Herodotus: A Peer-to-Peer Web Archival System

by

Timo Burkard

Bachelor of Science in Computer Science and Engineering,
Massachusetts Institute of Technology (2002)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2002

Author . . . . . .
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by . . . . . .
Robert T. Morris
Assistant Professor
Thesis Supervisor

Accepted by . . . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Herodotus: A Peer-to-Peer Web Archival System

by

Timo Burkard

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, we present the design and implementation of Herodotus, a peer-to-peer web archival system. Like the Wayback Machine, a website that currently offers a web archive, Herodotus periodically crawls the world wide web and stores copies of all downloaded web content. Unlike the Wayback Machine, Herodotus does not rely on a centralized server farm. Instead, many individual nodes spread out across the Internet collaboratively perform the task of crawling and storing the content. This allows a large group of people to contribute idle computer resources to jointly achieve the goal of creating an Internet archive. Herodotus uses replication to ensure the persistence of data as nodes join and leave.

Herodotus is implemented on top of Chord, a distributed peer-to-peer lookup service. It is written in C++ on FreeBSD.

Our analysis based on an estimated size of the World Wide Web shows that a set of 20,000 nodes would be required to archive the entire web, assuming that each node has a typical home broadband Internet connection and contributes 100 GB of storage.

Thesis Supervisor: Robert T. Morris
Title: Assistant Professor

Acknowledgments

I would like to thank my thesis advisor, Robert Morris, for his guidance and advice on this thesis. He helped me to define the topic of this thesis, and to focus on the real problems. I also worked closely with Frans Kaashoek and David Karger, and I would like to thank them for their invaluable feedback and assistance.

Many members of the PDOS group here at LCS have supported me a great deal. I would specifically like to thank Frank Dabek, who answered countless questions about Chord. David Mazières helped me when I had questions about the SFS libraries. Russ Cox and Emil Sit were helpful me when I had questions about Chord and RPC. Doug De Couto and Chuck Blake helped me find machines and disk space that I could use to perform my crawls. David Andersen let me use the RON testbed and answered all my Latex questions.

I would like to thank David Ziegler and Eamon Walsh for proofreading my thesis and providing valuable feedback.

Special thanks to David S. Bailey, whose many AGSes were truly inspirational.

Contents

Chapter 1

Introduction

In 1996, the Internet Archive Wayback Machine [3] started archiving the World Wide Web as it evolved over time. As a non-profit organization funded by several companies, the Wayback Machine captures snapshots of popular web sites (HTML and graphics) at periodic intervals.

Like a search engine, a web archive increases the value of the Internet. While a search engine facilitates finding certain pieces of information, a web archive ensures that data published on the web is stored persistently and remains accessible indefinitely. The web provides a wealth of information. However, many sites are frequently taken down or restructured, which could result in potentially interesting information becoming unavailable. A web archive constantly crawls the web and keeps a copy of all content, and allows users to type in a URL and date to see what a given site looked like in the past.

Archiving the Internet is difficult because of the vast amount of storage and bandwidth required to accomplish this goal. On their website, the Wayback machine reports total hardware expenses to date of $400,000 for servers at their central download site. In order to download 10 TB a month, an available bandwidth of 31 MBit/s is required. At current bandwidth prices, this translates into $375,000 of Internet access costs per year. These figures show that if run from one centralized site, a large investment, on a commercial or government scale, would be necessary.

With Herodotus, we present a solution that achieves the same goal using the same total amount of resources, but that massively distributes the task of archiving the web over

thousands of collaborating peer-to-peer nodes. As such, a large group of people or institutions (such as universities or corporations) can make small contributions of hardware and bandwidth resources in order to collaboratively archive all HTML and image content of the World Wide Web. The Wayback Machine is a joint effort of several parties, and each party contributes money to operate the central data center. Herodotus on the other hand allows participants to contribute machine and bandwidth resources directly. This scheme is economically more efficient because providing excess resources often has very little or no cost for the participating parties.

A distributed peer-to-peer web archive faces challenges that a centralized system does not. The work of fetching and storing pages must be partitioned across the nodes. Links found on newly downloaded pages must be forwarded in an efficient manner to the node responsible for that part of the URL space. User queries have to be forwarded to a node that stores the actual URL. Finally, since peer-to-peer nodes can be unreliable and join and leave the system, it is crucial to use replication to achieve persistent storage of the downloaded content over time.

Herodotus addresses all these issues. The way that Herodotus automatically replicates content as nodes join and go eliminates the need for maintenance staff that a centralized solution requires. As long as new nodes join the Herodotus network to accommodate the storage and bandwidth needs, Herodotus automatically manages resource allocation and achieves fault-tolerance.

While we have built a working version of Herodotus, we have only used it on a very small scale to download all content of the MIT domain, totaling about 1.4 million URLs. Due to the immense effort necessary to recruit a large number of participating peers, we have not yet deployed Herodotus on a larger scale.

The remainder of this thesis is organized as follows. After looking at related work, we will use data gathered from our MIT crawls to understand the nature of the problem of archiving the entire world wide web. Next, we will present the design of Herodotus. Then, we will describe the status of our current implementation. After that, we will analyze how many nodes would be required to use Herodotus to archive the entire Internet, and describe requirements for those nodes in terms of storage space, available bandwidth, and uptime.

Finally, we will end with a conclusion reviewing what we have accomplished.

Chapter 2

Related Work

As we have mentioned in the Introduction, the Internet Archive Wayback Machine [3] is currently the only system that is archiving the web. It has been operational since 1996. In its early years, only a small subset of the web was archived, and the rate at which sites were downloaded ranged from once every few days to once every few months. More recently, the Wayback machine is crawling the web much more aggressively, adding 10 TB of new data per month to their current data repository of 100 TB. As a project run by a non-profit organization backed by several companies, little public about the internals of the Wayback machine. However, the information available on its website makes clear that it is run from one central site, requiring a $400,000 investment in hardware, a large amount of available bandwidth, and expensive dedicated support staff to manage the server farm. In contrast, Herodotus operates in a peer-to-peer fashion, allowing a large number of small parties to contribute resources to collaboratively archive the web. As a self-managed application that automatically allocates work among peers and replicates data, the need for dedicated support staff is eliminated.

A subproblem of archiving the web is crawling the web. Popular existing crawler projects include Google [8] and Mercator [12]. However, both of these systems operate in LAN-environments with high-speed links between the collaborating machines. Machines in such an environment typically have a high level of reliability, so that machine failures are not really considered to be an issue. In contrast, Herodotus is a distributed crawler that operates in a peer-to-peer fashion. In such a setting, the links between cooperating

machines are very expensive, and machines frequently join and the leave the set of collaborating nodes due to temporary outages and machine failures. Herodotus uses a number of techniques to adequately address these issues.

Herodotus is built on top of the peer-to-peer lookup system Chord [16]. Chord provides a framework in which peer-to-peer machines are organized in a fault-tolerant communication scheme. Applications built on top of Chord are provided with a hash function that maps any key to a unique Chord node that is responsible for that key. It is up to the application to decide what keys actually constitute, and how they correspond to data or tasks that are divided across the Chord machines. In Herodotus, Chord is used to partition the URL space among all participating nodes. As we will see, the Chord hash function is applied on the URL to determine the node responsible for that URL. When the set of currently participating hosts changes because of joining or leaving Chord nodes, Chord signals to all affected Chord nodes when responsibilities of certain hash values have been reassigned so that the state associated with these hash values can be transferred accordingly.

One application of Chord is CFS, Cooperative File Storage [9]. CFS achieves fault-tolerant distributed storage among Chord nodes. While Herodotus could have used CFS to store downloaded content, we decided to use a simpler approach that stores downloaded data on local disks. One of the design goals of CFS was to achieve good load balancing of downloads across the nodes. An underlying assumption of CFS is that relatively few data files are being inserted into the system, but that some of that data is highly popular (like in shared music storage systems). In Herodotus, the opposite is true. A vast amount of data needs to be stored in a fault-tolerant manner, but the effect of accesses is negligible. The bulk of the operations constitute the download and storage of data, not the retrieval. As such, storing data locally is more efficient compared to CFS, where inserting new data triggers a large amount of information being communicated between nodes to store the new data at many locations.

Chapter 3

The Problem

In this chapter, we will analyze the complexity of the problem of archiving the entire Internet. Understanding what data volume our system needs to be able to deal with helps us to understand what requirements our design will need to satisfy in order to adequately solve the problem. Our analysis is based on data that we have obtained from archiving all web pages of the MIT domain. We will use the findings of related research to extrapolate the numbers we obtained in our MIT experiment to estimate corresponding metrics of the entire World Wide Web.

This chapter is organized as follows. We will first describe the setup of our MIT experiment and give the data that we have obtained. Next, we will derive similar metrics for the entire web. In the final part, we will discuss implications for our design.

3.1 The MIT experiment

In order to understand the composition of the World Wide Web, and at what rate it changes, we archived all HTML files and images of the MIT domain over the course of a week. Initially, we started at the main page of `web.mit.edu` and followed all links to download and store the entire MIT domain. Since we are only interested in the World Wide Web, we limited the download to HTML files and graphics (JPEG and GIF). In order to avoid overloading web servers with requests for dynamic content and potentially crawling infinitely many pages generated on the fly, we did not download any dynamic content (such as CGI).

When downloading a URL that we had already downloaded before to see if it had changed, we used conditional GETs to download only those pages that had been updated. We also followed new links that we had not seen the before.

The following table summarizes the results that we obtained from five crawls during a five-day period. Since we limited the download to MIT content, URLs pointing to non-MIT content were discarded.

Property	**Value**
Number of unique URLs downloaded over all crawls	1,399,701
Percentage of URLs that were images	43.5%
Average size of an HTML file	12 kByte
Average size of an image file	32 kByte
Average percentage of HTML files that changed per day	1%
Average percentage of images that changed per day	$< .01\%$
New objects per day as a percentage of existing objects	0.5%
gzip compressed size of HTML files	24.8%
gzip compressed size of image files	98.6%
Average number of links embedded in an HTML page	36
Average length of URLs (in characters)	59

Table 3.1: Data obtained in the MIT experiment.

This data shows that for every HTML page, there are 0.77 images. The size of an image is significantly larger than that of an HTML file. Furthermore, applying gzip on HTML files yields a compression ratio of 4:1, whereas on image files, gzip has almost no effect, which can be attributed to the fact that image files are in compressed form already.

In the next section, we will use the data above to extrapolate these properties to the entire World Wide Web.

3.2 Extrapolation to the entire World Wide Web

In this section, we will extrapolate the results obtained in the previous section to the entire World Wide Web. Specifically, we will estimate the size of the World Wide Web, and the amount of storage required to capture all changes over time. Finally, we will summarize

our results in tabular form. In the subsequent section, we will use these estimates to derive design requirements for Herodotus.

3.2.1 Size of the World Wide Web

Since we did not attempt to crawl the entire Internet for the purpose of this analysis, we rely on other sources of information. The most popular search engine, Google [2], claims that it has indexed a little more than 2 billion web pages. This size matches the rough estimate of a few billion pages that the founders of Google gave in their research paper in 1999 [8].

Douglis, Feldmann, and Krishnamurthy report that the average size of an HTML document is 6.5 kBytes [10]. Our MIT data shows an average size of an HTML document of 12 kBytes. Since the research report dates back to 1997, our higher number is most likely reflective of the fact that web pages have become more complex in the past five years. Consequently, we will use the MIT figure for our calculations. Multiplying this average size with the number of web pages gives us an expected size of all HTML documents on the web of 24 TB.

Our MIT experiment has shown that HTML files can be compressed by a factor of four. If Herodotus were to store the web in the most efficient manner, one snapshot would therefore require only 6 TB of storage.

Since the World Wide Web consists of both HTML and embedded images, we have to account for the latter as well. Unfortunately, we have found no research on the number and average size of images. Therefore, we extrapolate the numbers obtained from our MIT study to the entire population of web pages. Given an average of 0.77 images per HTML page, we would expect the web to contain about 1.54B image files. Since the average image size was 32 kBytes, all images on the web amount to roughly 50 TB of data. Unlike HTML, images are already in a compressed format. Therefore, Herodotus could not save storage space by applying additional compression on the downloaded data.

Combining the numbers above yields an overall size of 74 TB that could be stored in 56 TB of storage space after using gzip compression on HTML content.

3.2.2 Rate of change of the World Wide Web

In order to estimate the rate of change of the Internet, we have to consider two different types of changes: updates to existing objects, and newly created objects.

Let us first look at updates to existing web objects. Unfortunately, relatively few research results are available on this topic. Based on a sample of 100,000 pages observed over a long period of time, Brewington estimates that 5% of all web pages change every day [11]. Our MIT experiment shows that only 1% of the pages change within the MIT domain every day. However, our MIT figures might be biased, since MIT is an academic institution, and its web pages might change less frequently than those of many commercial web sites. To be conservative and rather overstate the effect of updates, we will assume a rate of change of 5% per day for HTML files. Our MIT study shows that image files almost never change ($< .01\%$), so we neglect that effect. Therefore, we expect to produce 1.2 TB of uncompressed data every day because of updates to existing objects (300 GB after accounting for compression). Notice that in these calculations, we have assumed that when a page changes, we store a compressed version of the new HTML file. If pages change only slightly, it might be more efficient to store diffs of pages relative to their previous versions.

Next, we will look at changes due to newly created objects. Since we have found no research results on this, we will base our estimate on the numbers of our MIT study. Our MIT study shows that on average, 0.5% of the current number of web pages and images is being added every day. This translates into 370 GB of uncompressed data per day or 280 GB per day after accounting for compression of HTML files (using the estimated size of the web in the previous subsection).

Since our goal is to capture changes of web pages on a daily basis, we therefore expect that we need to download about 1.57 TB of new uncompressed data every day. Using compression, this data can be stored using 580 GB of storage space every day. Over a one month period, this means a total storage capacity of roughly 17.5 TB of data.

As a sanity check, we compare these numbers to statistics of the Wayback Machine [3]. The Wayback Machine claims to add 10 TB of data every month, which is of the same order of magnitude as our 17.5 TB figure. However, the Wayback machine does not query

every URL every day, but bases the download frequency of a URL on the rate at which it has changed in the past. Therefore, the Wayback machine does not capture all changes, but only a large fraction of them. It is not clear whether or not that number is referring to compressed data or uncompressed data. In addition, the Wayback machine claims to have a total of 100 TB data, but claims that it is adding 10 TB every month. Since the Wayback machine has been operational since 1996, the total amount of data of 100 TB seems extremely low compared to the additional 10 TB per month. However, the fact that the Wayback machine only stored a very small subset of the Internet until recently might account for this discrepancy.

3.2.3 Summary of results

In this subsection, we will summarize the figures from the previous two subsections for ease of future reference.

In the table below, *HTML updated* refers to HTML that has been changed since the last download, *HTML added* refers to newly created HTML pages, and *HTML new* refers to the sum of the two. Since our MIT study shows that the effect of modified images can be neglected, we only have one category for images, *Images new*. This refers to newly created images.

Description	**# of objects**	**Size (uncompressed)**	**Size(compressed)**
Snapshot HTML	2B	24 TB	6 TB
Snapshot Images	1.54B	50 TB	50 TB
Snapshot total	3.54B	74 TB	56 TB
HTML updated per day	100M	1.2 TB	300 GB
HTML added per day	10M	120 GB	30 GB
HTML new per day	110M	1.3 TB	330 GB
Images new per day	7.7M	250 GB	250 GB
Total new per day	118M	1.6 TB	580 GB
HTML new per month	3.3B	40 TB	10 TB
Images new per month	231M	7.5 TB	7.5 TB
Total new per month	3.5B	47.5 TB	17.5 TB

Table 3.2: Summary of extrapolated characteristics for the entire WWW.

3.3 Design Implications

In this section, we will dicuss the requirements that Herodotus has to satisfy. First, we will discuss general properties that a distributed web archival system must have. In the second part, we will discuss what additional constraints the numbers that we have found in the previous sections impose on a design in order for it to be feasible.

3.3.1 General properties

The previous section has shown that archiving the World Wide Web on a day-to-day basis involves processing and storing large amounts of data. As we have pointed out, a peer-to-peer system is well suited if we want to achieve this goal by having a large number of small nodes collaborate. Herodotus will need to address the following issues that arise as a result of distributing the work load across a set of peers.

Keep track of the set of active peers. Since peers can fail or new peers can join the system, Herodotus will need to keep track of active peers so that the work is distributed correctly.

Distribute the work of downloading and archiving objects. Since an individual node can only deal with a small fraction of the entire web, Herodotus has to provide a way to partition the job of downloading and archiving the data every day across all currently active peers. In particular, Herodotus should ensure that work is not unnecessarily duplicated (e.g. by having too many peers download the same object), and that all work that needs to be done is actually completed by some node.

Balance the load across the peers. When distributing the work across the peers, Herodotus needs to balance the workload assigned to each node, taking into account the storage and download capacity of each node. This is important to avoid overloading certain nodes, and to ensure that the system as a whole can complete entire crawls of the web in a timely fashion.

Replicate content to achieve fault tolerance. In a peer-to-peer system, individual nodes can permanently fail or disappear. Since it is imperative for an archive to retain the historical data over extended periods of time, Herodotus must use replication to store the

same data on multiple peers in a redundant fashion. As peers holding certain pieces of data disappear, Herodotus needs to replicate that data to additional peers to maintain a high enough of level of fault tolerance that will make losses highly unlikely. Besides ensuring that the stored data will be persistent, replication also allows Herodotus to serve historical data to users while some peers holding that data might be temporarily unavailable (e.g. for reboot or maintenance).

Provide a user interface. In order to allow users to access the historical web pages stored in Herodotus, Herodotus needs to provide a simple interface that fetches the requested content from the peer node keeping that information.

3.3.2 Design consequences of the estimated dimensions of the web

In this subsection, we will describe what additional constraints Herodotus needs to satisfy when considering the dimensions of the web outlined in the previous section.

Large number of nodes. The tabulated results for the entire WWW show the tremendous amount of storage required to operate Herodotus. If we assumed compressed storage, only the first year of operation will consume 266 TB of storage if we archive images and HTML. Since replication will be necessary to achieve fault-tolerance, this number is multpilied by the level of replication that we choose. If the level of replication is 6 for example, 1.6 PB of total storage space will be necessary. If each node stores 100 GB of data, this means 16,000 nodes will be necessary. Therefore, it is important that Herodotus scales well to a large number of nodes.

Distributed list of seen URLs. A key component of a crawler is a list of URLs that has already been processed, to avoid duplicate downloads. Ideally, we would want to store that list on every node, so that we can identify links that have already been encountered early and do not need to waste bandwidth to send them to other peers. However, our tabulated results estimate a total of 3.54B URLs. If we decide to only store the SHA1 hash values of each URL (which is sufficient to identify URLs already encountered), this still translates into 70 GB of storage (since each SHA1 hash value is 160 bits long). As a consequence, it is impossible to keep the entire list of URLs already encountered everywhere. Instead, each

node should maintain a complete list of URLs already encountered only for those URLs that it is responsible for downloading and storing. In order to avoid resending popular URLs that appear over and over again, each node might decide to additionally cache the most popular URLs of all those URLs that it is not responsible for.

Cost of replacing nodes. Suppose each node stores 100 GB of data. If a node leaves, and a new node joins the network, 100 GB of data will need to be blasted to that new node to maintain the same level of replication. Even if we assume that nodes have downstream bandwidths of 1 MBit/s, it would take nine days to download the entire data from other nodes. Given the large number of peers necessary, we might very well have peers with lower bandwidth capabilities. Therefore, we conclude that nodes need to remain part of the system for long periods of time, on the order of at least a a few months, so that restoring lost state of nodes that have permanently left the system does not consume too much resources. In addition, if a node remains part of the system for less than a month, it should have rather not joined at all. This is because it causes other peers to dedicate a significant portion of their bandwidth to bring it up to speed, while it contributes very little to to the long-term archival process.

Deal with temporarily unavailable nodes. The previous point has demonstrated that nodes need to be part of Herodotus for at least a few months. On the flipside, very few machines or Internet connections have permanent uptimes of a few months. Therefore, Herodotus should be robust enough to tolerate temporary outages of nodes. In particular, it should make use of the data that is still persistently stored on the node, and use replication to restore only that information that the node missed while it was unavailable.

Chapter 4

Design

In this chapter, we will describe the design of Herodotus. First, we will give an overview of the general way in which Herodotus operates. While the overview conveys the general scheme in which Herodotus operates, many details are left open. The following sections fill these gaps by describing certain aspects of the design in more detail. All parts of this design have been fully implemented. In some sections, we give multiple possible design choices, and describe which ones we picked and why.

4.1 Overview

Herodotus performs three main functions: continuously crawling the web, replicating content to achieve fault tolerance, and providing users with an interface to view archived web content.

All three of these functions require that the collaborating peers are organized in some network topology, and Herodotus uses Chord [16] to achieve this goal. In a nutshell, Chord enables nodes to find each other and to know about each other. As an external interface, Chord exports a lookup function that allows a mapping of any kind of data to a node within the Chord network. This mapping function is the same across all participating machines. Internally, Chord nodes are organized in a ring structure (see next section). Herodotus uses the Chord lookup function to determine which node is responsible for a given URL, and to delegate the task of downloading and storing a URL to that node.

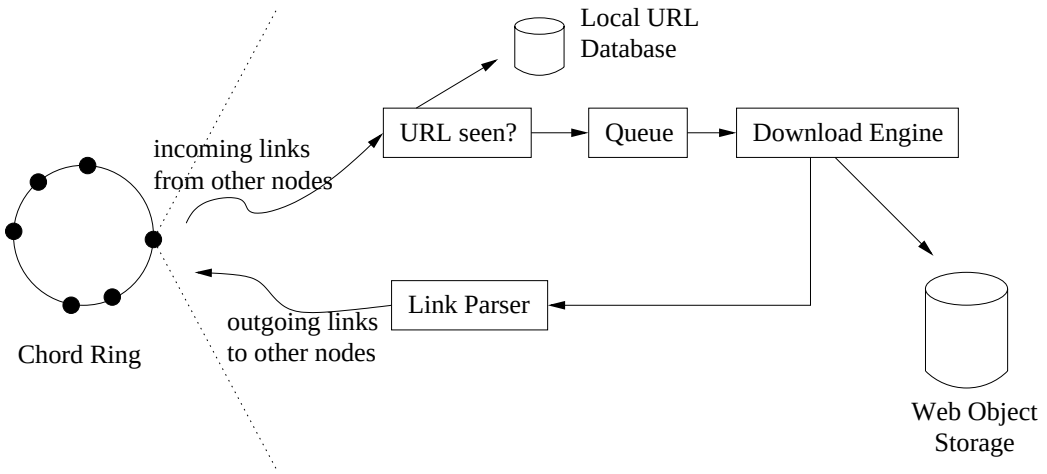

Figure 4-1: Design Overview: Chord ring on the left, the operations performed inside each node during a crawl on the right.

Figure 4-1 gives an overview of how Herodotus continuously crawls the web. The left hand side of the figure shows how collaborating peers are distributed on the Chord ring. The right hand side shows what goes on in each peer node. The node receives URLs that it is responsible for from other peers through Chord. If the node has already processed a given URL on that day, that URL is simply discarded to avoid multiple downloads. If not, the URL is put into a queue of objects that still need to be downloaded. The Download Engine maintains a number of concurrent connections to web servers to download the queued objects. Once the download of an object has completed, the data is stored in the Web Object Storage on the local file system, and if the document is an HTML page, it is forwarded to the Link Parser. The Link Parser identifies all references to other HTML files and to images. Those extracted links are then sent through the Chord network to the nodes responsible for the respective URLs.

While this overview gives a general picture of how Herodotus operates, the following sections will describe certain aspects in more detail. The next section will describe Chord and the way peers are maintained in more detail. Then, we will describe how the Chord lookup function is employed to map URLs to nodes. Next, we will examine how links can be sent between the nodes in the most efficient manner. After that, we will see how Herodotus keeps all state on nodes in a persistent manner so that it can safely recover from

temporary outages, such as reboots. Next, we will address how Herodotus uses replication to achieve fault tolerance as a protection against node failures. After that, we will examine two operational issues: optimizations to reduce bandwidth usage, and how the above framework can be used to continuously crawl the web on a daily basis. Finally, we will describe how users can use Herodotus to retrieve archived versions of web pages in their browser.

4.2 Using Chord to maintain peers

Herodotus uses Chord [16] to maintain the set of participating peers, to distribute work across the peers, and to locate archived content. Chord supports just one operation: given a key, it will determine the node responsible for that key. Chord does not itself store keys and values, but provides a primitive that allows higher-layer software to build a variety of applications that require load balancing across a peer-to-peer network. Herodotus is one such use of the Chord primitive. When a Herodotus node has found a link to a URL on a page that is has downloaded, it applies the Chord lookup function to that URL to determine which node is responsible for it. It then forwards the URL to that node so that it can download and store it. The following sections describe these processes in more detail. This section summarizes how Chord works. For a more detailed description of Chord, please refer to the Chord publication [16].

We will first explain how Chord relates to Consistent Hashing. Next, we will elaborate how Chord implements its lookup function and maintains its peers. Next, we will address how Chord provides some protection against attackers that might want to replace chosen content. Finally, we will describe how Chord achieves load balancing.

4.2.1 Consistent Hashing

Each Chord node has a unique m-bit node identifier (ID), obtained by hashing the node's IP address and a *virtual node index*. Chord views the IDs as occupying a circular identifier space. Keys are also mapped into this ID space, by hashing them to m-bit key IDs. Chord defines the node responsible for a key to be the *successor* of that key's ID. The *successor* of

an ID j is the node with the smallest ID that is greater than or equal to j (with wrap-around), much as in consistent hashing [13].

Consistent hashing lets nodes enter and leave the network with minimal movement of keys. To maintain correct successor mappings when a node n joins the network, certain keys previously assigned to n's successor become assigned to n. When node n leaves the network, all of n's assigned keys are reassigned to its successor. No other changes in the assignment of keys to nodes need occur.

Consistent hashing is straightforward to implement, with constant-time lookups, if all nodes have an up-to-date list of all other nodes. However, such a system does not scale, whereas Chord provides a scalable, distributed version of consistent hashing.

4.2.2 The Chord Lookup Algorithm

A Chord node uses two data structures to perform lookups: a successor list and a finger table. Only the successor list is required for correctness, so Chord is careful to maintain its accuracy. The finger table accelerates lookups, but does not need to be accurate, so Chord is less aggressive about maintaining it. The following discussion first describes how to perform correct (but slow) lookups with the successor list, and then describes how to accelerate them with the finger table. This discussion assumes that there are no malicious participants in the Chord protocol; while we believe that it should be possible for nodes to verify the routing information that other Chord participants send them, the algorithms to do so are left for future work.

Every Chord node maintains a list of the identities and IP addresses of its r immediate successors on the Chord ring. The fact that every node knows its own successor means that a node can always process a lookup correctly: if the desired key is between the node and its successor, the latter node is the key's successor; otherwise the lookup can be forwarded to the successor, which moves the lookup strictly closer to its destination.

A new node n learns of its successors when it first joins the Chord ring, by asking an existing node to perform a lookup for n's successor; n then asks that successor for its successor list. The r entries in the list provide fault tolerance: if a node's immediate

successor does not respond, the node can substitute the second entry in its successor list. All r successors would have to simultaneously fail in order to disrupt the Chord ring, an event that can be made very improbable with modest values of r. An implementation should use a fixed r, chosen to be $2 \log_2 N$ for the foreseeable maximum number of nodes N.

The main complexity involved with successor lists is in notifying an existing node when a new node should be its successor. The stabilization procedure described in [16] does this in a way that guarantees preserved connectivity of the Chord ring's successor pointers.

Lookups performed only with successor lists would require an average of $N/2$ message exchanges, where N is the number of servers. To reduce the number of messages required to $O(\log N)$, each node maintains a finger table table with m entries. The i^{th} entry in the table at node n contains the identity of the *first* node that succeeds n by at least 2^{i-1} on the ID circle. Thus every node knows the identities of nodes at power-of-two intervals on the ID circle from its own position. A new node initializes its finger table by querying an existing node. Existing nodes whose finger table or successor list entries should refer to the new node find out about it by periodic lookups performed as part of an asynchronous, ongoing stabilization process.

Figure 4-2 shows pseudo-code to look up the successor of the node with identifier id. The main loop is in *find_predecessor*, which sends *preceding_node_list* RPCs to a succession of other nodes; each RPC searches the tables of the other node for nodes yet closer to id. Each iteration will set n' to a node between the current n' and id. Since *preceding_node_list* never returns an ID greater than id, this process will never overshoot the correct successor. It may under-shoot, especially if a new node has recently joined with an ID just before id; in that case the check for $id \notin (n', n'.\textit{successor}]$ ensures that *find_predecessor* persists until it finds a pair of nodes that straddle id.

Two aspects of the lookup algorithm make it robust. First, an RPC to *preceding_node_list* on node n returns a list of nodes that n believes are between it and the desired id. Any one of them can be used to make progress towards the successor of id; they must all be unresponsive for a lookup to fail. Second, the **while** loop ensures that *find_predecessor* will keep trying as long as it can find any next node closer to id. As long as nodes are careful

```
// Ask node n to find id's successor; first
// finds id's predecessor, then asks that
// predecessor for its own successor.
n.find_successor(id)
  n' = find_predecessor(id);
  return n'.successor();

// Ask node n to find id's predecessor.
n.find_predecessor(id)
  n' = n;
  while (id ∉ (n', n'.successor()])
    l = n'.preceding_node_list(id);
    n' = max n'' ∈ l s.t. n'' is alive
  return n';

// Ask node n for a list of nodes in its finger table or
// successor list that precede id.
n.preceding_node_list(id)
  return {n' ∈ {fingers ∪ successors}
    s.t. n' ∈ (n, id]}
```

Figure 4-2: The pseudo-code to find the successor node of an identifier id. Remote procedure calls are preceded by the remote node.

to maintain correct successor pointers, *find_predecessor* will eventually succeed.

In the usual case in which most nodes have correct finger table information, each iteration of the **while** loop eliminates half the remaining distance to the target. This means that the hops early in a lookup travel long distances in the ID space, and later hops travel small distances. It is worthwhile to note that this algorithm does not itself provide an $O(\lg N)$ bound; the structure of the finger table, which the algorithm examines, guarantees that each hop will cover half of the remaining distance. This behavior is the source of the algorithm's logarithmic properties.

The following two theorems, show that neither the success nor the performance of Chord lookups is likely to be affected even by massive simultaneous failures. Both theorems assume that the successor list has length $r = O(\log N)$. A Chord ring is *stable* if every node's successor list is correct.

Theorem 1 *In a network that is initially stable, if every node then fails with probability 1/2, then with high probability find_successor returns the closest living successor to the query key.*

Theorem 2 *In a network that is initially stable, if every node then fails with probability 1/2, then the expected time to execute find_successor is* $O(\log N)$.

These theorems about Chord ensure that it is very unlikely that failing peers can cause the Herodotus network to be partitioned into separate subnetworks that do not know of each other.

4.2.3 Node ID Authentication

If Chord nodes could use arbitrary IDs, an attacker could take control of a chosen URL space by choosing a node ID corresponding to that URL space (i.e. the successor). With control of the successor, the attacker could effectively determine what archived contents users see. This trick could be employed by people want to make information that was once published on a website inaccessible to users of the system.

To limit the opportunity for this attack, a Chord node ID must be of the form $h(x)$, where h is the SHA-1 hash function and x is the node's IP address concatenated with a

virtual node index. The virtual node index must fall between 0 and some small maximum. As a result, a node cannot easily control the choice of its own Chord ID.

When a new node n joins the system, some existing nodes may decide to add n to their finger tables. As part of this process, each such existing node sends a message to n's claimed IP address containing a nonce. If the node at that IP address admits to having n's ID, and the claimed IP address and virtual node index hash to the ID, then the existing node accepts n.

With this defense in place, an attacker would have to control roughly as many IP addresses as there are total other nodes in the Chord system in order to have a good chance of targeting arbitrary blocks. However, owners of large blocks of IP address space tend to be more easily identifiable (and less likely to be malicious) than individuals.

4.2.4 Load Balancing

As we will see in the subsequent sections, Herodotus will spread the URL space evenly around the ID space, as the hash function used uniformly distributes URLs. If each participating Herodotus node had one ID, the fact that IDs are uniformly distributed would mean that every server would carry roughly the same download and storage burden. This might not be desirable if participating nodes have greatly varying storage and bandwidth capacities.

To accommodate heterogeneous node capacities, Herodotus uses the notion of a real server acting as multiple *virtual servers*. CFS [9] uses the exact same scheme to account for varying node capabilities. The Herodotus service operates at the virtual server level. A virtual server uses a Chord ID that is derived from hashing both the real server's IP address and the index of the virtual server within the real server.

A Herodotus server administrator configures the server with a number of virtual servers in proportion to the server's storage and bandwidth capacity.

4.3 Mapping URL → Node

In this section, we will first discuss the issue of work distribution when crawling the web in a peer-to-peer fashion for the purpose of creating a web archive. Then, we will describe a simple solution that uses a direct mapping of URLs to nodes using the Chord lookup function. Next, we will discuss a more sophisticated assignment scheme in which URLs are assigned to Chord nodes based on domain names. Finally, we will describe our design choice and justify it.

4.3.1 Work distribution

Herodotus should distribute the work of downloading, processing, and storing web objects in such a way that all participating hosts are used as efficiently as possible for the duration of the crawl. Specifically, load should be distributed in such a way that all nodes finish their share of the crawl at about the same time. In the general case, such a scheme would have to take into account the available bandwidth on each machine and the sizes of the download jobs. Another resource that could vary between participating nodes is disk size, which is especially important since the goal of Herodotus is to create a persistent archive of the web. When assigning which node should store what web objects, the available resources on the machines should therefore be taken into account.

Initially, one could assume that all machines have roughly the same amount of bandwidth and storage available. To account for large differences in available machine resources, one could use the concept of virtual Chord servers that we have seen in the previous section to achieve a better use of available resources. Using this concept, load balancing can be achieved by simply spreading the work across the Chord ID space as uniformly as possible. The subsequent two sections show two ways in which this could be done.

4.3.2 Simple direct mapping

Chord provides a lookup function that maps keys to node identifiers, and the successor of that identifier is the responsible node [16]. One can simply apply the Chord lookup function to a given URL (after applying a hash function like SHA1) and delegate the job of

downloading the object to the node returned by the lookup. That node is then responsible for maintaining that part of the URL space, which means that it has to download, process, and store the associated web object.

The way in which URLs are assigned to nodes in this scheme is extremely simple, and we expect to achieve good load balancing due to the hash function properties. This is because each host will be assigned a roughly equal amount of work with high probability (assuming that amortized over all the URLs assigned to each node, the size of each object will be roughly equal).

We believe that this scheme is very effective and easy to implement, and therefore best suited for Herodotus. In the next section, we will explore a more sophisticated alternative to this approach.

4.3.3 Domain-based mapping

While the scheme described in the previous section is extremely simple to implement, it does not make use of the most important property of links, namely the fact that a large share (76%) of all links is local, i.e. within the same domain [7]. Since the Chord lookup function is applied to the hash of the entire URL, URLs of objects residing on the same host are spread out all over the Chord ring.

Because 76% of all links are within the same domain, it would make sense to map URLs to nodes in such a way that all objects residing on the same host are crawled by the same node. This would reduce the number of links that have to be forwarded to other nodes via expensive network connections (as opposed to being processed by the same node) by 76%.

A simple way to accomplish this is by applying the Chord lookup function only on the hostname of a URL (as opposed to the entire URL as in the previous scheme). In this scheme, there is a domain name to node mapping, and URLs are distributed to nodes based on their domain name and the mapping.

This method has a severe problem which renders it unusable in practice. The problem is that because of the skewed distribution of the size of websites, work will be distributed

very unevenly across the participating nodes. In particular, sites with a huge amount of data (such as `cnn.com` or `yahoo.com`) will be mapped to a single node. Clearly, an individual node cannot handle such a large site (both in terms of the available bandwidth and in terms of the available disk storage).

To make this domain-based mapping scheme scale to the skewed distribution of domain sizes that we see on the Internet today, the scheme needs to be refined so that once domains exceed a certain threshold size, they are split up across multiple nodes. The node responsible for the domain name would have to provide this level of indirection by forwarding URLs of that domain according to some second Chord lookup. That lookup could either be performed on the entire URL, or on the domain name plus some prefix of the URI, such as the first few letters, or the entire first directory (such as `cnn.com/sports/`).

4.3.4 Design Choice

Because of the scaling problems that domain-based mapping has for large domains, we have chosen the simple direct mapping scheme for Herodotus. The direct mapping scheme is much simpler, and it is easy to see how it distributes the URL space evenly across the Chord ring. While domain-based mapping could be improved by using the tricks described in the previous section, we feel this unnecessarily complicates the design by requiring additional state to be kept.

In addition, as we will see later on, we will use other ways to greatly reduce the actual number of links being sent, which will alleviate the slight disadvantage of simple direct mapping.

4.4 Exchanging links

In the overview, we have seen that the URL space is split between all participating peers, and that peers that have discovered an URL that they are not responsible for forward that URL to the peer responsible for it. The previous section has established how URLs are being mapped to peers, but we have not seen yet how links are actually transmitted to the recipient peers in an efficient manner. This will now be discussed.

First, we will describe why batching the forwarding of URLs is important. Next, we will discuss three two methods to perform the forwarding operation: direct lookup and ring-based forwarding. Finally, we will state our design choice for Herodotus.

4.4.1 Batching

Independent of which of the three forwarding schemes is being used, the forwarding can be made more efficient if batching is used. This means that URLs are not forwarded as they are extracted from webpages, but they are first put in some batch queue for the respective recipient. If either the queue of that recipient exceeds a certain length, or some timer expires, we contact the node and send it all the links that have been accumulated. The advantage is that we send multiple links at a time, reducing the amortized overhead of looking up the node and establishing a communication channel (which is very expensive compared to the small amount of bandwidth required to exchange a single link).

To illustrate how batching improves the bandwidth usage needed for link exchanges, let us take a look at actual numbers. Suppose we already know the IP address of the recipient node. Issuing an RPC request within the Chord framework to that recipient involves sending a UDP packet with 60 bytes of RPC/Chord headers and the actual URL. In chapter 3, we have seen that the average length of a URL is 59 characters. If we issue a separate RPC for each URL, the header overhead is more than 50%. If we use batching, we can simply concatenate URLs using a special separator symbol, which amortizes the 60 byte overhead over all the URLs sent at a time. If we use a set of 20,000 peers for example, storing 120,000 URLs before sending them out reduces the amortized cost of sending a URL from roughly 120 bytes to only 70 bytes.

4.4.2 Direct lookup

In this intuitive forwarding scheme, we perform a Chord lookup on every URL that we encounter, and batch URLs for each node. Once we have accumulated enough URLs for a node, we send them via RPC to the recipient node. Since every Chord lookup requires $O(\log n)$ bandwidth (where n is the number of nodes), this algorithm has the cost

$O(u \log n)$ for u URLs.

Let us again look at actual numbers. In chapter 3, we have seen that in order to store images and web content, we need at least on the order of 20,000 hosts. For that number of nodes, Stoica et al. have experimentally determined an average Chord lookup path of length 8 [16]. Since each RPC packet that is sent for a Chord lookup requires approximately 100 bytes, the lookup cost for each URL is roughly 800 bytes if we do not use batching.

Without batching, the cost of sending a single URL using this scheme is 920 bytes – which means that the overhead to send the link is 15 times as large as the actual payload (roughly 60 bytes).

Even if we use batching, we still have to perform a Chord lookup on every URL to determine what node to send it to. Therefore, even batching over a large number of nodes means an overhead of at least 800 bytes to perform the lookup for each URL. Therefore, even if we use batching, the amortized cost per URL is at least 13 times as large as the actual payload.

4.4.3 Ring-based forwarding

The main idea of ring-based forwarding is to use batching to accumulate a large number of URLs, and then "walk" around the Chord ring without any lookups to distribute the URLs to the respective nodes, asking each node for its successor. Ring-based forwarding is asymptotically better than the two schemes described above, however it requires a much larger URL buffer than the two previous schemes. The idea is to apply the SHA1 hash function on every URL, but not the actual Chord lookup. As soon as the total number of URLs collected exceeds the number of nodes times some constant, the URLs are sent to their recipients using the following algorithm. The node first contacts its successor, and sends it all URLs that it is responsible for. This operation can be performed in time proportional to the number of URLs sent to the node. In addition, the recipient reports its successor to the sending node, and the sending node next contacts that node and sends it the URLs that it is responsible for. This scheme continues until all URLs have been distributed and the successor that the last node reports is the originating node itself. Since each time

a node "walks" around the ring, it has accumulated $O(n)$ URLs, and the cost for sending these URLs is $O(n)$, the amortized cost for sending each URL is constant. This scheme effectively eliminates the $O(\log n)$ cost of performing Chord lookups, and still ships URLs directly to their recipients.

4.4.4 Design Choice

In terms of asymptotic cost, ring-based forwarding is most efficient, with an amortized cost for shipping a single link of only $O(1)$, as opposed to $O(\log n)$ that the other two methods exhibit. The problem with ring-based forwarding is that it does not scale very well to a large number of nodes, as each node has to collect a number of URLs on the order of the number of nodes before those URLs can be shipped.

In the context of Herodotus, we expect to have only on the order of a few tens of thousands of nodes. Therefore, we have chosen to use ring-based forwarding in Herodotus. To amortize the RPC overhead, each nodes accumulates an aggregate of roughly ten URLs per node. The resulting number of URLs that need to be stored simultaneously is on the order of a few hundred thousand. Given the average size of a URL of 59 bytes, this can be accomplished using only a few megabytes of memory, which is acceptable. In order to perform the temporary storage and retrieval of these URLs in an efficient manner, we use a priority queue. This allows us to quickly serve the URLs that the next node we visit on the ring is responsible for.

At the beginning and at the end of the crawl, a queue might contain very few entries, which means that we might not manage to accumulate this large number of links before we start distributing the next set of links. Therefore, we also use a timeout that ensures that links are distributed every so often. Our experiments on the MIT domain show that very soon after we start crawling, we essentially continuously traverse the ring and distribute URLs due to the large number of URLs extracted. Therefore, the overhead of distributing only a small number of links across the ring at higher amortized cost during start-up can be neglected.

Using the clever ring-based forwarding scheme, we have essentially eliminated all

costly Chord lookups. In addition, batching has reduced the amortized overhead of RPC and Chord headers to roughly 10%.

Compared to the naive direct lookup scheme, we have managed to reduce the bandwidth required to exchange links by a factor of 13.

4.5 Persistence on an individual node

As we have seen in chapter 3, it is crucial that Herodotus is robust in the face of short outages such as temporary network failures or machines crashing and rebooting. We achieve this goal by maintaining all state of Herodotus in a persistent manner on disk. If the node crashes, the contents of the file that was just being changed might be lost, but we would expect the rest of the data to remain intact, unless there was a hard disk failure.

To achieve this robustness against crashes, Herodotus stores all downloaded content as well as the current state of the queue on disk. In the following two subsections, we will look at each of these individually.

4.5.1 Downloaded content

We obviously have to store downloaded content on disk. Even if it were not for fault tolerance, it would be impossible to hold that much data in memory. There are many ways in which this data could be stored on disk. One option would be a database; however, we want to avoid having to deploy a database on each of the thousands of participating nodes. Instead, we use the local file system to organize the vast amounts of data. In order to facilitate the handling, we decided to store each downloaded object in a separate file.

Chapter 3 shows that with a reasonable network size of 20,000 nodes, each node is responsible for 177,000 URLs on average. Besides the initial snapshot, we expect that we will have to store a significant number of page changes for each URL over time. Since no commonly used file system can accommodate such a large number of files in a single directory, we decided to use a hierarchical directory structure. In the main download directory, Herodotus automatically creates 256 subdirectories with the names 00, 01, ..., FF, representing the last byte of the SHA1 value of the corresponding URL. Within each

subdirectory, there are an additional 256 subdirectories with the same naming scheme, corresponding to the second last byte of the SHA1 value of the URL. Since we will have $256^2 = 65,536$ directories to store data, the number of entries in each directory will remain small enough so that a traditional file system will have no problem with it.

File names are structured as follows. They consist of the SHA1 value of the URL concatenated with the SHA1 value of the content concatenated with the day on which it was downloaded, in Unix time (divided by the number of seconds per day). This choice of filename allows Herodotus to determine the download timestamp, as well as whether or not a file has changed or not, by simply looking at the filename in the directory, without even having to open the file.

Each file contains a one-line header with the actual URL downloaded for completeness.

4.5.2 Queue

The Queue contains the list of all elements that still need to be downloaded. As a Herodotus node receives new links from other Herodotus nodes, it not only puts those URLs into its in-memory data structures, but also stores them on disk. This is achieved by creating text files in a separate queue directory. The queue is split into separate files, each consisting of 1,000 elements, and a special pointer file indicates the current head of the queue. When new links are created, they are appended to the most recent file, and the file is stored on disk. As links are being dequeued, the pointer file is updated to reflect the current head of the queue.

In the event of a crash or reboot, Herodotus analyzes the queue directory on restart. It uses all queue entries to seed the URL-seen module, and the special pointer file to continue crawling at the correct queue position.

4.6 Replication

While the previous section described how we achieve persistence on one node, we have not yet addressed the issue of nodes joining and leaving the network. In chapter 3, we have identified that it is crucial for an archive to retain all stored content over long periods of

time. Especially in a peer-to-peer environment with a frequently changing set of active nodes, it is important to store data redundantly to achieve fault tolerance.

If nodes in Chord fail, their URL space automatically falls to their successor. As such, it is natural to keep replicated content on the successors of the respective node that is primarily responsible for the content. If the node with the primary responsibility fails, its successor with the replicated data will be addressed through the Chord lookup function and also be able to serve requests for the data.

Once a node permanently disappears, its successors holding replicas have the responsibility of creating a new replica to their successor to achieve the same level of replication as before. The number of replicas that should be kept is highly dependent on the failure/uptime characteristics of the participating peers, and how long they remain in the network on average. For typical peer-to-peer assumptions, we have found 6 to be a good level of replication (i.e. one node that is primarily responsible, plus 5 backup copies on its successors). In chapter 6, we will provide the assumptions underlying this choice.

In order to achieve this level of replication, each node effectively has to hold the data of the URL space corresponding to six different nodes on the Chord ring. There are two ways in which backup copies could be created: peers could open connections to each other and replicate the content, or the data could be fetched from the origin servers multiple times.

We feel that the latter solution is more appropriate, because it reduces the intra-node communication, and in fact the total bandwidth usage of Herodotus (because the content provider now has to provide the data to Herodotus six times). Consequently, we adapt our ring-based forwarding scheme in such a way that each link is forwarded not only to the node responsible for it, but also to its 5 successors. This can be achieved very efficiently by maintaining a rotating array of pointers to linked lists containing all the links for the respective previous nodes. Sending links to 6 nodes instead of one also reduces the probability that links could potentially get lost due to RPC errors.

We have shown in the previous section that if nodes go down temporarily, they correctly maintain all state. However, they might have missed URLs sent by other nodes during the downtime. Therefore, nodes periodically (once a day) compare the set of downloaded data with that of their successor and that of their predecessor. Since this could result in poten-

tially large data transfers, we have chosen to use TCP sockets instead of RPC function calls to achieve this replication. Every node opens a connection to its successor on a regular basis. When communicating with its successor or predecessor, the node determines which of its URLs its peer should also have (about 5/6 of the URLs it has stored). The two peers then compare this set of URLs using the directory structure described in the previous section. For each directory, they calculate the SHA1 value of the list of (SHA1 value of URL, timestamp) pairs of all files contained in that directory. As described above, timestamps are Unix times divided by the number of seconds per day, so downloads that occurred at different times on the same day will have the same timestamp. If those SHA1 checksums match, the directory contents are equivalent. If not, they transmit all directory entries so that they can determine the difference, and then request the corresponding file that is missing. The reason why only not the entire filename, but only the first two components of the filename (SHA1 of URL, timestamp) are being used is because two peers might have downloaded the same URL on a given day, but have received different content. In that case, we do not care which SHA1 checksum the downloaded content has.

If new nodes join, the same replication algorithm is used to restore all the information that they are responsible for maintaining. Since some data that had been stored on 6 nodes is now stored on 7 nodes, the replication protocol is also designed to delete the files on the node that is no longer responsible for them.

If nodes leave permanently, each of its six successors needs to assume additional storage responsibilities. The above replication scheme will automatically restore these conditions.

4.7 Optimizations to reduce bandwidth usage

In table 3.2, we have determined a total number of 3.54B web objects. Since the goal of Herodotus is to capture changes of the web on a daily basis, we have to query each of those 3.54B URLs every day. A naive solution would simply start at some well known URL like `www.yahoo.com`, and continue with the download and link distribution scheme described above. However, this would involve downloading all web objects and communicating all links between the peers every day. In this section, we will mention two improve-

ments that significantly reduce bandwidth usage. Those improvements are related to how objects are downloaded, how objects are stored, and how links are distributed.

4.7.1 Object download

Since each peer querying a web server for changes of a given URL is also the node responsible for maintaining the archived version of that URL, the use of conditional GETs greatly reduces bandwidth usage. In conditional GETs, we specified the date of the last download in the If-Modified-Since HTTP header. If the object has not changed, the server simply responds with a 304 (Not modified), otherwise, it returns a normal 200 (OK) with the updated content. For URLs for which no previous version has been archived, traditional GETs are used.

4.7.2 Object storage

As we have mentioned above, filenames contain the SHA1 content hash of each URL. When a supposedly modified web object has been downloaded, Herodotus first calculates the SHA1 content hash and compares it against the most recent archived version. If they match, the web server apparently responded to the conditional GET incorrectly, and the new version is simply discarded.

4.7.3 Link distribution

The description above suggests that a node only downloads so many objects because it receives so many links from its peers (which were extracted from HTML pages). However, since the same nodes are responsible for the same set of URLs every day, it is wasteful to send these links over the network over and over again. Instead, nodes only send the links contained in those HTML documents that have actually changed since the last download to their peers, i.e. if both the conditional GET returned a 200 and the content hash does not match the previous version. This method can be improved even further by the following observation. Even if pages change, a huge fraction of the links on the new page also appear on the old page. But we know that the responsible nodes already know about all the links

on the old page. Therefore, we should send only those links on the new version of the page that do not appear on the old page.

As a consequence of this scheme, nodes not only have to crawl the URLs they receive from their peers every day, but also all the URLs they have ever seen. We will describe this in more detail in the next section.

4.8 Daily crawls

Many aspects of this design suggest that the system would crawl the web only once, especially since the URL-seen test discards pages that have already been downloaded. In order to cause Herodotus to download all pages on a daily basis, the URL-seen test has to be "reset" every day.

In addition, if we use the improvements in link distribution described in the previous section, once a new day starts, we also have to initialize the queue with all the URLs we have ever seen.

Both of these goals can be achieved very easily considering the way the queue is represented on disk. The special pointer file points to the currently active portion of the queue on disk. If we simply reset this pointer to the first queue element, and then reinitialize the queue the way we would when Herodotus restarts, all URLs that have ever been in the queue will now constitute the active queue, and also be registered for the URL-seen test, to avoid multiple downloads of the same URL.

4.9 User interface

While the parts of the design described so far crawl and archive the web in a reliable and fault-tolerant fashion, we have not yet presented a way for users interested in the archived versions of URLs to actually access and view them. This last section describes how our design enables users to interactively view contents of the archive.

Each Herodotus node listens on a designated port (the port of the Chord service plus a constant offset). On this port, it accepts HTTP connections. By opening the root page (/)

on any Herodotus node, the user will be presented with a start page with a URL submission form. Once the user submits the form, the same Herodotus node that provided the HTML page will process the request. If the entered URL turns out not to reside on that Herodotus node, the node determines the IP address and HTTP port of the node responsible for the URL (by using a Chord lookup), and sends a HTTP response 301 (Moved permanently), which will cause the web server to query the correct Herodotus node for the requested object. The Herodotus server responsible for the node will then find all entries that match the URL in its on-disk file storage, and return an HTML page listing all the dates for which changes of the page have been recorded. These entries have hyperlinks that allow the user to request the respective versions of the URL. Once an archived version of a page is requested, all embedded references to other HTTP objects are automatically rewritten so that they represent correct Herodotus URLs for the corresponding archived version. As such, they contain a timestamp. When the links of an HTML page are rewritten, the new links will have the same date as the HTML page version requested. Once the user follows a link (either manually by clicking on it, or automatically in the case of embedded graphics), the corresponding Herodotus node will return the newest object in the archive that is not newer than the specified date.

Chapter 5

Implementation

In this chapter, we will discuss our implementation of Herodotus. First, we will give an overview of our implementation. We will conclude by giving the current status of our implementation.

5.1 Overview

Herodotus has been implemented in 5,000 lines of C++ code, not counting the Chord implementation which was linked as a library. The program has been integrated into the Location Service Daemon `lsd` of CFS [9]. As such, it is running as an additional service, DCrawl, on top of Chord, just like DHash. Figure 5-1 depicts this relationship.

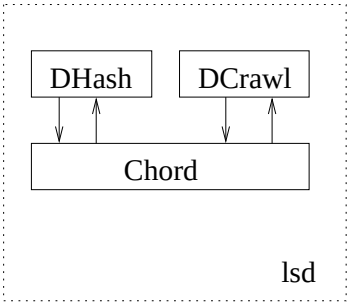

Figure 5-1: Integration of the Herodotus Service DCrawl into `lsd`.

Herodotus' Chord lookups and link distribution transactions are communicated over UDP using a C++ RPC package provided by the SFS toolkit [14]. Since for Chord lookups

and link distribution, each node sends short messages to many different nodes in a short period of time, UDP is better suited than TCP because it does not have the overhead for connection setup. Since replication only occurs between two adjacent peers on the Chord ring, and those sessions involve potentially large amounts of data to be exchanged, replication occurs over TCP connections. Since the downloads from the origin servers and the user interface transactions are HTTP connections, they must be done via TCP as well. Herodotus maintains a maximum of 56 simultaneous TCP connections to origin servers. While the number of concurrent connections could be raised, the memory required to store all the partially completed downloads is significant. For this reason, we decided to limit the number of concurrent downloads to 56. Internally, the program uses asynchronous I/O and callbacks, rather than separate threads. Herodotus has only been tested on FreeBSD, which is the platform that Chord and CFS have been designed for as well.

5.2 Status

Herodotus is currently fairly stable and can crawl the 1.4M web objects of the MIT domain on a set of four machines within a few hours without any problems. The crash recovery and replication services, as well as the user interface, have been tested on this small scale, and they seem to work well. However, we have little information about how Herodotus behaves in larger settings. While Herodotus has been designed to scale well to tens of thousands of peers, we did not have the resources and time to run Herodotus in these larger scale settings. Unlike with Chord, simulations are not very useful. In those simulations, hundreds or thousands of nodes typically run on one single machine. However, in such settings, a node does not see the typical traffic volume that it would if running on a dedicated machine (due to the hardware limitations and the high number of concurrent nodes on one machine), which make the behavior and internode communication patterns very different.

While Herodotus works well for our MIT experiments, it is still in a prototype stage. Before it could be deployed in an actual production system and run over many years, Herodotus should maybe be made more robust. In addition, system administrators should be allowed to configure the many parameters in a manner that is more comfortable than

changing the source code.

Chapter 6

Analysis

In this chapter, we analyze how feasible it is to deploy an actual system based on the design and implementation described in the previous chapters. In order to do this, we proceed as follows. We will relate the numbers obtained in chapter 3 to the properties of Herodotus to determine the number of participating peer-to-peer nodes necessary, as well as bandwidth, storage, and uptime requirements. Since this analysis shows that a substantial number of nodes is necessary, we will then present a "light" version of Herodotus that could be deployed with a much smaller number of nodes. Finally, we will describe how nodes could be recruited for a system like Herodotus, and we will in particular take a look at how suitable Gnutella nodes [1] are for running Herodotus.

6.1 Deploying Herodotus

In this section, we will relate the numbers of chapter 3 to the characteristics of Herodotus. First, we will estimate a required minimum number of Herodotus nodes based on total storage capacity and an assumed average contribution of storage space of each node. From that lower bound on the number of nodes, we will calculate estimates for the bandwidth requirements of each node. Next, we will address requirements for the uptime of each node. In the final part, we will summarize the requirements that we have derived.

6.1.1 Number of nodes

Chapter 3 shows that the required storage capacity is 56 TB for the first crawl plus 17.5 TB per month for changes. For the first nine months of operation, we therefore expect to store about 213.5 TB of data. Since web content is replicated to the five successors of each node for fault tolerance, the total storage volume comes out to 1281 TB. Assuming that machines contribute 100 GB per virtual node they run, we expect to need at least 12,810 nodes. Because we want to be conservative and not risk running out of disk space, we will set the requirement to 20,000 nodes for the first nine months.

It is hard to predict how well the growth numbers of chapter 3 might change after 9 months. In addition, we can expect the participating nodes to upgrade their available hard disk and bandwidth according to Moore's Law, with a doubling every 12 to 18 months. Therefore, we might assume that an initial set of 20,000 nodes should be sufficient for a few years. Statistics obtained during the operation of Herodotus should give more meaningful numbers and indicate when additional machines are required.

6.1.2 Bandwidth requirements

Using a set of 20,000 nodes and five replicated copies of each object for fault tolerance, each node is responsible for processing 6/20,000 of the daily crawl volume. In section 3, we have established a total number of 3.54 billion web pages and images, and a daily volume of data that has changed of 1.6 TB (uncompressed). Therefore, each node has to issue 1,062,000 conditional GETs, and is expected to download 480 MB of uncompressed new web content per day. Our MIT experiments show that URLs have an average length of 59 characters. Therefore, each conditional GET uses less than 100 Bytes of network traffic in total. Of these 100 Bytes, 80 are upstream and 20 are downstream. This means that all conditional GETs for each node combined amount to about 85 MB upstream and 21 MB downstream. Therefore, the total bandwidth for the daily downloads amounts to roughly 85 MB upstream and 501 MB downstream. These numbers correspond to 8 kBit/s upstream and 46 kBit/s downstream when amortized over 24 hours.

In addition to the communication between nodes and web servers, we also have to

account for communication between nodes for exchanging links and to achieve replication. Chapter 3 shows that an average web page contains 36 embedded references to other web objects. In addition, we found that an average URL is 59 characters long. Our ring-based forwarding scheme ensures that the aggregate bandwidth cost of shipping a URL is only slightly above the length of the URL. Shipping all the URLs contained in downloaded HTML documents to the six nodes (because of replication) responsible for them therefore amounts to a total of 12,744 bytes. This figure is roughly equal to the size of an HTML page. Table 3.2 shows that of the daily download volume, 75% corresponds to updated HTML, 7.5% to added HTML, and 17.5% to new images. In the design section, we have explained that for updated HTML, we only ship the new links that did not appear on the old version. Our MIT data shows that no more than 10% of the links of updated HTML are new. Consequently, we have to ship a "full set" of links for 15% of the downloads. Therefore, the cost for sending links is 15% of the 46 kBit/s rate at which we receive downloaded content, i.e. 7 kBit/s. Since each node sends and also receives links, the corresponding downstream bandwidth is also 7 kBit/s.

Therefore, the overall bandwidth requirement for crawling, i.e. for downloads and link exchanges, is roughly 15 kBit/s upstream and 53 kBit/s downstream.

The other component of bandwidth usage is data that is sent between nodes during replication. We assume that each node stores 100 GB. If a new node joins the network, it needs to download 100 GB from neighboring nodes. Even if this is done over a period of time of 60 days, the downstream bandwidth amounts to 150 kBit/s over that 60 day period. This data is provided from both its successor and its predecessor, which means that each of them has to provide roughly 75 kBit/s upstream.

If we assume that nodes stay up long enough that each node has to deal with a join or a permanent leave every 60 days, the total bandwidth requirement is roughly 90 kBit/s upstream and 203 kBit/s downstream.

Since we have 20,000 nodes, the total downstream bandwidth used by Herodotus is 4 GBit/s. In the Introduction, we estimated that the Wayback Machine needs at least 31 MBit/s. This shows that Herodotus has a total bandwidth 132 times as large as that of a centralized system. The reason for this much higher total bandwidth usage is replication

and the overhead to exchange URLs. We need to download every object six times instead of once. In addition, our conservative assumptions about the average participation times of nodes result in an immense amount of network traffic to maintain fault-tolerance.

6.1.3 Uptime requirements

The previous section has shown that replication for nodes that join the network or fail permanently is very costly. We have established that nodes should have to deal with at most one join/permanent leave per 60 day period so that the cost of replication does not grow beyond reasonable bounds.

Since during replication, the two adjacent nodes provide data, for each set of three nodes, we want to have at most one permanent failure/new join that needs to be restored for each 60 day period. If we assume that nodes join and permanently leave the network at the same rates to maintain the network size of 20,000 nodes, nodes should therefore stay up for a period of one year on average. This is because when one node leaves permanently and is replaced by a new node, replication has to occur at two places: at the location on the ring where the old node disappeared, and at the location where the new node joined.

If a node leaves and the content that it needs to store is being replicated over a 60 day period, the probability that all the other five nodes that have a certain piece of information fail within those 60 days is $(60/360)^5 = 0.013\%$. Therefore, with the given degree of replication and the given uptime requirements on nodes, only a minimal fraction of the archived data gets lost.

Besides permanently leaving the ring, nodes can also be temporarily down, e.g. for reboot. If we want each URL to be available with a 99.97% probability, nodes should be up 80% of the time (since for an uptime of 80%, the probability of all six nodes that keep copies of a URL failing is .3%). The main reason why uptimes should be at least 80% however is replication. If nodes are down for too long, the need for replication to bring them up to date rises, as well as their aggregate bandwidth usage due to the long downtime.

6.1.4 Summary of the requirements

In this subsection, we will summarize the requirements that we have established in the previous subsections.

We assumed that each virtual node has to contribute 100 GB of disk space. Bandwidth requirements for each virtual node are 90 kBit/s upstream and 203 kBit/s downstream. Each node should be participating for 360 days on average, and during that time period, have an uptime of 80%.

These bandwidth requirements are typical for home broadband Internet connections. Corporations or universities typically have huge multiples of these bandwidths. Therefore, it would be perfectly reasonable for individuals whose broadband connection is idle most of the time to contribute their resources to such a project. Universities, other research institutions, or corporations with their higher available bandwidths could provide an even larger number of nodes.

This analysis shows that the limiting factor is storage capacity, not bandwidth. While 100 GB is a significant amount of storage for a volunteer to contribute, the bandwidth rates that we have found are very commonplace today.

6.2 Herodotus light

The previous section has shown that 20,000 nodes are required for a Herodotus deployment to be feasible. Actually recruiting this huge number of machines seems very challenging. In order to get Herodotus up and running on a smaller scale, we will examine in this section the requirements if we decide to only store HTML, but no images.

Given a contribution of 100 GB of storage space per node, we have identified in the previous section that bandwidth is not really the issue. This doesn't change much if we limit the download to HTML data, since images accounted for only 17.5% of the daily download volume. Therefore, the cost for having to ship more links because we download relatively more HTML increases by no more than 20%. The bandwidth requirements for replication are identical.

If we decide to store only HTML, we need 6 TB for the initial download and an addi-

tional 10 TB per month. Over a nine month period, this amounts to a total of 96 TB. With a replication level of 6, the required storage capacity is 576 TB. With a storage capacity of 100 GB per node, this means that we need on the order of at least 6,000 nodes.

While this number is lower than 20,000 nodes, it is still high. If on the other hand, we could ensure that the nodes we recruit are more reliable than we have assumed in the previous section, a replication level of 3 might be sufficient. In that case, 3,000 nodes would do the job. Especially in controlled environments like universities, we can probably assume much higher uptimes and average participation times. In this context, the recommendation of 3,000 nodes seems realistic.

6.3 Recruiting nodes

In this section, we discuss how nodes could be recruited for participating in the Herodotus system.

The previous sections have shown that each node must contribute a significant amount of disk space, on the order of 100 GB. If this criteria is not satisfied, a node can contribute only very little to the overall Herodotus effort. In addition, we have identified that bandwidth requirements are not too outrageous, but rather moderate given today's broadband and LAN Internet connections.

The only other crucial criteria are uptime and the average time that a node contributes to Herodotus. Our analysis has made clear that nodes which participate for less than at least a few months cause more work for other nodes to achieve replication than they actually contribute to Herodotus. As such, when recruiting nodes, it is important that we have a good idea of the availability characteristics of the new nodes. One possibility includes not really using new nodes for the production Herodotus system yet, but just monitoring their uptime over a period of time. Only those nodes that pass this screening may then become part of Herodotus.

While a set of nodes deployed in controlled environments like universities could easily meet these criteria, we also want to look at how suitable Gnutella nodes would be for Herodotus.

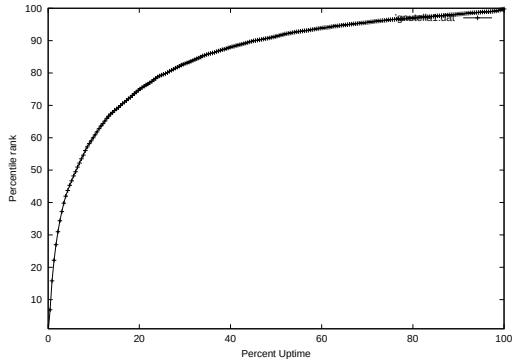

Figure 6-1: Percentile rank of Gnutella nodes with different uptime levels

Saroiu et. al have conducted an extensive measurement study of the peer-to-peer systems Gnutella and Napster [15]. They find that of a given set of Gnutella nodes, roughly 93% have disappeared after 10 hours. They did not perform any long-term tracking. In addition, the nature of the Gnutella has changed dramatically since late 2001. Limewire has made numerous attempts to make Gnutella more stable and scalable through the introduction of Ultrapeers, better routing schemes, and a less aggressive propagation of queries [4]. This has caused a number of music file sharing systems such as Morpheus to use Gnutella instead of proprietary protocols.

In order to obtain more current numbers on the uptime of nodes over longer time periods, we connected to the Gnutella network and gathered the IP addresses of 13,122 nodes that were connected to the Gnutella network at that time (which represents only a subset of all the nodes online at that point in time). These IP addresses were obtained by connecting to a small number of nodes that were known to us, issuing Ping messages, and gathering the replies. We then continued to iteratively connect to all those nodes, asking them for nodes they knew about, and so forth, until we had 13,122 unique IPs and ports.

Next, we attempted to open Gnutella connections to each of those 13,122 IPs every hour over the course of a ten day period. Figure 6-1 shows the percent uptime plotted over the percentile rank of Gnutella nodes that fall into that category. The graph shows that only 3% of the Gnutella nodes have an uptime level that is greater than 80%. Therefore, given the uptime levels of average Gnutella nodes, only a small subset could satisfy the requirements that we imposed in the previous section.

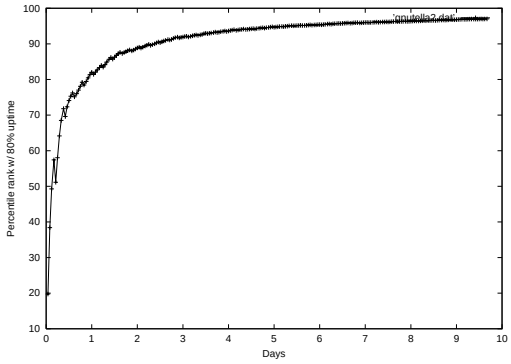

Figure 6-2: Percentile rank of Gnutella nodes with 80% uptime level over time (in days)

Figure 6-2 shows how the percentile of nodes satisfying an 80% uptime level has changed over the course of the ten days.

It is hard to predict what this graph would look like for time ranges of a few weeks or months.

For May 8th, Limewire reported an estimated total size for the Gnutella network of 250,000 nodes, 40,000 of which accepted incoming connections [5]. Since hosts that are not accepting incoming connections are typically low-bandwidth clients on dial-up connections that only download files, we should really only consider the 40,000 nodes accepting connections as reliable peers that could satisfy the bandwidth requirements. If we assume that 3% of the nodes satisfy our requirements, this would mean that we could use only 1,200 of the Gnutella peers on a given day.

Many assumptions are unclear. While these 1,200 nodes would have an expected uptime of at least 80% over the next two weeks, we do not have enough data to make long-term predictions about their availability past that period.

Another issue of Gnutella uptime data is the motivation of the end users. In Gnutella, the goal of a node is not to achieve a high uptime. It is only important that a large number of nodes is up at any given point in time. Therefore, most people do not really care whether or not their Gnutella client is running. In Herodotus however, people would have to be told to keep Herodotus up and running all the time. This is similar to Seti At Home [6], which also tells its members to use their system whenever their PC is idle. Seti At Home uses online rankings of their busiest members as an incentive for people to actually contribute as much

resources as possible. In addition, the Seti At Home website [6] shows that a large portion of people who signed up for the service a year ago are still active members. Herodotus would have to create a similar level of user awareness for the importance of high uptime levels to enhance the quality of the participating nodes.

Chapter 7

Conclusion

In this thesis, we have have presented the design and implementation of Herodotus, a distributed peer-to-peer web archival system. First, we used empirical data from crawls of the MIT domain to gauge the characteristics of the task of archiving the entire HTML and image contents of the World Wide Web. From those characteristics, we have derived a set of requirements. In the main section, we have described a design that adequately addresses the issues involved in a peer-to-peer archival system. We have implemented Herodotus and tested it on a very small scale, namely for archiving MIT's web pages over a few days on a set of four machines. Finally, we analyzed how Herodotus could be used to actually archive the entire web, and found that roughly 20,000 machines are needed initially. We have defined storage and bandwidth requirements for each participating node. Since that number of peers seems very high, we have also come up with an HTML-only solution that assumes more robust peers. For this scenario, a few thousand peers would suffice. Due to the political and administrative effort involved in recruiting such a large number of nodes, we did not deploy Herodotus on a large scale production system.

We have provided meaningful insights into understanding the complexity of archiving the web, and have designed a scalable solution that is based on Chord. Future work could refine our design. In particular, one could improve the way content is stored (using diffs to store only slight differences between two versions of a web site more efficiently). While our current implementation is a robust proof of concept, it needs to be improved to become ready for a large deployment. Finally, actually recruiting peer-to-peer nodes and then

deploying and running Herodotus on them constitutes an interesting challenge. It would provide interesting data that would show whether our design actually scales as well as we anticipate, and whether it adequately solves the difficult problem of archiving the web in a distributed fashion.

Bibliography

[1] Gnutella. gnutella.wego.com, 2002.

[2] Google. www.google.com, 2002.

[3] Internet Archive. The Wayback Machine. www.archive.org, 2002.

[4] Limewire Extensions to Gnutella. www.limewire.com/index.jsp/tech_papers, 2002.

[5] Limewire Statistics. www.limewire.com/index.jps/size, 2002.

[6] Seti At Home. setiathome.ssl.berkeley.edu, 2002.

[7] Krishna Bharat, Bay-Wei Chang, Monika Henzinger, and Matthias Ruhl. Who links to whom: Mining linkage between web sites. In *IEEE International Conference on Data Mining (ICDM '01)*, San Jose, California, November 2001.

[8] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[9] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[10] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey C. Mogul. Rate of change and other metrics: a live study of the world wide web. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[11] Brian Brewington George. How dynamic is the web? estimating the information highway speed limit.

[12] Allan Heydon and Marc Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, 1999.

[13] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[14] David Mazières. A toolkit for user-level file systems. In *Proc. Usenix Technical Conference*, pages 261–274, June 2001.

[15] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, January 2002.

[16] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.