. We then need to fan out, checking neighbouring sectors for hosts. If we have fanned out too far an angle with respect to the coordinate and not found a suitable host to route too, a Link is established with the final destination host and the message is routed to it directly. Once routed on, the host can then also merge the routing tables in the message with its own, giving all hosts on the message’s path the ability to learn about other hosts!

Complications do arise however. Each host must check for looping, by checking it has not already seen this routing message before. If it has seen the message before, then a link is created and the message is sent directly to the destination.

Figure 6.4: Shows the next hop procedure done by each host when routing a message to a locator coordinate.

The next hop procedure can be seen in Figure 6.4, where at each step the procedure is executed. The next message is then routed to the next host and the process repeats. The diagram shows
two alternative endings. If the coordinate locator of the original requesting host is correct and tells us where the host actually is, **Step 7a** is performed. If however, the node at **Step 6** finds that no other node is closer to the coordinate in the message, it creates a brand new link with the original (displaced) requester shown in red. The message is then routed and **Step 7b** is performed.

Note that along the path on the figure, the nodes always try to take the furthest host if the locator coordinate is outside of its routing table range. The fanning out process is exaggerated in the diagram for demonstration purposes.

Allowing for the creation of links to route a message back to its origin is a novel technique to overcome the problems of outdated locator coordinates. Such problems arise when network coordinates are unstable and inaccurate (Section 2.2.4) and could otherwise result in the original sender of the routing message never receiving its message back. This way all routing messages get back to their original senders, making sure information is not lost in the system. However, such a scheme does not work if there is no method for closing links down. This was the primary reason for having implemented the `LinkInterestAction`, a method for links to express their (dis)interest in having the link open.

Note that neatly, the routing action can also be performed on close-by neighbours to retrieve their routing tables instantly. This saves having to define new actions and messages for learning about nearby changes within the system.

### 6.4 Proxy Server

The next component of the system implemented is the Proxy server. The proxy server is what accepts HTTP requests and deals with them appropriately. Each system type has a different proxy handler. A proxy handler is a single thread than implements the `run` method with a connected socket in a global variable. It is the handlers job to interpret the incoming request and respond to it with the content.

All implemented handlers are create through their designated factories. Each concrete factory type is instantiated by its appropriate high-level manager and then passed to the proxy server which created handlers whenever any incoming requests are made.

There are three implemented handlers. One of which is used for the client machine (Forward Handler) and the two others for a cache machine (Cache Handler & Neighbour Help Handler).

#### 6.4.1 Forward Handler

The `ForwardHandler` class defines what is done with a HTTP request on a client’s machine. As described early on in the report (Section 3.2), the client’s application creating requests must be pointing to the local proxy server. The local proxy server then picks up these requests and creates a new `ForwardHandler` object for each one, which is then `run`.

The forward handler’s task is to forward the request to a cache, defined within its global parameters. When a forward handler is instantiated, the factory retrieves the nearest cache neighbour information from its high-level manager (which extends the abstract class `ClientManager`), and passes it to the handler’s constructor. This ensures each request is forwarded to the nearest
cache neighbour at the time of receipt.

The forward handler then continues with its task by opening a new connection to the selected nearest cache. With the help of the **HttpComponents** library (Section 2.6.2), defining HTTP client and server connections as well as *requests* and *responses* is very simple. The cache then gets sent the original request with the addition of a new header field called *FromNode*. *FromNode* helps the recipient of the request to know if the request was sent from a cache or a client node. This is important when dealing with the more advanced developed handler (Neighbour Help Handler).

The cache receiving the request then responds to the client with a HTTP response, indicating (in the status line) whether the content is found (response code 200) or if an error has occurred (e.g. response code 500). The forward handler simply forwards this onto the client requester (browser). If the content has been found, the following response will contain the *entity* (content). The content is also then piped to the client requester.

The forward handler has very little interaction with the requests and responses as it simply pipes all data back and forth between the client requester and the chosen nearest cache.

### 6.4.2 Cache Handler

On the cache machine, a different type of handler needs to deal with requests. The **CacheHandler** is an implementation of the most basic cache handler you could possibly have. This was implemented first, to be able to test the foundations of the caching mechanism, the forwarding mechanism and to familiarize myself with the HttpComponents library.

The **CacheHandler** class implements a *check-locally-only* scheme, whereby the cache only checks his local cache for the content. The presence of the content is checked with the high-level manager (one which extends **CacheManager**), which keeps a map of all content cached, referenced by the URL where it was originally hosted.

**If a cache ‘miss’ occurs,** a **CacheContent** object (Section 6.5.1) is created to store the content’s information and a URL connection is established to the origin web server. A response for the client’s request is then created and sent, so that the client can expect the content to follow.

For logging purposes, some headers are added to the response before sending it to the client. These headers identify whether the item was in fact returned from cache or the origin web server as well as the length of the content.

An input stream is retrieved from the URL connection and is passed into a **ResponseContentProducer** object. The **ResponseContentProducer** is an implementation of a **ContentProducer**, defined by the HTTPComponents library. The **ContentProducer** interface contains one method; *writeTo(OutputStream outputStream)*, which writes out the content to the assigned output stream for the recipient to read. The **ResponseContentProducer** is passed into an **EntityTemplate** and is sent to the client. This creates a direct input-output stream link with the URL connection made locally and the client.

However, the customized **ResponseContentProducer** does slightly more than forward the con-
tent from an input stream to an output stream. It also writes to another output stream, so that the content received through the URL connection input stream can be simultaneously written to disk and sent to the client. This is what is called a **write-through cache**. The constructor of a `ResponseContentProducer` takes in the URL connection input stream as well as the `CacheContent` item created in the beginning of the process. This `CacheContent` item supplies an output stream to the file on disk. When the `writeTo` method is called, content is written to both output streams simultaneously so that no time is wasted. If we did not do this, we would have to; first, write the content to disk and then secondly, forward it to the client.

Once the content has been received in its entirety, the `CacheContent` element, along with a key (the content URL) can be given to the high-level manager (extending `CacheManager`) to be stored for easy future retrieval of the content.

**If a cache 'hit' occurs;** a `CacheContent` object is returned and the item is immediately set as being recently used. An HTTP response message is created and sent to the client, so to expect the content to follow. The `CacheContent` object gives a direct new input stream to the file stored on disk. This input stream can be given to an entity object (an object that holds the HTTP response content) which is understood by the HttpComponents library and subsequently, sent to the client.

Again, for logging purposes, the same headers added in the response for a cache 'miss' are added to this response.

The implementation for the handler took much trial and error to understand how the library worked. One difficult section was to implement the **write-through cache** functionality. However, it was worth the struggle as it is essential to performance of the web cache. Once the basic handler had been perfected, a more adventurous and interesting handler was developed (Neighbour Help Handler).

### 6.4.3 Neighbour Help Handler

As discussed in Section 4.4, the Cache-Centric technique is one well worth taking advantage of. The implementation of the technique is defined in the `NeighbourHelpHandler`. This handler is in fact an alternative handler from the basic `CacheHandler` discussed in the previous section.

The implementation for this handler is however quite difficult. Although in design it is quite a simple concept, in practice it is not. For example, a cache needs to know whether it is allowed to forward a request to another cache, or to only look locally. This problem was solved with additional headers in the HTTP requests sent. When a client forwards a request to a cache, it explicitly adds a header telling the cache it is from a client. Similarly, when a cache forwards a request to another cache, a header with different contents is added.

The neighbour help handling technique is not the same for all caches. If the cache has received the request from the client (identified by the `FromNode` header), it is what I call a **primary cache**. If however, the request is received from another cache, it is called a **secondary cache**. Primary and secondary caches deal with requests differently when a local cache 'miss' occurs. If there is a local cache 'hit', the process is the same as that of the basic `CacheHandler` i.e. the

---

1Note that the cache roles will be different for each request.
content stored on disk is returned directly to the client.

Although different in functionality, the implementation for both types of caches (primary and secondary) is defined in the same handler, separated by a conditional statement on the header.

**Primary Cache**

The primary cache is the first cache that has contact with the client’s request. If a local cache ‘miss’ occurs, the cache sends out requests to all its nearest cache neighbours. The technique therefore, needs the handler to know about its nearest cache neighbours and to then spawn off multiple requests in parallel to these neighbours.

The way this was defined was by implementing a *NeighbourRequester* class which extends the *Thread* class. The *NeighbourRequester* is supplied with a reference to its handler, the URL of the content (the cache key) and the host identifier (name and port) which it should be requesting to. The handler first retrieves a list of nearest cache neighbours with a call to its high-level manager, and iterates through this list, initializing and running multiple *NeighbourRequester*.

Each *NeighbourRequester* sets up a socket to the assigned host and sends an HTTP request with the additional header telling the recipient it is from a cache. According to the response, if the remote cache has it locally stored, the *NeighbourRequester* sets the socket in the handler object (to notify the handler of a success). The *NeighbourRequester* thread then closes as the socket has been passed on to be dealt with by the handler. If the cache does not have the content cached, the *NeighbourRequester* thread simply closes.

Although potentially many threads could be running simultaneously, they are extremely short-lived. This is because no matter what the response is, the thread will close very quickly.

Whilst the *NeighbourRequesters* are running, the handler is continuously checking a synchronized global variable for the presence of a *socket* object. However the checking will stop if a certain amount of time has elapsed or if all responses have returned with no luck.

If the timing threshold has been reached without the presence of a *Socket* or if all *NeighbourRequesters* have finished with no luck, the handler retrieves the content from the origin web server, identical to how the basic *CacheHandler* does.

Hopefully, this is not the case and one of the *NeighbourRequesters* sets the handlers global socket variable \(^2\). The handler will then pick up the socket and continue its communication by doing a *write-through cache operation*. It is a write-through operation because the primary cache takes the content received from the secondary cache and simultaneously caches the content and forwards it to the client node.

**Secondary Cache**

A secondary cache has a very simple task. All that is required is for the node to check its local cache store for the content (defined in the HTTP request as the URL). If the cache node

---

\(^2\)Note that only one *NeighbourRequester* will set the socket variable held by the handler. This was done with a single *test and set* method that is synchronized.
has the content then a response with the status code 200 is returned followed by the content itself. If however the content is not cached an HTTP response with the status code 204 is returned, indicating that the content is not cached.

All mechanisms for retrieving content and storing content are the same across all handlers and is defined by CacheManager and CacheContent.

6.5 Cache Manager

The CacheManager is an abstract class which high-level managers extend when they wish to have caching functionality. The CacheManager holds two main data structures both which have synchronized access. These data structures are listed below with a brief explanation to their functionality.

- **Map<String,CacheContent> cache** - This is where the CacheContent objects are stored for access to files stored on disk. They are accessed with a key which is in fact the URL of where the data was retrieved from. This data structure has synchronized access methods to allow handlers to retrieve and insert objects concurrently.

- **List<String> orderedCache** - This data structure is what supplies the LRU replacement strategy. Whenever an object is retrieved from the cache data structure, the objects position in this list is reset to the beginning. This is used for when the total amount of data stored on disk exceeds a certain predefined capacity. Clearly the total amount of data stored needs to be recorded and updated so that to know when such an event occurs.

The storing and retrieving of the content from disk is done with the CacheContent class.

6.5.1 Cache Content

Each object cached is represented with a CacheContent object. When the object is instantiated, a uniquely identified file is created in a cache folder. If for whatever reason the file cannot be created, the instantiation fails and the user is notified. Once set up, the object allows for the files input and output streams to be accessed. However, new streams are created each time which can be problematic if for example, one stream is left open, disallowing the content to be deleted. Therefore each stream given is kept track of and appropriate exception handling is defined.

6.6 High-Level Managers

As described in the design section (Section 4.2), there are a number of different high-level managers. Their responsibility is to provide interaction between the main components running.

To initiate the implemented system on a machine, one must run the main methods defined in these concrete high-level managers, which instantiates a high-level manager.

A summary of the concrete high-level managers that are defined for the oracle system can be seen below.
- **MinionCacheManager** - Runs on the cache machine and provides interaction between the local node manager (*MinionNodeManager*), the proxy server (with an instantiated factory to create **cache handlers**) and the caching functionality.

- **MinionClientManager** - Runs on the client machine and provides interaction between the local node manager (*MinionNodeManager*) and the proxy server (with an instantiated factory to create **forward handlers**).

- **OracleManager** - Runs on the cache machine and simply runs the local node manager (*OracleNodeManager*).

A summary of the concrete high-level managers that are defined for the *distributed system* can be seen below.

- **DistributedCacheManager** - Runs on the cache machine and provides the interaction between the local DistributedNodeManager, the proxy server (with an instantiated factory to create **cache handlers**) and the caching functionality.

- **DistributedClientManager** - Runs on the client machine and provides the interaction between the local DistributedNodeManager and the proxy server (with an instantiated factory to create **forward handlers**).

### 6.7 Challenges

Much of the implementation was straightforward, however like in all substantial software engineering projects, there are always big challenges. Some of the biggest challenges I had encountered when implementing the system are listed (no particular order) in summary format below.

- **Data structure synchronization** - It is easy to make the system synchronized on all the data structures as you could simply use one single lock for everything. However, this is too coarse and could impact performance quite drastically. I therefore had to define a more fine grain locking mechanism with multiple locks. This was hard as I had to make sure no deadlocking occurred when using the multiple locks.

- **Channel locking mechanism** - Hard to get perfectly correct, without any deadlocking.

- **2-way application-level measurement action** - Difficult to get completely symmetrical and to truly represent the elapsed time.

- **Routing Table** - Ensuring hosts are placed correctly into the table was hard. No other way of checking other than doing so manually.

- **Cache-centric technique** - Although the other handlers were relatively simple to implement, the more advanced cache-centric one was not. With the use of yet another set of threads, concurrency problems as well as many others were of a major concern.
Chapter 7

Testing & Visualization

The testing of the system required more implementation to be done. Not only did modules within the system need extending, but whole new modules needed to be built. In addition to testing, a visualization module was also implemented to show node positioning within the virtual space. This was done to better see the effects of the testing environments on the nodes.

7.1 Proxy Handler Tests

In order to isolate communication code with cache code, I decided to extend the CacheProxyHandler class by adding two testing methods; handleDiskTest and handleMemoryTest.

When performing a disk test, a file of a certain size is pre-generated when the CacheManager is being initialized. The file is then used as a response for all requests sent to that cache.

When performing a memory test, a newly defined ContentProducer is used (MemoryTestContentProducer). This new content producer writes out a certain number of zeros on the output stream, to simulate data from memory being accessed and written. The reason for not using random data is to avoid using any computation in the experiment.

The experiment method and results can be seen later on in the evaluation section.

7.2 Resource Eater

One of the first testing modules implemented was the Resource Eater. It is a very crude testing module and its sole purpose is to exert a huge strain on the CPU running it.

The resource eater works by continuously iterating through a while loop for a certain specified period of time. It may then sleep for a while before starting the iterative process again.

Although crude, the resource experiment mimics computational load very well. It can be used to see the effects of load on network coordinates and application-level round-trip time measurements.

Interestingly enough, when initially testing the resource eater on a machine, many artefacts were found in the results. The cause of the artefacts was that the machine running the resource eater and node manager had in fact two CPUs. This allowed for scheduling to be done on the two CPUs with sometimes both components running on the same one (giving the results we
want) and sometimes running on different CPUs. The two processes had to therefore be bound to the same CPU when testing.

7.3 Log Server

A large testing component developed for the system is the Log Server. The log server was built so that other components running on multiple machines could send back crucial data. The log server can then log this data centrally, so that analysis is quicker and less painful.

I was aware before starting development that having such a component could be potentially dangerous. This is because if the number of deployed machines is anything non-trivial, the log server could easily become overloaded, disrupting the logging process. However if the information was kept to a minimum when doing such tests, there should be no problem.

The component works by having an open server-socket to a particular port. *LogUpdaters* then connect to this socket and send *LogUpdateMsgs* to the log server. *LogUpdateMsgs* are defined with a handle method which alter the log server’s data structures accordingly. This may be for example, to update a particular piece of information about the host. There are a number of log messages defined (*LogIDMsg*, *LogCacheChangeMsg*, *LogCoordinateMsg* etc...) all with their own purpose and all informing the log server something about the remote host.

7.3.1 Log Updater

The *LogUpdater* class is instantiated and run by high-level managers. Each system running has a *LogUpdater* and is used to send messages back to the log server. The *LogUpdater* offers synchronous access methods to send *LogUpdateMsgs* instantly as well as to pool messages into a queue, sending them in batches, periodically.

7.4 Client Requester

The client requester simulates multiple clients on one machine. It is highly customizable to fit particular tests and to generate requests to specific URLs. The neat thing about having a client requester is that we can time the end-to-end request time for particular pieces of data as well as log other useful information such as if the request was a cache hit or miss. The client requester also has a *LogUpdater* which communicated directly with the Log Server. Request time information is sent back to the log server for it to be logged onto disk.

The client requester (*ClientRequester*) takes in an XML file which defines the URLs the requester should be generating requests too. The XML file is then parsed to give a set of items. The items in the file are then randomized and an infinite loop is then started. On each iteration a request is made and the data returned from the client proxy server is handled until finished. The client requester’s main method can spawn multiple client requesters so to simulate multiple users running on one client. This then generates requests concurrently to the local proxy server.

The XML file which I have compiled contains a cut down list of web servers spread across 6 continents. The web servers are all hosting a small linux distribution called *Gentoo*. Linux distributions are typical large files a normal day-to-day user would be requesting. As working with full linux distribution of 700 megabytes would be tedious, the small Gentoo distributions were used, which are only approximately 60 megabytes in size. This is a good size for testing
as the saving of the web cache will be seen in larger files rather than smaller ones.

Sometimes, even the 60 megabyte files were too big to use during testing and even smaller header files had to be used. The sizes of such files are approximately 1-3 megabytes and allow for more requests to be handled in a shorter period of time. The smaller files were also used due to PlanetLab monitoring the total amount of data being transferred from the slice and capping node’s burst rates.

The XML file compiled, with the details of each location can be seen in the appendix (Section 9.4).

The client requester was built as a separate module in order to best simulate a true environment. If a user is using the system, they will have some sort of application running, this may be a file downloading program or a browser. The client requester mimics this application by running as a separate program.

### 7.5 Coordinate Space Visualization

An addition to the project was to build a visualization module for the system. When testing to see how node’s coordinates change over time, looking at numbers is tedious. The best and most enjoyable method to see patterns and behaviours is to have the data visualized. The visualization module uses information about each host such as links and coordinates, to place it in a virtual space.

The module is integrated into the Log Server. This is because the Log Server has all the knowledge necessary about each node in the system. A coordinate update for example, would send the Log Server a message notifying of the update. The visualization module uses this data to update the position of a node in the virtual space.

Initially, the module was built by piping data into GNUPlot, an open-source graphing system. This however failed to provide interactive features and although pleasing on the eye, it had insufficient functionality.

The module was instead built with the use of a very neat library called Prefuse (Section 2.6.4). Prefuse is in active development and unfortunately still has many bugs. The worst part of the library is its ability to handle dynamic alterations to nodes and edges. Clearly there are bugs with its synchronization techniques which were not fixable in the time frame at hand. However, I managed to find a few workarounds which made the visualization module still very usable.

The module was designed to show the tester/user a huge amount of information in a readable format. As the Prefuse library only supports 2D coordinates, when placing nodes on the screen, the height vector in our coordinates (in effect another dimension) need to be interpreted in another way. This was shown by altering the intensity of the colour of a node to represent its height. The higher the node is the deeper/darker the colour of the node. Although not the best solution, having a visual graph showing all connections and coordinate positions is still a very useful tool.

The visualization module has a number of features implemented. These include:
- Dynamic movement of nodes and link distances is recorded and shown.
- Highlighting of links and neighbouring hosts when mouse is over a particular host.
- Shape of node depicts the type of host; defaulted to round being a client and square a cache.
- Link distances shown numerically both with and without height when mouse over.
- Hostname and coordinate shown on mouse over.
- Zoom in and out and panning around a screen all implemented.

A snapshot of a virtual space using the visualization module can be seen in Figures 7.1 and 7.2. These images were taken when running an experiment on PlanetLab nodes. Note the shapes and colours of nodes in the zoomed in figure. More images of the visualization module running can be seen in Appendix 9.5.

Figure 7.1: Shows a zoomed out snapshot of a deployment on PlanetLab.
Figure 7.2: Shows a zoomed in snapshot of a deployment on PlanetLab.
Chapter 8

Evaluation & Discussion

The evaluation of the system can be divided up into two sections: **correctness** and **performance**. Some components cannot be easily verified in terms of correctness with formal methods, but have still been evaluated in a more practical manner.

### 8.1 Correctness

There is a distinction between total and partial correctness. **Partial correctness** requires that if the algorithm or system gives us output, it is the desired output. **Total correctness** requires for the algorithm/system to be both partial and to also terminate.

Testing correctness of the entire system is not a feasible option. What is feasible however, is to test the most critical core components of the system. This can be done by individualizing the components and analysing the output.

Nearly all components of the system were tested to some extent. Whether that be by monitoring output, or by building test cases. However, the key components that I have tested for correctness in more depth are:

- Channel locking mechanism
- Application-level round-trip time measuring
- Routing table consistency and Routing algorithm

#### 8.1.1 Channel Locking Mechanism

In order for a single link to be useful to two hosts, a locking mechanism needed to be implemented and tested. Implementing a correct and sound mechanism proved to be quite challenging.

Once a scheme was implemented that seemed to provide consistently correct results, it was written up in *SVM* format (Section 2.5.3). Communication protocols can be represented very easily in SMV. The purpose of this was to be able to safely say that the implementation has the following properties:

- **Safety** - Only one host can be communicating at one time (master), the other host must be handling these messages (slave).
• **Liveness** - If a host wishes to communicate (i.e. become the master) it eventually will.

• **Strongly Fair** - Hosts are automatically placed one after the other if any conflicts occur.

The basic SMV program can be seen in the Appendix 9.2. The MAIN function is where the two hosts are instantiated and where the properties we want to ensure are written. There are four rules written and they work in pairs.

\[ SPEC \ AG \ (\neg (host0\_res \ \& \ host1\_res)); \]
\[ SPEC \ AG \ (\neg (host0\_send \ \& \ host1\_send)); \]

The first pair seen above expresses that; there is no state where both hosts are responding (ensures deadlock freedom) or are sending (ensures safety property).

\[ SPEC \ AG \ (host0\_req \rightarrow AF \ host0\_send); \]
\[ SPEC \ AG \ (host1\_req \rightarrow AF \ host1\_send); \]

The second pair seen above expresses that; whenever a host is in the request state (wishes to communicate), it will eventually be in the send state (communicating). This is the *Liveness* property we wanted to satisfy.

### 8.1.2 Application-level Round-Trip Time Measuring

#### Symmetry in Samples

The communication code of the measurement action can be trusted to be correct by simple inspection whilst running. If any problem occurs such as excess data sent or too little data received, an error will occur on the next message transmitted making it very obvious the code is incorrect. However, if the data is both sent and received correctly then no error will occur, rendering the communication process correct.

There is another heuristic for the correctness of the 2-way measurement action. That is that the timings recorded are in fact the true representation of the elapsed time when sending/receiving data. One way of identifying an error is by ensuring that the timings generated are symmetrical. With our 2-way measuring technique, two RTT measurements are generated with a single action. These two RTT samples should in theory be identical or at least very similar.

Unfortunately, looking at recorded RTTs and ensuring that they are the same for both hosts will not be accurate. This is because our new RTTs are influenced by load and can therefore be different for the two hosts. However, if the computational load on the nodes doing such measurements is negligible and the network load is also low, there is no reason why the RTT measurements should be different. If they are different, then this would clearly show that the implementation of the measurement is incorrect.

An experiment was therefore set up in a closed environment (Imperial College DoC local area network) where network load is irrelevant. Machine in the laboratory were used as to ensure minimal computational load on them whilst performing measurements.

Although quite a dull experiment, initial results showed a skewness in measurements which enabled me to pinpoint a problem. The problem was that there was no synchronization between the sender and receiver on the first measurement. The sender would start sending before the
receiver was reading, slowing down the process. This was not an issue on the second measurement of the 2-way measurement. A synchronization message was placed in the code which is sent from the receiver to the sender to tell the host it is ready to start reading.

Once altered, the experiment was re-run countless times and results showed that there was never any difference between the two generated RTT samples with each action. This shows that the process is now symmetrical and therefore will not produce odd coordinates.

**Load within our Coordinates**

The most important analysis of the system is the impact of application-level round-trip time measurements on a node’s coordinate when the load on the node is altered. In order to test this, an experiment was set up in a closed environment (local area network) with two machines (**Hp-Data** and **Xps-Laptop**). Both machines were running a basic node manager and were set to periodically take measurements at a rate of 1 every 4 seconds. Clearly this is ridiculously high for an actual deployment of the system but it was done for testing purposes so to see a quicker dynamic change in coordinates.

After 2 and a half minutes of waiting for coordinates to stabilize, a resource eater (Section 7.2) was run on one of the machines (**Hp-Data**). This is to hog the CPU time and overload the machine. After 3 minutes of running, the resource eater is shut down and the coordinates are left to stabilize again.

Figure 8.2 shows the results for this experiment, where the latency/distance between the two nodes is plotted against time. The results show how the stability of the coordinates is very good. When the resource eater is run, there is a slight delay until the RTT measurements take effect on their coordinates. The distances between the nodes then suddenly shoot up and remain constant until the resource eater is stopped again. This shows clear evidence for computational load being embedded into the coordinates.

The delay in the rise and fall of the latencies/distances between the hosts exists because of Pyxida’s latency filter. The filter implements a window which gets filled up with samples. The sample used to calculate the coordinate is then some percentile of the samples within the window (smoothed sample). So for high latency samples to take effect, the window must be filled up with these which takes time. Once filled up the smoothed sample will be closer to that of the high samples. This theory is justified by looking at the actual RTT latency samples being generated during the experiment. Figure 8.1 shows us that the RTT latency samples increase drastically as soon as the resource eater is run. Equally, the measurements drop when the resource eater is stopped.

The RTT latency samples (Figure 8.1) generated before and after the resource eater is running, are generally quite stable, but with the occasional odd spiking. It is due to these spikes that Pyxida has implemented the latency filtering system as we would not want the coordinates to be altered whenever a spike occurs. Instead we only wish the coordinate to move when there is a constant change in the latency samples.

Figure 8.1 also shows how the fluctuation in RTT latency samples is much higher when the **Hp-Data** machine is under load. This is because there will be times when the CPU scheduler
Figure 8.1: Shows the relationship between application-level RTT measurement for network coordinates and computational load. Note that the resource eater has its own vertical axis, with the lowest point running at 0% and highest at 100%.
Figure 8.2: Shows the actual application-level RTT measurements of two hosts against computational load. Note that the resource eater has its own vertical axis, with the lowest point running at 0% and highest at 100%.
acts in the measurements benefit and times when it does not, giving the resource eater more resources.

These figures clearly show how load is now a big factor within the positioning of a node in a virtual coordinate space. This is the result we had hoped for and gives us reason to believe that the implemented cache systems will perform well with such a foundation.

8.1.3 Routing table consistency and Routing algorithm

The correctness of the routing algorithm follows from the correctness of the routing tables. If the entries in the routing tables are correctly inserted and updated, then there is little code which could render the routing algorithm incorrect.

Thanks to the visualization panel (node graph) built with Prefuse (Section 7.5), one can inspect the configuration of nodes within the virtual space and compare it with a nodes routing table. We can then determine if its entries are correct. The node graph shows the positioning of each node within the system and if turned on can receive a nodes routing table updates. This allows for us to pause the system and take a snapshot of its configuration. We then can manually compare routing tables with the configuration by hand. The comparison process is identical to the geometric process described in the implementation section (Section 6.3).

Similarly, when analysing the actual routing algorithm, one can see the path a message has taken and compare it with the routing tables generated from the previous process. This process was done multiple times as many bugs were found in both the routing algorithm and the visualization module. Once repeatedly fixed, it soon became clear that the process was partially correct as the actions being performed by the algorithm were as expected.

For the routing algorithm to be totally correct, it must terminate. Due to the problem of the reliability of coordinate locators, this would be hard to show. The coordinate (coordinate locator) embedded into the routing message may be outdated as the node could have moved elsewhere. Will the message therefore end up in the origins hands? The original technique would have problems ensuring this but a quick engineering modification to the algorithm was added to ensure all messages eventually return back to their host. The modification (described in Section 6.3.1) allows hosts to create links to the final destination if it believes it is the closest node to the coordinate locator. The initiation of links guarantees a host will get its routing message back and therefore ensures completeness of the algorithm in the presence of no link or node failures.

More testing was also done on other components of the system. Components such as the proxy caching handlers needed to be tested to ensure content was actually being written to disk and returned. The whole process of forwarding and caching content was also tested with a web browser. When a web browser is linked up to the system, content is cached and returned. Retrieval times makes it obvious when content is cached and debug information also helps to pinpoint any failures with statements.
8.2 Performance & Complexity

A very interesting part of this investigation is its performance evaluation. I have shown that load has been embedded successfully into our coordinates and now I will show how well the system performs when compared to other techniques.

8.2.1 Performance and Scalability

The first test was to analyse how well the distributed web cache can handle an increasing number of requests. An experiment was therefore set up in a closed environment (Imperial College DoC local area network). The distributed cache manager was set up and run on one machine (vertex40.doc.ic.ac.uk) and a single distributed client manager on another (vertex41.doc.ic.ac.uk). Once the two hosts where connected, a new client requester was run on the client machine every 3 minutes. At each 3 minute time point before starting a new requester, all previously running client requesters are stopped. They are then started again with the additional new client requester. This was done so that the previously running requesters did not pollute the following time point results. Every 3 minutes, the requesters start again from scratch logging new request times.

With the help of the Gentoo linux distribution mirror list, the client requesters were requesting for the same exact 7 megabyte file placed on multiple web servers all over the world (Section 7.4).

![Requests Times against Number of Clients Increased over Time](image)

Figure 8.3: Shows the maximum, average and minimum request times against the number of clients running, increased by 1 every 3 minutes.

The results for the experiment can be seen in Figures 8.3 and 8.4.
Figure 8.4: Shows the number of requests handled against the number of clients running, increased by 1 every 3 minutes.

Figure 8.3 shows how the increase of clients on a single web cache increases the average request time linearly. This is what we would have expected as the more concurrent requests being handled by the cache, the slower the request time for each of them will be. This is because the more concurrent requests being made, the less CPU time each one gets. Notice on the graph that the initial point when there is only one single client, the maximum request time is off the scale. The 50126ms point has occurred because the content was being retrieved from the origin web server. Once all items have been cached, the request times drop drastically.

Figure 8.4 shows us the number of requests handled when again, the number of clients is increased every 3 minutes. The graph clearly shows that the maximum total number of requests the cache can handle in a 3 minute span is approximately 2000. This surprisingly does not increase with the number of concurrent clients, as one would expect that the overhead of having multiple threads to decrease the number of requests handled. This shows that the system is very scalable as increasing the number of concurrent requests does not decrease the total number of requests handled. Instead each request made suffers a little more with each client added.

### 8.2.2 PlanetLab Disk-Access

Before deploying the system onto PlanetLab nodes, another small experiment was done. There was a slight worry that deploying the system on PlanetLab nodes would have a huge

---

\(^1\)Note that when this test was done, the thread pool implemented was turned off. This allowed as many handlers as we wanted, to be spawned simultaneously, potentially grinding the cache down to a halt.
performance hit. This was because of the number of concurrent processes running at one time not only consuming a large amount of memory but also performing concurrent disk actions. There is a possibility that a overloaded node performing multiple disk accesses could slow down the retrieval of the content. This would make the disk a potential performance bottleneck in the system. I investigated this theory with a simple test.

I developed two test situations inside the CacheProxyHandler class which was discussed earlier on (Section 7.1). These are the tests I used to investigate this theory.

I deployed the distributed cache manager on a reasonably high loaded PlanetLab node (Host-Name: planetlab-3.amst.nodes.planet-lab.org). I then deployed the distributed client manager and client requester onto another node (HostName: planetlab01.sys.virginia.edu). The load of the machines was recorded at the beginning of each test (using CoTop, a monitoring tool for PlanetLab nodes (Appendix 9.3)).

When testing the disk, the load on the cache machine was 5.07. After 42 requests were made, an average request time of 7295.8ms was calculated. When the test was done with memory access only, the load on the cache machine was approximately the same (4.86). This time, after 45 requests the average request time was 5122.8ms.

Although a little variation, the results showed little evidence for the problem described. It is normal for retrieval of data from disk to be a slightly slower than that of memory. I was expecting a huge difference, one that might cripple the system drastically.

8.2.3 System Comparison

The rest of the evaluation is based on the comparison of four different systems. These four systems can be seen below. Note that CC stands for Cache-Centric technique and LO stands for Local-Only.

- **Distributed Scheme (CC)** - where there is no central process and routing is used to learn about nearby nodes. Caches ask neighbours if the content is not cached locally.

- **Distributed Scheme (LO)** - the difference between this system and the last one is that caches do not ask neighbouring caches for help. Instead, they get the content from the origin if not cached locally.

- **Oracle Scheme (CC)** - where a central process manages links to all other nodes. All system based queries are handled by this centralize machine (oracle). Caches ask neighbours if the content is not cached locally.

- **Random Cache Scheme** - where clients make no decisions based on locality information. Clients were given a predefined list of caches and they choose a cache at random for each request made. Caches also acted on their own with no help from surrounding caches. This is to simulate a purely random strategy.

Each experiment was done three times and was deployed on the exact same nodes and with the same number of caches and clients. The deployment took place on 36 distinct PlanetLab nodes, with a ratio of 1/3 of clients and caches (12 caches and 24 clients). Each client had a client requester working on it and was told to retrieve a 3.4 megabyte (3577919 byte) file from
6 different locations across 6 continents iteratively.

When the content was retrieved from a 8mb commercial home internet connection, it took on average of 10177ms to download at an average rate of 343kB/s

The first step of the experiment was to start up the log server on a remote computer to log all incoming data. The next step was to start up the bootstrap/oracle host. It was left for 20 seconds so to ensure the host was properly set up. Following this, the cache nodes were started. They where left for 20 minutes, so to make sure their coordinates were stable and accurate when testing. The client nodes then followed, which were also left for an additional 20 minutes. The final step of the experiment was to start the client requesters which generated the requests to test the systems. The client requesters were then run continuously for 2 hours and then stopped.

Note that this whole process was scripted to ensure the timing were the same for each experiment, giving no unfair advantages to any system tested.

The results of the experiments can be seen in Table 8.2.3, where they are listed in their respective categories. The best performing system is the distributed one with the cache-centric technique, with the local-only technique and oracle system not far behind and the random cache technique performing extremely poorly. This is clearly due to the locality and most importantly load awareness of the other systems, something the random cache strategy does not have.

The two distributed systems show to have handled the most number of requests. This is because the system has avoided the caches which are overloaded, handling requests quicker and therefore more of them. Not all PlanetLab nodes could handle the requests in a reasonable time, which meant that if used, the client requesters would be stuck with that cache until the request had finished. The random system unfortunately chose many of these caches and therefore suffered with severe penalties. It only made a total of 199 requests which is almost 60 times less than both the distributed versions.

Equally, because the random strategy never got to finish caching all the data items, its number of cache misses is quite low. However its true caching nature can be seen in the cache hit ratio, where is performed much more poorly than any other. Expectedly, its average request time is over the roof. It in fact performed nearly 50 times worse than if the content was retrieved from the origin itself.
Both the distributed and oracle based systems did a good job avoiding these highly loaded nodes as their average request times were considerably lower than that of the random system. They managed to cache the data on all cache nodes being used and therefore return the content at very high speeds. This early caching explains the high number of requests and the high data transfer rates achieved.

However, one must realize that the number of cache misses shown here is not a very reliable measure. This is because requests were being made concurrently and therefore sometimes, two requests may be asking for the same content. This would unfortunately be counted as 2 cache misses. Much of it is therefore down to luck as the requests are generated in a random order. Taking this into account however, the systems using the cache-centric technique show less cache misses than the systems using the local-only technique. This is what we would expect and has helped the two systems (Dist. (CC) and Oracle (CC)) perform well. Note that parameters can also be altered in the cache-centric technique to make sure only caches very close-by are asked, getting rid of the potential danger of asking a poorly performing cache and having to stick it out with the cache.

The two distributed systems however, performed better than the oracle system. This is possibly due to the oracle node becoming overloaded and not being able to respond to closest neighbour queries as quickly as we would have hoped. The slower the oracle is in responding to queries the less the accuracy of the response. Clients then might be forwarding requests to caches that are overloaded or actually very far away in the latency space.

The two distributed systems performed better than when retrieving the content from an 8mb commercial home line (10177ms), with the Oracle system not far behind. If however, we tested the system with a much large file (say 300−400mb), we would most probably see a much higher performance gain with all of these systems. Due to limitations of PlanetLab’s maximum data transmission allowance, this was not possible to do.

There are a few anomalies in the experiment data however. For example, the Distributed Local-Only system, managed to handle many more requests than any other system. This is due to PlanetLab’s dynamism. Nodes alter their performance on an hourly basis, making it hard to test such systems without giving unfair advantages. A better method would be to have tested all systems at once. However, there where issues with memory on my slice which made it impossible for the testing of all four systems to be performed concurrently.

The major disadvantage of the distributed and oracle system is that measurements have to be taken periodically. The total amount of coordinate maintenance data also increases with more nodes in the system. The random cache strategy has no such overheads as no estimation of locality or load is done. However, these measurements add only a small amount of data overhead as they are taken periodically rather than on each request as with Meridian.

The amount of overhead data sent across the all links of the system to maintain the coordinates can easily be estimated. If there are a total of $n$ nodes all with approximately $\log_2(n)$ links, then we have a total of approximately $((n \times \log_2(n))/2)$ links all carrying out RTT measurements (this excludes shared links). Each 2-way measurement sends 3 sets of $d$ kB i.e. $(3 \times d)$ kB. As each measurement is taken every $t$ minutes, there is approximately $((3 \times d \times ((n \times \log_2(n))/2))/t)$ kB of coordinate maintenance data sent every minute throughout the system. The estimated
The total amount of maintenance is therefore of $O(n \times \log_2(n))$. This amount will in fact be close to double due to additional information and routing data being sent.

Take our experiment as an example. There were 36 nodes, all with approximately 5 links each, which could be shared. This gives us approximately 90 links ($(36 \times 5)/2$). Each link is sending 3 sets of 4 kB and therefore sending a total of 12kB every 4 minutes. The estimated total amount of data being transferred for the maintenance of 36 nodes is therefore 270kB per minute. This figure gives you an idea of how small an amount of data is needed to support 36 nodes which can potentially handle a very large number of requests. In fact our results show that the cache-centric distributed system sent on average 28884.6kB every minute (481.41 × 60), which is over a hundred times that of the maintenance data sent. With the random strategy, which has no overheads, only a mere 427.2kB was sent every minute. The benefits of having the overhead therefore clearly outweigh those when not.

The distributed cache-centric system has many more benefits than any of its competing systems.

Firstly, it does not have a single point of failure like the oracle system does. The oracle system relies solely on the oracle. If anything were to happen to the machine running the oracle or its internet connection, the entire system would break down. The oracle system is also a bottleneck to the accuracy and scalability of the system. If the number of nodes in the system was increased we should see a greater separation in performance between oracle and the distributed version.

Secondly, the distributed systems (and oracle) have locality-awareness which does not exist in the random cache strategy. In the random cache strategy, clients cannot avoid far away cache nodes. Clients therefore end up sending requests to caches all over the world when there may be one very close-by that is underused.

Thirdly, the distributed systems (and oracle) has load-awareness which avoids it choosing caches that simply cannot perform. These overloaded caches end up giving ridiculously high end-to-end response times, rendering any scheme using them such as the random scheme, utterly useless.

Fourthly, the distributed (CC) system (and oracle (CC)) has a loose content location strategy. Its ability to ask neighbours for cached content saves it a huge amount of time when replying to clients.

Fifthly, the loose content location scheme used by the distributed (CC) and oracle (CC) systems has a side affect, being content replication. Content replication also helps its performance by giving users additional points of access.

Lastly, the oracle and distributed system especially, allows for high dynamism within the network. All location and load properties are updated dynamically at a very low costs due to network coordinates being a proactive network-aware overlay. Dynamism is a huge advantage as load and locality of a node will inevitable change. Changes in routing paths, data traffic or computational are all picked up with network coordinates. The systems also allow for clients and caches to enter and exit the system without disrupting the system. The basic random cache strategy needed each client to have a list of predefined cache host names. This means that new caches entering or exiting the system are not handled at all by this strategy.

8.2.4 State-of-the-art Comparison

The cache-centric distributed system developed offers huge benefits that many state-of-the-art system fail to deliver. Although accuracy can be poorer in my system, for an application such as a distributed web cache, network coordinates work very well.
CoralCDN has both locality and load aware features. However, its load-awareness is not as sophisticated as one would have hoped. It also uses its load information to only prioritize caches. My system however, uses load in a more promising method. My system manages to calculate a good trade-off between both locality and load information, letting the coordinates decide which cache to pick. It also provides this information under one mechanism, simplifying the process and eradicating any additional overheads that might have been used in any other state-of-the-art system.

OASIS’s use of a geographic location mapping mechanism has shown to be very accurate. However, its location-awareness is relatively fixed as it is based on geographic locations (although probing is still done). OASIS also has poor load-awareness properties, making it quite a bad system to be used for an application such as web caching. As described before, locality awareness without load awareness is a dangerous combination. This could potentially lead to clients choosing caches in the same geographical area, overloading caches in dense cities and underusing caches in geographically remote locations.

In comparison to Meridian, the distributed web cache developed is highly competitive. Meridian does on-demand probing on each request and therefore its overheads are very high. Latency penalties also exist and is for this reason that it is usually used in applications that need long-lived contact. Our system however, has fixed maintenance overheads, as proactive measurements are taken periodically. These measurements have been shown to produce coordinates that are accurate enough for short-lived requests such as HTTP requests. For a web cache application, Meridian would be highly inefficient and although give accurate decisions for which cache to use, would contain too high a penalty for finding this cache.

However, a downside to my system would be that the measurements being performed give an additional computational strain on the machines. This however is minimal and most probably less than the strain placed on machines with other state-of-the-art systems running.

Another more serious disadvantage to my system is the need for clients to have an application running. This is something that is unfavourable amongst internet users. Other state-of-the-art systems do not have such a problem as they have other methods such as the running of DNS server etc...

All in all, both the oracle and distributed systems developed show good competition with leading systems such as Meridian, CoralCDN and OASIS. This is because no scheme can offer both locality and true load awareness at such a low fixed overhead cost. The distributed web cache also has benefited greatly from its content location and replication technique (cache-centric) as hit ratios were improved when used.

8.3 Challenges

Testing the system was much harder than anticipated. Although PlanetLab is a great tool at academics disposal, there is a huge amount of work needed to be able to deploy a system out to multiple nodes. Just some of the operations and tasks I needed to constantly perform are listed below.
• Synchronization of executable class files and a Java distribution onto multiple nodes.
• Script to run the required parts of the system with the copied Java distribution.
• Scripts to run the system on multiple nodes in parallel.
• Scripts to manage the nodes on the PlanetLab slice.
• Constant checking of nodes to see if alive.
• Scripts to analyse huge amounts of logged data.

A number of other challenges needed to be overcome whilst evaluating the system. A summary of a few issues can be seen below.

• Machines being rebooted during experimentation - Quite a problem when running experiments remotely as you have no way of restricting access to the physical machine. Many machines were rebooted when long-term experiments were being performed.

• Bugs found only with large number of nodes running - A common problem when developing large multi-threaded systems. Bugs seemed to crop up when working with a larger number of nodes. This required remote logging, understanding and fixing.

• PlanetLab burst rate capped - One of the most irritating of issues was exceeding my slice’s data transmission allowance. I was penalized by having nodes’ data transfer burst rates capped to 15kb/s and sometimes even 0kb/s during experimentation. Clearly this brought anomalies in the data collected and experiments needed to be performed again.

• Analysis of data - Once all the data of an experiment is logged, analysing it can be hard. Log data filled a large number of files with many tens of thousands of entries each. Python scripts were used to iterate through data and select the pieces that were relevant. Developing such scripts was time consuming and quite tricky to get right.
Chapter 9

Conclusion & Further Work

Before starting this project, the advantages of using network coordinates in an application were not so clear. Although in theory they seemed promising, in practice they could have been quite the opposite. This investigation has shown that network coordinates are in fact very under-rated. Once extended, they are an excellent choice for applications such as a distributed web cache, as they can potentially offer much more than any other technique available.

We have shown that network coordinates can be extended to include true computational and network load properties. This is a great achievement on its own. With such findings, a fully functional and scalable distributed web cache has been designed and developed. The distributed web cache offers features no other web caching system can offer. Thanks to the extended network coordinates, it not only has unique locality awareness properties but also has awesome load balancing features built into the same mechanism. It also has incorporated an ingenious content location and replication mechanism, dubbed the Cache-Centric request handling technique.

The distributed web cache itself is modular in design and allows for additional features and extensions to be easily integrated. It also allows for communication to be defined in a very high-level format, making the addition of messages and actions extremely simple.

Many existing systems struggle to offer users so many benefits within such a dynamic environment. It is however the dynamism that makes this system thrive. For example, if any other system were to be permanently deployed on PlanetLab, much work would have to be done in selecting the appropriate stable nodes on which to run the caches. With my system, the separation of clearly inappropriate nodes is dynamically defined. Nodes that struggle to perform even to a reasonable level will be left out as they will be placed far away in the coordinate space from any other nodes. Although systems have similar properties, they do not have such accurate load metrics in their calculations.

With all its features combined, the developed distributed web cache is a highly sophisticated and intricate system. In a world of increasing internet demands, large concentrated pockets of activity and an increase in the spread of users, a system like ours will feel right at home.

On the whole, the project can therefore be seen as a successful investigation. All initial aims of the project have been completed and evaluated successfully.
9.1 Further Work

If time had allowed, there are a number of refinements and possibly new directions to take this project. These have been divided up into two sections respectively.

9.1.1 Refinements

- The caching functionality could be extended to handle concurrent identical requests with one single connection to the origin or secondary cache. We could therefore send the requested data to multiple clients without having to open more than one connection. This would save on bandwidth of the cache and therefore reduce the total request time.

- Try and improve the efficiency of the node manager. At the moment the system works with object serialization, however, one could minimize the data overhead by defining custom low-level message formats.

- The visualization module still contains bugs thanks to the Prefuse library. These bugs were out of my control as I was unaware of their presence once started development. I would ideally like to fix these bugs and therefore clean up any nasty work-arounds I had to incorporate.

9.1.2 New Directions

There are also a number of very interesting extensions to the project which warrant further investigation.

One possible extension to the investigation would be to try using Proxy Network Coordinates for clients. Proxy network coordinates is a new notion only recently proposed [41]. They allow for the coordinates of a node to be calculated without the need of RTT samples, but instead calculated by another remote host. This host could for example be a cache. Interestingly enough, proxy network coordinates’ accuracy and stability properties are comparable to directly maintained network coordinates. We would therefore be improving the efficiency of the system and making the client much more light-weight at no extra cost.

Another potential direction to take this investigation is to develop a more sophisticated content location mechanism on top of the system. This may keep track of which caches have what content and direct users requests to them when needed. Such an extension would have to be evaluated against the existing system in order to be justified as an improvement on performance.
9.2 SMV Model for the Channel Locking Mechanism

module host(nr, request, requestOther, response, responseOther, 
    sending, release, releaseOther, turninit) {

    output request, response, release, sending : boolean;
    input nr, requestOther, responseOther, releaseOther, 
        turninit : boolean;

    state: {idle, requesting, slave, responding, master, 
        finishing, wonCompetition, lostCompetition};
    turn: boolean;

    init(state) := idle;
    init(turn) := turninit;

    if (state = idle)
        if (requestOther)
            next(state) := responding;
        else
            next(state) := {idle, requesting};
    else if (state = requesting)
        if (requestOther)
            if (turn = nr)
                next(state) := wonCompetition;
            else
                next(state) := lostCompetition;
        else if (responseOther)
            next(state) := master;
        else
            next(state) := requesting;
    else if (state = lostCompetition)
        next(state) := slave;
    else if (state = wonCompetition)
        next(state) := master;
    else if (state = master)
        next(state) := {master, finishing};
else if (state = responding)
    next(state) := slave;

else if (state = slave)
    if (releaseOther)
        next(state) := idle;
    else
        next(state) := slave;

else if (state = finishing)
    next(state) := idle;

if (state = wonCompetition | state = lostCompetition)
    next(turn) := ¬turn;

request := (state = requesting);
response := (state = responding);
sending := (state = master);
release := (state = finishing);
}

module main() {

    host0_req, host0_res, host0_rel, host1_req, host1_res, host1_rel,
    host0_send, host1_send : boolean;
    turn1, turn0: boolean;

    init(turn0) := 0;
    init(turn1) := 0;

    host0 : host(0, host0_req, host1_req, host0_res, host1_res,
        host0_send, host1_send : boolean;
    host1 : host(1, host1_req, host0_req, host1_res, host0_res,
        host1_send, host1_rel, host0_rel, turn0);
    host1 : host(1, host1_req, host0_req, host1_res, host0_res,
        host1_send, host1_rel, host0_rel, turn0);

    SPEC AG (¬(host0_res & host1_res));
    SPEC AG (¬(host0_send & host1_send));

    SPEC AG (host0_req -> AF host0_send);
    SPEC AG (host1_req -> AF host1_send);

    FAIRNESS host0.state = requesting;
    FAIRNESS host1.state = requesting;
}

9.3 CoTop Recordings for Disk-Memory Experiment

9.3.1 Cache Node - Disk Test

Cotop URL : http://planetlab-3.amst.nodes.planet-lab.org:3120/cotop

cotop - 11:22:26 up 54 days, 22:12, 0 users, load average: 5.07, 6.23, 6.36
Tasks: 63 total, 1 running, 62 sleeping, 0 stopped, 0 zombie
Cpu(s): 52.2% us, 42.2% sy, 0.0% ni, 1.1% id, 2.6% wa, 0.5% hi, 1.4% si
Mem: 1035932k total, 1016080k used, 19852k free, 10328k buffers
9.3.2 Cache Node - Memory Test

Cotop URL: http://planetlab-3.amst.nodes.planet-lab.org:3120/cotop

cotop - 11:35:13 up 54 days, 22:25, 0 users, load average: 4.86, 6.27, 6.37
Tasks: 63 total, 1 running, 62 sleeping, 0 stopped, 0 zombie
Cpu(s): 52.2% us, 42.2% sy, 0.0% ni, 1.1% id, 2.6% wa, 0.5% hi, 1.4% si
Mem: 1035932k total, 1011088k used, 24844k free, 7260k buffers
Swap: 1048568k total, 659244k used, 389324k free, 200912k cached
Swap: 1048568k total, 45492k used, 1003076k free, 256992k cached

9.3.3 Client Node - Both Tests

Cotop URL: http://planetlab01.sys.virginia.edu:3120/cotop?sort=1

cotop - 11:22:15 up 24 days, 10:02, 0 users, load average: 2.65, 2.55, 2.43
Tasks: 63 total, 1 running, 62 sleeping, 0 stopped, 0 zombie
Cpu(s): 19.5% us, 17.6% sy, 0.0% ni, 60.4% id, 1.9% wa, 0.1% hi, 0.5% si
Mem: 1034716k total, 1017276k used, 17440k free, 12972k buffers
Swap: 1048568k total, 45492k used, 1003076k free, 256992k cached

9.4 Linux Distribution Mirror XML File

The XML list used when running the client requester can be seen below. Location tags gave the base URL for the distribution directory. The client requester then appended an additional path to get to a specific file in the directory. All directories within all the mirrors have the same directory structure.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<list>
  <country value="Asia">
      Japan Internet Initiative Japan (Asia)
    </location>
  </country>

  <country value="Europe">
    <location value="http://mirrors.sec.informatik.tu-darmstadt.de/gentoo/" class="country_DE continent_EU">
      Germany Technical University of Darmstadt (Europe)
    </location>
    <location value="http://mirror.ing.unibo.it/gentoo/" class="country_IT continent_EU">
      Italy University of Bologna (Europe)
    </location>
  </country>

  <country value="North America">
    <location value="http://mirror.datapipe.net/gentoo/" class="country_US continent_NA">
      USA Datapipe Managed Hosting (North America)
    </location>
  </country>
</list>
```
9.5 Visualization Module Screenshots
Figure 9.1: Shows a screenshot of a deployment of caches on the PlanetLab network using the visualization module developed.
Figure 9.2: Shows a screenshot of a large PlanetLab deployment using the visualization module developed.
Figure 9.3: Shows a screenshot of a large PlanetLab deployment using the visualization module developed.
Figure 9.4: Shows a screenshot of a small local area network deployment with a single client and two caches.
Bibliography


