Abstract

Internet's continuous growth have made it harder and harder for small businesses and non-profit organizations to find solutions for surviving a flash crowd. To use a Content Delivery Network (CDN) is usually not an option since it is simply too costly. This also includes solutions for replicating their content in a cost-efficient way. In recent years extensive research has been done in the field of Distributed Hash Tables (DHT), producing structured overlay networks, that have features that can be used by many applications. In this Master Thesis we present DOH, DKS Organized Hosting, a Content Distribution Peer-to-Peer Network (CDP2PN), where distributed web server's cooperatively will work together to share their load. DKS, which is a DHT, will provide the system with the well balanced placement and replication features that is needed in any CDN, and application level request routing will be used for load-balancing the DOH nodes. DOH will provide the same features as corporate CDNs to almost the same cost as a regular low-end web server.

The main delivery of this project is a prototype implemented in Java, using DKS[4], the Jetty[27] web server, and a modified JavaFTP[11] server package. This prototype, along with system model, was used in the evaluation tests of the proposed CDN design.

Sammanfattning

Internets fortsatta utbredning har gjort det allt svårare för småföretagare och ideella organisationer att hitta lösningar för att överleva en så kallad ”flash crowd”. Användning av nätverk för att distribuera webbsidesinnehåll (CDN) är oftast inte ett alternativ, eftersom det är för dyrt. Detta inkluderar även lösningar för att replikera sina websidor på ett kostnadseffektivt sätt. De senaste åren har mycket forskning ägnats åt distribuerade hash-tabeller (DHT) vilket har resulterat i virtuella nätverksstrukturer med funktionalitet som är anpassade till många olika applikationer. I denna examensrapport presenteras DOH, DKS Organized Hosting, vilket är ett peer-to-peer CDN: ett nätverk av samverkande webbservrar arbetandes tillsammans för att dela på belastningen. DHT:n DKS ger systemet välbalanserad placering av innehållet och möjligheter till replikation, vilket är funktioner som varje CDN behöver. Tillsammans med DNS-baserad request routing, som används för att distribuera webserverbelastning, kommer detta att skapa ett CDN med samma funktionalitet som ett kommersiellt, men till en kostnad som motsvarar densamma som företaget hade för endast sin webserver innan.

Detta examensarbetes huvudsakliga utkomst är en prototyp implementerad i Java, vilken använder sig av DKS[4], web servern Jetty[27] och en modifierad variant av JavaFTP[11] server paketet. Denna prototyp, samt en tillika utvecklad systemmodell, används sen för att testa och utvärdera den föreslagna systemdesignen.

Contents

List of Figures

List of Tables

1 Introduction

In the past couple of years the peer-to-peer (P2P) community has grown rapidly, not to say avalanche-like. P2P applications such as Napster, Kazaa and Gnutella are well-known, and have had millions of users worldwide. The goal of the peer-to-peer community is to create distributed and decentralized solutions to solve problems such as single points of failure and those related to scalability.

The main focus of this report is defining and creating a pull based Content Delivery Network (CDN) on top of a structured P2P system. This also includes developing an architectural prototype implementing the basic functionality of such a system.

1.1 Motivation

Take the example of a small company that, for example, creates web site solutions. The company have a web server located on a 10 Mbit broadband line, which usually serves them well. One day a big news portal review their site and recommend it to the portal users. Since the site has become a "hot object" it will generate a huge amount of hits. Subsequently the company's server will not be able to cope with the strain and their bandwidth will be consumed, making the page unavailable.

The situation described above is called the flash crowd effect (also known as the SlashDot effect[1]), where a sudden increase in traffic makes whole web sites go down. One solution for surviving a flash crowd is that a company pays for joining a CDN. For now we use a general definition of a CDN taken from [35]: "A CDN is a network optimized to deliver specific content [...] Its purpose is to quickly give users the most current content in a highly available fashion." For small companies which do not have the need for a CDN on a daily basis, it may be considered cost-inefficient to pay for this kind of features.

Imagine instead that the company in the example above had been a peer in a peer-to-peer network designed to cooperatively divide load and replicate content between the participating web servers. Then they would have access to the same features as if they where a part of a CDN but without paying a third party.

1.2 Vision

The use of a peer-to-peer system that cooperatively divides load between web servers should be interesting for many small companies. The vision for this master thesis project is to provide technology making it possible for small businesses and non-profit organizations to obtain the same hosting services that have been available to big companies for years, at an affordable price. By using DKS(N,k,f)[4], a distributed hash table (DHT) mainly developed at KTH[33], this master thesis project aims at creating a Content Delivery Peer-to-Peer Network (CDP2PN) called DKS Organized Hosting (DOH). The idea is to combine

DKS with a web server and create DOH-nodes with both DHT and web server functionality, as seen in Figure 1.1.

1.3 Related Work

Many P2P systems, like [44], [52], [36], [46], and [58], have been used for creating DHTs. However, the reasons for choosing DKS(N,k,f) is manifold. Two of the main arguments for choosing DKS(N,k,f) is that it has local atomic joins and leaves. I.e. by serializing all joins and leaves DKS(N,k,f) guarantees that the DHT never will be in an inconsistent state. Furthermore, by using symmetric replication DKS(N,k,f) allows for concurrent lookups. I.e. the client using the DHT will get more then one result when doing a lookup. This speeds up the lookup-time, if only one response is needed, or could be used as a base of a voting protocol in the case that the layer on top of the DHT want to be sure that the retrieved object not has been tampered with. These are features that, to the author's knowledge, no other P2P overlay network provides.

Until recently Content Distribution Networks has been proprietary solutions owned by companies like e.g. Akamai[2, 21]. Akamai's solution uses DNS redirection to reroute client requests and is based on BGP (Border Gateway Protocol) information and detailed knowledge of the underlying network topology. This is not a viable solution for a non-corporate CDN, since this knowledge is not always easily obtained. RaDaR[43] uses a hierarchical Multiplexing service to redirect the clients and a Replication service to keep track of replicas. All incoming requests goes through the multiplexing service which distributes the requests by looking at server load. RaDar is an example of a non-P2P approach of solving the CDN problem. Other solutions like CoralCDN[25], Globule[40] and SCAN[16], all propose some type of peer-to-peer content delivery network to solve the same issues. The authors of Globule make the observation that local web space is cheap, and could be traded for non-local, creating replicas on different slaves over the world. Client requests are then routed to the master or one of these slaves using for example the number of hops between Autonomous Systems (AS) as a metric of proximity. The problem is however, that the negotiation for replication space is not handled automatically but needs to be done by a human administrator, whereas DOH is aimed at being completely autonomous. SCAN uses Tapestry[58] and has all the features of a CDN and is indeed a P2P system. One of the main goals of SCAN, however, is to keep the number of replicas at a minimum to reduce

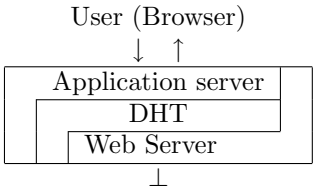

Figure 1.1: A high level view of a DOH node in the CDP2PN.

overhead. This will cause sites to be unavailable whenever the master copy is unavailable for some time. CoralCDN uses the Coral[26] implementation of a Distributed Sloppy Hash Table (DSHT) to keep pointers to the master (or valid caches) on different nodes. A round-trip time (RTT) clustering mechanism is used to exploit node proximity information and CoralCDN probes the closest cluster first for a copy. Redirection is done by using the systems own CoralDNS-servers which stores information about Coral-nodes. In CoralCDN, as in SCAN, if the master copy of a site becomes unavailable for Coral, the site will soon not be reachable for any user. In DOH however, when using DKS(N,k,f) and its symmetric replication, we make sure that sites always are available.

PAST[47] is a storage utility built on top of Pastry[46] and share many of the key ideas with the system presented in this thesis. Unfortunately as stated in [47]: "PAST does not provide facilities for searching, directory lookup, or key distribution." Where keys are needed to decrypt content. This makes it unusable from DOHs point of view in its current state, since searching for files is a key issue for the system to work as intended.

Open Content Network (OCN)[38] also is an effort to make a P2P CDN. OCN extends the HTTP protocol to create a Content-Adressable Web and, unlike DOH, it uses a browser plug-in for its clients to recognize it. The features of OCN is multiple parallel downloads of the same file and an ad hoc creation of a CDN. By ad hoc the developers mean that anyone that has installed the browser plug-in can help out to share the files recently downloaded, as is the case in e.g. the BitTorrent network[12]. The creators claim that OCN is a P2P CDN, however they seem to be aiming at P2P more then CDN: OCN is designed for handling large files and uses the users extensively to distribute files, as most existing P2P systems do.

DotSlash[59] is described by the authors as being a rescue system for web servers during hotspots but do share the same motivation as this thesis: to help web servers survive a flash crowd. DotSlash, however, does not store content globally but all servers will store their own content. When a flash crowd occurs, an overlay network with rescue servers will be created, and the "hot objects" will be cached at these servers during the flash crowd. This network will be abandoned when workloads are back to normal.

Some of these systems will be described in greater detail in Section 2.5, to see what can be learned before creating DOH.

2 Background

To be able to fully understand the problem addressed in this thesis, some background information may be needed. In this section we will introduce some technologies and concepts that are vital for understanding the rest of this document. First there will be an explanation about overlay networks and consistency models, then a discussion of DHTs in general and DKS(N,k,f) in particular. Then different request routing mechanisms are reviewed. Combined, these techniques

might lead to CDNs, which concludes this section.

2.1 Overlay Networks

Introducing new services into the Internet routing structure is problematic, slowing down progress. As a consequence overlay and peer-to-peer networks have been created for the introduction of new technologies, offered, not by network routers, but by end-systems and other new intermediates. Overlay networks are virtual communication structures that are logically "laid over" a physical network such as the Internet. Manually configured overlay networks is nothing new, e.g. [23] was published 1994, the new and exiting feature is that they are getting more and more self-organizing and nowadays handles changes and failures autonomously. Different overlays focuses on different issues, for instance: resilience[5]; security[34]; and semantics[18]. In this report the focus will be structured peer-to-peer overlay networks, implementing DHTs. The definition of "structure" is taken from DKS(N,k,f)[28]. The assumptions made are that the identifier space is discrete and defined as $I = \{0, \ldots, N-1\}$ for some large constant N ($N \in \mathbb{N}$). A set S is now defined, with a distance function $d : I \times I \rightarrow \mathbb{R}$ satisfying the following criteria:

1. d(x, y) > 0
2. d(x, y) = 0, iff x = y

We can now formally define a structured P2P system: A structured P2P system is a P2P system with a set S, where each peer in the system has got an identifier from the set and the choice of the neighbors of a peer is constrained by the distance function of that set. This simply means that: all nodes should have an ID from the identifier space; there should be an ID-based joining mechanism for joining nodes; there should be a way of measuring distance between node's IDs and finally that all nodes has distance 0 to themselves and themselves only.

Using this definition, and generating peer IDs from the same namespace as the keys, an overlay network can implement a DHT abstraction by deterministically mapping each key in the space to the peer with the numerically closest identifier.

2.2 Consistency

If a system only allow one copy of each object to exist or do not allow objects to change, consistency issues will not be a problem. Since such a system has grave limitations, a complete field of research exist concerning how to maintain consistency of replicated objects and which guarantees such a system can give its users. In this section the two main policies in this field will be introduced.

Strong consistency
If a system guarantees strong consistency it guarantees that no client of the system will ever access a stale, i.e. an old, object. In a highly dynamic environment with multiple, dispersed, replicas this is not easy to achieve and the

overhead might not be worth the gain. Therefore strong consistency seldom is used in wide-area systems, unless it is absolutely necessary.[1]

Weak consistency
Since strong consistency is hard to maintain and sometimes is not required, a weaker, more flexible, consistency model is introduced. In a system that guarantees weak consistency replicas may differ, but all updates are guaranteed to reach all replicas eventually (i.e. in a timely fashion). In wide-area systems with large RTTs, or in systems where updates do not have to be propagated instantaneously this consistency model might suffice.

Since all CDNs uses replicated objects, and allow them to change over time, they must have a consistency model. The overlay structure and the request routing mechanisms must be adjusted, or chosen, to fit that model. How to actually achieve this will be described later, when the CDNs are reviewed in more detail.

2.3 Distributed Hash Tables

A DHT is a self-organizing overlay network of nodes that supports lookup, insertion and deletion of hash keys. The core of a DHT protocol is a distributed lookup scheme that maps each hash key into a deterministic location with well balanced object placement and low overhead.

2.3.1 General DHT characteristics

When a node is joining the DHT overlay network it is assigned an ID from some namespace. Keep in mind the hash table terminology with `<key,value>`-pairs: then a `key` usually is hashed to the node with the closest ID, for some notion of closeness, and the `value` stored can either be actual data or merely a directory of pointers to several data storage locations. When a node is doing an insertion, `put(key, value)`, the key is hashed and the node then searches the network for the node with the closest ID. This node is responsible for storing the value. When a node is doing a retrieval, `get(key)`, it starts by doing the same thing: it hashes the key, searches for the closest node and retrieves the value from that node. It is not the case that all the nodes know all the other nodes, but rather that they have partial knowledge of the network to keep the routing information scalable. Internally, when executing the hash table operations, the DHT system will have to perform a lookup, `lookup(key)`, that returns the host identifier for the host that stores the key. There are several known DHT routing strategies and usually they provide O(log N) lookup time, where N is the number of nodes in the system. A degree of fault-tolerance, by using replication, is normally built-in in the DHT since the nature of P2P networks is somewhat unstable with nodes joining and leaving the system at a high rate.

The well balanced object placement, or load-balance, is given since each node is responsible for a chunk of the entire hash table and the hash function used should be producing uniformly distributed output. As described in [50]:

[1] See [49], p.30-33

"If the hash function used by the table is uniform, then regardless of the distribution of resource names stored, resources are distributed uniformly over the hash space. As long as the chunks of the hash space assigned to participating nodes are of roughly equal size, then each node maintains a roughly equal portion of all resources stored into the distributed hash table, thereby achieving load balancing."

The features that makes DHTs good candidates for building distributed applications are the following:

- decentralized
- self-organizing
- well balanced object placement
- scalable
- robust

2.3.2 DKS(N,k,f)

Distributed K-ary Search (DKS)[4], has three parameters N, k and f defining the system. N is the maximum number of nodes that can be present in the network, k is the search arity within the network and f is the degree of fault tolerance. These parameters are known by all nodes in the system, fixed and decided upon when creating the network. N, k and f are chosen such that $N = k^L$, where L is big enough to achieve a very large network, and due to the replication scheme used also that N mod f = 0.

The approach is based on two main ideas: the distributed k-ary search method and a novel technique called correction-on-use. The principle of DKS[2] is to be able to resolve a key identifier t, i.e. to find its corresponding value, in at most $\log_k(N)$ steps. This can be guaranteed by dividing the search space into k equal intervals in each step of the search. (This can be seen as a generalization of [52], where k = 2.) The difference between correction-on-change, which is used in e.g. [52] and correction-on-use, that is used in DKS is the following: correction-on-change uses periodic stabilization to correct routing entries when the network changes, while the correction-on-use technique basically corrects routing failures on the fly.
All the nodes in a DKS system has a unique identifier x∈I. The identifier space is denoted $I = \{0, 1, \ldots, N\text{-}1\}$ and is organized as a ring, see Figure 2.1. A value v with the associated key t is inserted at the first node met when moving clockwise in the identifier space starting at t. Therefore a node is responsible for storing all the elements with keys mapped to the identifier space between it and its predecessor p.

Routing in DKS
To guarantee that lookups can be resolved in at most $\log_k(N)$ hops, each node in the network organizes its routing table in different levels ($L = \log_k(N)$). At

[2]DKS will throughout the report be used as short term notation for DKS(N,k,f).

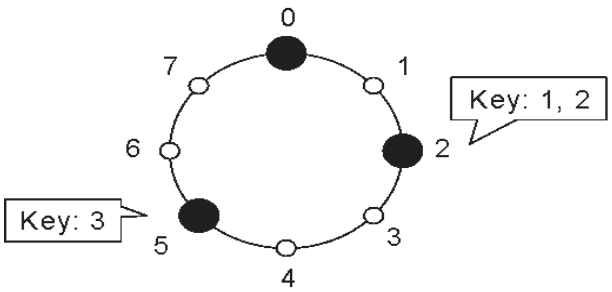

Figure 2.1: Shows the ring topology if the DKS(N,k,f) network. Nodes 0, 2 and 5 are present in the system. N is 8, k is set to 2 and the keys 1, 2 and 3 are inserted.

each level the node has a different view of the identifier space, i.e. it divides some part of the identifier space into k equal intervals and holds one node responsible for each interval. (See Table 2.1 for an example of a DKS routing table.) When moving towards higher levels the view is narrowed down until each interval corresponds to a single node. At the first hop the lookup is routed with information from level one and at the second hop with information from level two and so on. Generally lookups are forwarded to the node that at the current level is responsible for the identifier space that the lookup concerns. The lookup is forwarded until it is received at the node with ability to resolve the lookup locally. E.g. assume that a client request key 3 from node 0 in the DKS network described by Figure 2.1 and Table 2.1, then the following steps are performed by the system:

1. Node 0 starts off by checking if 3 is between itself and its predecessor 5, which it is not.
2. Node 0 looks in its routing table in level 1 for the node responsible for the interval that 3 is in, which is itself.
3. Node 0 looks in the second level of its routing table to see which node is responsible for that interval. It turns out to be node 5.
4. Node 0 sends a lookup request to node 5.
5. Node 5 checks if 3 is between itself and its predecessor 2, which it is.
6. Node 5 performs a local lookup and returns the value to the client or to node 0.

Replication

The replication in DKS is based on the assumption of symmetry, i.e. if node i stores replicate objects for node r, node r should store i's objects as well. This is achieved using the mathematical concept of equivalence classes. Using the f parameter, the DKS identifier space is divided into N $\div$ f equivalence classes, where all nodes in a class store each others objects. This gives the system f replicas of each object. Look again at the DKS network described in Figure

	Node ID: 0		Node ID: 2		Node ID: 5	
Level	Interval	Resp	Interval	Resp	Interval	Resp
1	[0,4[	0	[2,6[	2	[5,1[	5
	[4,0[	5	[6,2[	0	[1,5[	2
2	[0,2[	0	[2,4[	2	[5,7[	5
	[2,4[	5	[4,6[	0	[7,1[	2
3	[0,1[	0	[2,3[	2	[5,6[	5
	[1,2[	2	[3,4[	5	[6,7[	0
	Predecessor: 5		Predecessor: 0		Predecessor: 2	
	Keys:		Keys: 1,2		Keys: 3	

Table 2.1: The DKS routing tables for Figure 2.1 showing for each node on all levels which node is responsible for an interval. $N = 8$, $k = 2$ and $f = 1$ makes $L = 3$. Nodes 0, 2, 5 and keys 1, 2, 3 are currently in the system.

2.1. With f = 2, and assuming that all nodes are present, node 0 and node 4 would store each others keys, node 1 should store keys for node 5 and vice versa, etc. (But since not all nodes are present in this network, we end up with nodes storing keys as described in Table 2.2.) This means that f nodes in the system store each key and that those f nodes can reply to a request for that key. Since N and f is known by all nodes in the system, a node can calculate which nodes that are associated with each other and send a lookup to any of them when requesting an object. This advantage, that the system can share the lookup load internally, is one of the reasons for choosing DKS as the DHT to use in DOH.

Self-organization
The DKS system uses atomic leave and join operations along with correction-on-use to achieve self-organization, a quick overview of these techniques is found below.

Atomic leaves: When the node is leaving the network it should appear as if it is done in an atomic fashion, to make sure that no keys are lost. This is managed by letting the application layer on top of DKS transfer the node's state to its successor, meanwhile the node buffers all relevant incoming messages. When the state transition is finished the successor sends a message telling the node that it can leave. The node forwards all buffered messages to the successor and then leaves quietly. Since DKS has correction-on-use nodes trying to communicate with the node that left will find out that the node is gone and update their routing table information. If two leave requests interfere with each other, the

Node ID: 0	Node ID: 2	Node ID: 5
Keys: 3	Keys: 1,2	Keys: 3, 1, 2

Table 2.2: Keys stored in DKS network showed in Figure 2.1 when $f = 2$.

requests will be serialized and therefore the atomic property will still hold.

Atomic joins: First a component called pP is added to all nodes in the network which will be used only for making the insertion of nodes in the network atomic. At any given time, in node n, pP is either an address to a node that is being inserted or `nil` if no node is being inserted at that instant. There are two main cases when a node, n_j, is attempting to join a DKS network: if it is empty or non-empty.

Joining an empty DKS network
This is the base case, when n is the first node in the network. Thus n performs the following actions:

- Set all routing entries at all levels to its own address.

- Set its predecessor pointer to point to itself and pP to `nil`.

Joining a non-empty DKS network
To join a non-empty DKS network, the joining node, n_j, sends a join request to a known node, n, which already is in the network. The join request might be forwarded so that the new node can be inserted at its correct position. Then n calculates the new node's routing table. Very simplified the way that the routing table for n_j is calculated is as follows: First n makes n_j responsible for the interval closest to n_j on all levels. Then node n divides the area that it is responsible for in two parts. It sets itself as responsible for the values between n_j - n, n_j as responsible between n's predecessor p - n_j and the node that to its knowledge is n_j's successor on that level, responsible for the rest. After that, both nodes updates their predecessor pointers and n sets pP to `nil`, since no node is being inserted. When done, it sends over the routing table to n_j. The insertions are done in a local atomic fashion, i.e. that if two concurrent joins occurs at the same node, the node will serialize them.

Correction-on-use: This technique is built on two observations. First that by sending level and interval information along with lookup/insertion messages the receiving node, n, can derive whether the message came to the right node or not. (This is because of the made assumptions of N and k.) If this not is the case, n informs the sending node, m, that it has an error in its routing table, and points it in the right direction (i.e. n's predecessor). When m receives the error message it updates its routing table and tries to send the same request to the new node. The second observation is that whenever a node n receives a message from a node m, it can detect if it shall have m in its routing table. If that is the case it will update the routing table accordingly. E.g. assume that a new node 3 has joined the network described in Figure 2.1 making node 0's routing table in Table 2.1 erroneous. Now node 3 stores key 3, and since node 5 inserted node 3, node 5's routing table are up-to-date. As in the previous example node 0 makes a lookup request to node 5. Node 5 calculates which predecessor node 0 thinks it has, and sees that node 0 thinks that node 2 still is node 5's predecessor. It then sends an error message to node 0 and points it to node 3, node 0 updates its routing entry and queries node 3 instead. Node 3 responds by giving node 0 the value of key 3.

Some drawbacks of using DHTs for the purpose of creating CDNs exist, for instance that the same mechanism that provides the load-balancing (hashing) also destroys locality. This can be solved using multiple DHTs, creating several overlay networks as in [26], or by using multiple ID spaces and using the locality of DNS-names as in [30]. Another is the case when replication is done statically and replicas cannot be migrated towards big user groups without destroying the routing mechanism, but this limitation could be handled by using for example caching, or other mechanisms that will be described later when specific CDNs are reviewed.

2.4 Request Routing

In a CDN request routing normally is used to redirect clients to the best replica of a replicated object, where best is determined based on different policies and metrics. The steps involved when a client request is made to see e.g. a web document are shown in Figure 2.2. Steps A-D concerns translating the URL into a network address and steps E-F is the request-response dialog. (Arrows B and C are divided into two because the client's DNS server normally has to go through several intermediate servers before finding the server's DNS server.) [10] differentiates between three major request routing mechanisms: DNS based, transport layer and application layer request routing. DNS based modifies step C, transport layer modifies step F and application layer request routing modifies step F or might add a whole new request cycle.

But before looking on DNS based request routing, it is appropriate to do a review of the domain name system containing some terminology and other aspects that is important for this thesis.

DNS
The Domain Name System is the naming scheme used on the Internet. It provides features such as a more human friendly naming and translation between those names and IP addresses. Domain names are hierarchical, with the most significant part of the name on the right. The left-most segment of the name is the name of an individual computer. E.g. `foo.bar.com` where `com` is the top-level domain, `bar` a domain and `foo` the name of a computer. The mapping between a computers human name and its IP address is called a bind.

The DNS servers are arranged in a hierarchy that matches the naming hierarchy. A root server is on top of the hierarchy and is responsible for the top-level domains. Naturally one DNS server cannot contain all the DNS entries for a top-level domain, but rather it contains information about how to reach other servers. As in the example above, the root server does not know the IP address of `foo`, but it knows the address to the DNS server responsible for the `bar` domain, as explained below.

Resolving a name: The translation of a domain name into an equivalent IP address is called name resolution. The software used for doing this is usually referred to as a resolver and is a part of the operating system. Each resolver is configured with one or more local domain name server's IP addresses. The

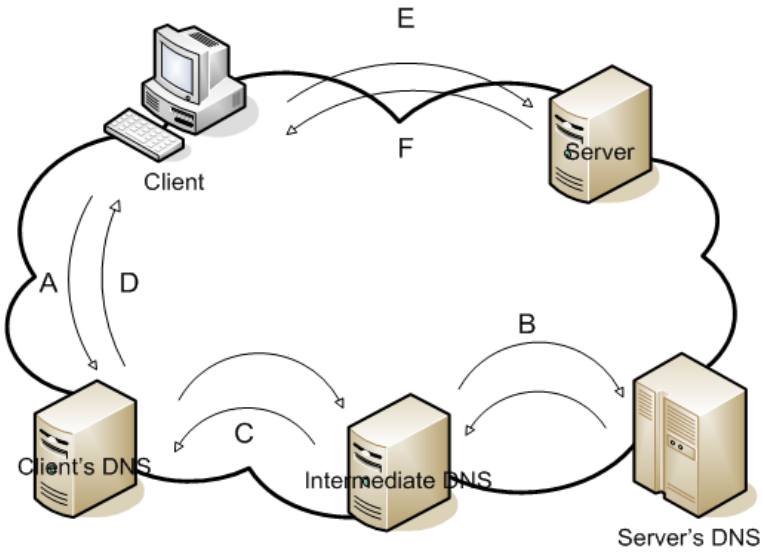

Figure 2.2: The steps involved in retrieving a client HTTP-request.

resolver sends a DNS request message containing the name to its local DNS server and starts waiting for a DNS response from the server. If the server is an authority for the name, it will do a local lookup and respond to the resolver. Otherwise it will have to ask one of the root servers on how to continue. The root server, which knows the top-level domain will point the querying server in the right direction and the query will proceed until an authority for the name is found. Take, once again, the example of `foo.bar.com`, then the resolving process is shown in Figure 2.3.

Each entry of the DNS database has three items: a domain name, a record type, and a value. The record type specifies how the value is to be interpreted. The three types that are relevant for this thesis are the following:

A the A record, where A stands for address, is used for the binding of a domain name to an IP address.

NS the authoritative name server for the domain.

CNAME the CNAME record is similar to a symbolic link in a file system. The entry provides an alias for another DNS entry. E.g. when a company might not want to name their web server www. Then a CNAME record could be used to redirect DNS requests for `www` to the actual web server.

This concludes the DNS review, using this information DNS based rerouting will be explained.

2.4.1 DNS based

Using their own augmented DNS servers somewhere in the resolution process, CDN systems often uses the DNS based request routing approach. The differ-

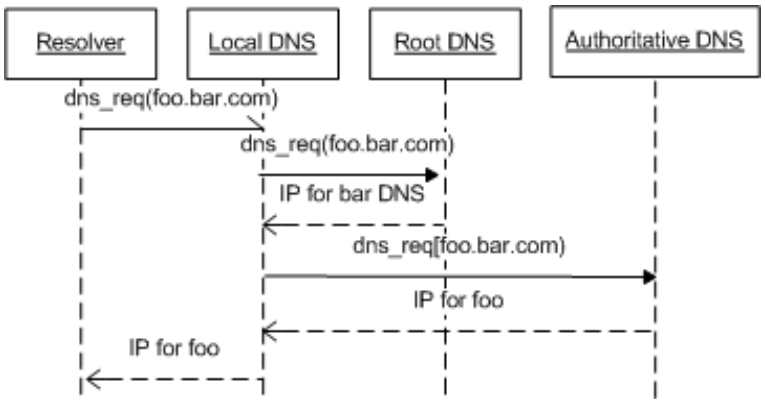

Figure 2.3: The DNS resolution process.

ence between regular DNS servers and the customized ones used by CDNs are the mechanisms implemented for choosing a reply. It is not uncommon that a site has several IP-addresses, and a round-robin strategy is then used in regular DNS servers to choose a machine when a request is made. When operating, regular DNS servers usually do not handle information about client-server proximity or server load metrics. This is what is added to the CDNs DNS servers: different policies for choosing which replica to direct clients to, using e.g. the client's IP and AS information and estimating server utilization to find close replicas on idle servers.

Single reply: The single reply technique can be used when the DNS server is authoritative for the whole domain or subdomain. It will return the IP address for the best replica server.

Multiple replies: Here several IP-addresses are returned and the local DNS server uses round-robin when replying to client requests. This technique could also be seen as a two level round-robin algorithm, since a number of web servers are given to the local DNS by the remote DNS in a round-robin fashion, then the local DNS distributes these web servers to clients, once again using round-robin.

Multi-level Resolution: In this approach multiple DNS servers can take part in a single resolution process. As described in [10]: "An example would be the case where a higher level DNS server operates within a territory, directing the DNS lookup to a more specific DNS server within that territory to provide a more accurate resolution." This resolution process will eventually end up at a server with a CNAME or a NS record. CNAMEs, or canonical names, can be used in a DNS server to give a host several domain names. This can also be used to redirect the client to a whole new domain, where the resolution process starts all over again. This could be seen as a bit slow, but imagine that one DNS server is getting to many requests, then this is a way to share the load. NS records could be used to transfer the authority from the domain to a sub-domain, which gives finer redirection granularity. Unfortunately the NS approach is limited by

the number of parts in the name, due to DNS policies. This approach can be combined with single or multiple replies.

Object Encoding: A useful technique is to code some information concerning document content in the URL, using this information to redirect clients. E.g. the URL to an image could be `image.url.com`, then requests for images will be redirected to the image server of `url.com`.

These are some of the DNS techniques currently used, and even though they are quite useful they have some drawbacks as well. The granularity of replication using DNS based request routing solutions are normally in the size of complete sites, since redirection is done using the domain or subdomain names. When 4-5% of that replication is enough to get comparable performance result, according to [15], this might be considered to costly. Every DNS lookup response comes with information on how long the answer is valid, a so called time-to-live value (TTL), and the answer and its TTL is cached along the lookup path. This is good in the normal case since any DNS with a valid entry will reply to a lookup and thus share the load and shorten lookup latency. But when redirecting requests, this is not a desired feature because you loose control over the redirection when other DNS servers reply instead of the system's customized ones: Since all DNS servers along the resolution path caches the result, as long as the entry is valid, the proceeding requests following that path will be redirected to the same replica and might choke that host. Small TTL values are used for handling this problem and to be able to quickly adapt the system to node changes. Unfortunately this much-used technique has two major drawbacks. When interpreted correctly it will produce more lookup traffic, congesting the network. Furthermore a lot of DNS servers rewrite the TTL to their own minimum if the received value is considered to small, thus destroying the purpose of the whole scheme. Another drawback is that there is nothing that says that a client's DNS server and the client itself is close to each other, making it tricky to do proximity calculations. All in all, DNS based redirection is a bit crude, but is very useful and combining different DNS based strategies along with transport or application layer request routing can be very attractive indeed.

2.4.2 Transport layer

Transport layer request routing, or TCP-handoff, is quite obviously implemented in the transport layer. Since more client information is available, such as e.g client IP addresses, finer granularity can be achieved. The first packet sent in the client request is examined and routing decisions are made using the IP, port and the application level protocol information along with system policies and metrics. In general the connection flow is divided so that client-to-replica-host packets are sent via the redirecting server and replica-host-to-client packets are sent directly to the client. This is ok, since the flow from the replica host to the client is assumed to be much larger. However doing this for many requests will still slow the redirecting server down and this solution therefore does not scale as well as the others. One other thing; when creating a large system using this type of redirection, a complete network with modified infrastructure have to be created and maintained. As stated earlier, one of the reasons that overlay

networks even began to be deployed in the first place was the problems arising when the need of this occurred.

2.4.3 Application layer

Application layer request routing has even more information available and therefore it can make even more fine-grained routing decisions, down to a level of individual objects. With the client's IP address and content request it should be able to determine which replica host that is the best for this request. There are two main approaches for doing these decisions and they will now be described briefly.

Header Inspection: There are several ways for application layer request routing to use the first packets sent to determine what the content of the request is and where to redirect it. HTTP[24] for example describe the content of the requested URL, and in many cases this is all the information needed to redirect the client properly. Redirection can be done by using the built-in functionality of HTTP as well. Using e.g. the 302-error message, "this page has moved temporarily", which can be used for redirecting clients to another replica in the system.

Content Modification: Using this technique a content provider can get full control in redirecting the client without the help of intermediate devices. Basically a content provider directly communicates to the client which replica that should be chosen and these decisions are made depending on system policies. The method explores the possibilities of the basic structure of HTML. I.e. that a page normally contains embedded objects and by modifying the references to them, either beforehand or on the fly, so each object can be fetched from the best replica. This technique is also known as URL-rewriting, since the URLs within a page is rewrited. A drawback is that the pages become hard to cache since the URLs used might not be valid any more. And also that the URL-rewriting, if done beforehand, requires that server load are quite stable since best replica is calculated in advance.

There is not one of these strategies that can be used in all systems for all purposes, which of course would have been nice. For achieving the goals of this thesis, where the system should be used by small companies, the DNS based single reply or multiple replies request routing combined with application based header inspection might be the way to go. Since smaller sites usually only have one DNS server that is responsible for the whole domain or sub-domain single reply DNS based request routing is applicable. But then again using the replication provided by DKS, the multiple reply technique could be a natural choice, slightly modified it actually implements a two-tier round-robin scheduling algorithm described in [17], which is proven to yield good performance results.

One thing that was not discussed is the transparency of the different strategies. From user point of view, and developer as well, it would be preferable if the rerouting was done in a transparent manner. In general it can be said that DNS based and transport layer request routing is transparent

while application layer is not. These different request routing mechanisms can be combined though, to obtain even better performance and as we will see later when looking at actual Replication systems (CDNs), they are.

2.5 Replication systems

Content delivery networks exists for mainly two reasons: First for companies that do not want to buy and maintain their own web-hosting structure and second to decrease user-perceived latency. It is the latter which is relevant for this work. Decreasing the latencies is usually done by combining two methods: replication (or caching) and by redirecting clients to servers "close" to them, where close can be defined using geographical, network topology or time metrics.

According to [49], there are five important issues when creating a web replica hosting system:

1. How do we select and estimate the metrics for taking replication decisions?
2. When do we replicate a given Web document?
3. Where do we place the replicas of a given document?
4. How do we ensure consistency of all replicas of the same document?
5. How do we route client requests to appropriate replicas?

Which combined with what to replicate, makes up a replication system. Let us look briefly at how some of the related systems solves these issues.

2.5.1 Akamai

In November 2003, according to [57], Akamai[21] had a commercially deployed infrastructure that contained more then 12000 servers, creating more then 1000 networks in 62 countries. Most of the servers are located in clusters on the edge of the Akamai topology and using their massive infrastructure, Akamai just allocates more servers to sites experiencing high load. The replication is mainly done by caching the web documents and where to replicate them is decided by three functions calculating a combined metric. The functions are Nearest, Available and Likely. Nearest is time (the smaller the better), Available is load and network bandwidth and Likely is which servers usually provide the customers object. Replicas are placed on edge servers, so the traffic within the Akamai network is kept to a minimum. Consistency is ensured by a versioning scheme that encodes version information in the document's names. All Akamai edge servers have assigned a DNS name. The client request rerouting is done by firstly using regular DNS-servers and then low-level Akamai DNS-servers that has knowledge of which of these edge servers that has valid copies of the requested object. With every response a TTL value is sent. The TTL usually is small to encourage frequent refreshes, which allows the Akamai system to reroute later requests from the same user. (See Figure 2.4.)

The fundamental approach for Akamai when creating its CDN, differs widely from ours. An Akamai client do not own any web hosting infrastructure

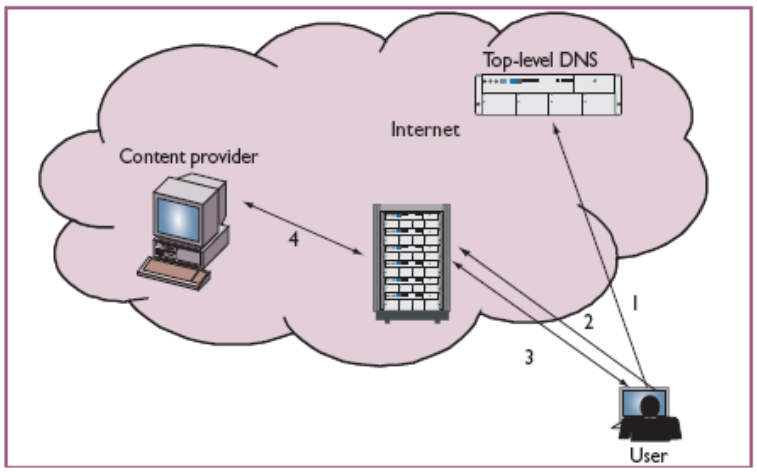

Figure 2.4: "Client HTTP content request. Once DNS resolves the edge server's name (steps 1 and 2), the client request is issued to the edge server (step 3), which then requests content (if necessary) from the content provider's server, satisfies the request, and logs its completion." ([21], Figure 1)

but instead buy the service from Akamai. Akamai has its own backbone and clusters content servers at the edge of this backbone to serve clients. So to use their approach for creating a CDP2PN is not an option.

2.5.2 RaDaR

RaDaR[43], as seen in Figure 2.5, uses a Multiplexing service (MUX) and a Replication service. The multiplexing service keeps mappings between physical names and symbolic names. When an object is created the host server registers the physical name at the MUX which assigns a symbolic name to be used by the external clients. All requests have to go through the MUX which fetches the physical object at one of host servers and returns it to the client. The decision on how many replicas there should be and where to place them are taken by the hosting servers, using access statistics. RaDaR also supports migration to move whole sites if the system detect that clients are far from the content. The Replication service keeps track of all the hosting servers on the network and the server loads and the system uses this information for choosing a host to migrate or replicate content to.

The multiplexing scheme provides the ability to use very advanced schemes for load-balancing and replication decisions but unfortunately it might become a bottleneck when dealing with large systems. Also the idea of peer-to-peer networks where all peers have the same capabilities is not very applicable to the RaDaR system.

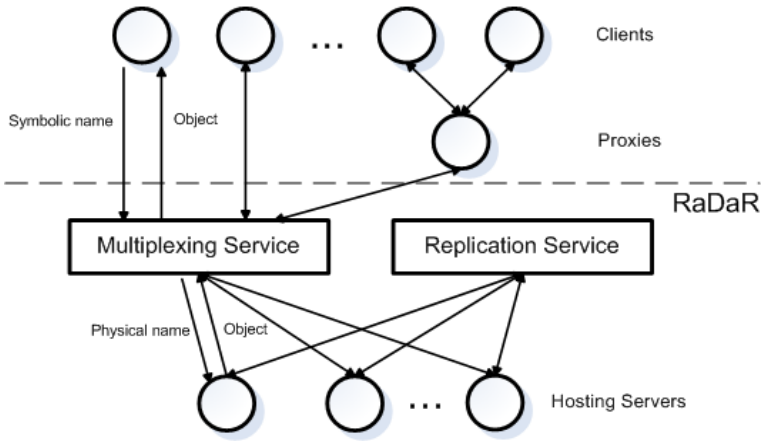

Figure 2.5: A high level view of RaDaR. ([43], Figure 2.)

2.5.3 SPREAD

SPREAD[45] is an example of using an entirely different approach. It is designed to be deployed in the network layer rather than the application layer as is the other CDNs described in this section. Client redirection is done by using a network packet-handoff mechanism which uses router hops as a metric, and IP-tunneling to route each packet along the same path. (Similar to the TCP-handoff mechanism described in Section 2.4.2) Replication is done by considering the network path between the edge server and the origin server and by determining which proxies to put replicas on along that path. So it is actually the routers who takes the replication decisions. Depending on documents characteristics, for instance its access pattern, strong consistency is maintained by using three different methods: Client validation (if-modified-since), where clients check if the replica is valid; server-invalidation, where servers send out invalidation messages and proxies get fresh copies when needed and finally replication, where new copies of documents is pushed to proxies when updated.

Obviously this is far from the approach taken in this thesis, but has its place here since it shows that CDNs can be implemented using the existing routing infrastructure. It also, and more importantly, shows how different methods to maintain consistency can be used for different documents, which most certainly could be applied to objects in the DOH network.

2.5.4 Globule

Globule[40] is implemented as a module to the popular web server Apache[6]. Replication in Globule is done on a per-document basis, where a document object contains two things: the actual content, a page and all its embedded objects, and a meta-object that stores information for making replication decisions and consistency checks. In Globule, replication decisions is described as minimizing a cost function over the nodes involved in storing an object. This is done by simulating different strategies based on recent client requests. The

minimum corresponds to the best replication strategy for that document. An overview of the replication strategies used and their performance can be found in [41]. Where to place replicas depends on available servers. The observation that the authors of Globule make, is that local space is cheap, compared to non-local. Therefore Globule server administrators negotiate with each other to exchange server resources, and the local server keeps track and limits the peers resource usage. The master copy is kept on your own local server and the slave replicas are kept at negotiated host servers. Client redirection is done by DNS-redirection using a customized DNS server, which is part of the Globule implementation, as described in [40]: "before sending an HTTP request, the client needs to resolve the DNS name of the service. The DNS request eventually reaches the authoritative server for that zone, which is configured to identify the location of the client and return the IP address of the replica closest to it."

The fundamental idea in Globule, to exchange local space for non-local, differs from the one in this thesis. Globule only replicates a part of the data to its slaves and if the master goes down, the slaves will not be to much use. In DOH all data should be replicated. Also DOH should be totally self-organizing compared to a Globule system, where administrators have to negotiate with each other for obtaining non-local space.

2.5.5 SCAN

SCAN[16] uses dynamic, adaptive replication and the system considers client latency and server load when deciding what to replicate. The goal of the system is to keep replicated objects to a minimum to decrease replication overhead. To achieve this there are two interlocking phases of the dynamic placement algorithm: replica search and replica placement. The replica search phase tries to find a replica that meets the client latency constraint on a server which is not overloaded. If no such replica is found the replica placement phase starts and a new replica will be created. SCAN creates multicast trees for keeping track of replicas and these trees are used for propagating updates. Therefore consistency is not strong, but updates are assumed to be propagated in a timely fashion as in the weak consistency model. SCAN is built upon Tapestry[58] and uses the Tapestry infrastructure that provides locality information. Tapestry has several defined root nodes and when a new replica is published that information is pushed towards a root node, where each node on the way stores a pointer to it. When a query is made it is also pushed towards the root node and the first node that knows the answer, i.e. the first node that has a pointer to a replica of the requested document, replys to it. (See Figure 2.6.) This guarantees that clients will be redirected to the closest replica of the document they are requesting.

The authors of SCAN assumes that SCAN servers are placed in Internet data centers of major ISPs with good connectivity, direct connected to the backbone. This approach limits the participants of a peer-to-peer network severely and can not be used in DOH.

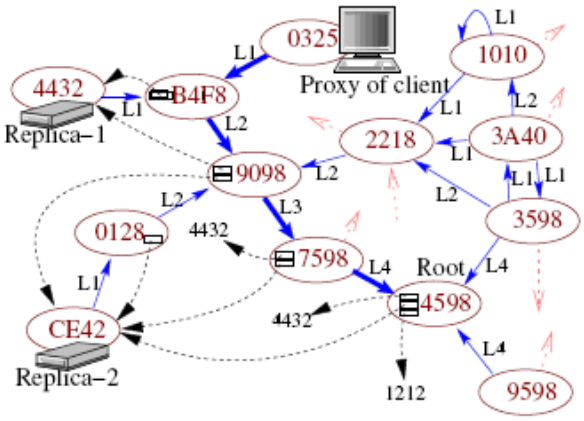

Figure 2.6: "The Tapestry Infrastructure: Nodes route to nodes one digit at a time: e.g. 0325, B4F8, 9098, 7598, 4598. Objects are associated with a particular "root" node (e.g. 4598). Servers publish replicas by sending messages toward root, leaving back-pointers (dotted arrows). Clients route directly to replicas by sending messages toward root until encountering a pointer (e.g. 0325, B4F8, 4432)" ([16], Figure 2.)

2.5.6 Backslash

The authors of Backslash[50] describe their own system as "a collaborative web mirroring system run by a collective of web sites that wish to protect themselves from flash crowds." In other terms, since their solution has features such as request routing and replication, it fits our definition of a CDN. In [50] a peer-to-peer network of web servers are suggested for helping each other during flash crowds. The disk space in a participating backslash peer will be divided into three parts: the web site of the server; a replica storage part and a part that is a temporary cache. It is implemented on top of CAN[44] and every object is replicated just once, but cached as many as needed.[3] Under normal operation a participating web server does nothing extra. But if it is starting to get overloaded it will enter another mode of operation and start to redirect requests to other servers that stores the hot object, either cached or as a replica. A simplified DNS server is present in every node and the request routing mechanism described uses both DNS-based and application layer URL-rewriting along with object encoding. This intercepts request traffic on two different levels, one before the congested network is reached and another where the overloaded node has to redirect the request itself. (The latter can be used e.g. when redirecting requests for embedded objects using the object encoding technique.) An example of this redirection technique taken from [50]: "... the original URL `http://www.backslash.stanford.edu/image.jpg` is rewritten as `http://<hash>.backslash.berkeley.edu/www.backslash.stanford.edu/image.jpg`, so as to redirect the requester to a surrogate Backslash node at Berkeley,

[3]The authors claim that Backslash can use most existing DHTs, as long as it provides neighborhood information.

where `<hash>` denotes the base-32 encoding of a SHA-1 hash of the entire original URL." (Where SHA-1 is short for Secure Hashing Algorithm 1 and is described in [22].) The URL subdomain `backslash` is used for distinguishing backslash nodes, which the backslash DNS server is responsible for and therefore it can redirect client requests coming to those nodes. `<hash>` might be used for lookup within CAN when arriving to the new node and the original URL might be forwarded for transparency reasons, but the authors does not say.

The paper mainly focuses on different caching strategies to be used during a flash crowd, which is certainly interesting but out of scope for this thesis. The described request routing mechanism however, looks very appealing indeed even though it is not described in detail.

2.5.7 CoralCDN

CoralCDN[25] uses Coral[26] and its DSHT for replication and is designed with a DNS-server per node. A Coral node hierarchically divides its peers into three different clusters, based on RTT. Replication is done on demand, by scanning those clusters for the requested object. If the object not is found close enough, a copy is fetched from the origin server and a reference is stored in the Coral system that the node now has the object. The idea is that pointers to popular objects will overflow a node and spill to other nodes, hence distributed sloppy hash table. Consistency is not strong since every request gets a random set of the pointers to an object and the fetched replica might be stale. For a URL to be a part of the CoralCDN, it needs to be "coralized", which is done by appending the suffix `.nyud.net:8090`. Using existing `.net` DNS-servers and the built-in CoralDNS-servers, which contain entries for the `.nyud.net` domain, along with the coralized URLs the system can redirect a client to a close CoralCDN node. If that node has a copy of the requested object it returns it to the client, else it searches the Coral network for the object, fetches a copy and returns it to the client, as described earlier.

CoralCDN is the CDN that comes closest to DOH, the approach is similar to the one in this thesis. There exist differences though, mainly that if the master copy in a CoralCDN system becomes unavailable, all replicas also soon will be discarded as well and the whole site would have disappeared from the Internet until the master comes back. In DOH, DKS provides a basic degree of replication that will guarantee that the site still would be up even if the "master" copy is unavailable for some time. The nice thing about this approach is that any site can be added to the system by just adding`.nyud.net:8090` at the end of the URL.

3 Key issues when creating DOH

With the knowledge now gained, a look on which solutions and techniques that actually could be applied to the DOH system is justified. Listed below are some of the major design issues that have to be decided upon when creating such a system as this:

Request Routing: In the ideal case, the user types the URL into the browser

and get to see the page, with out ever knowing that he was rerouted. The problem is that URLs are location dependent and we want to create a system that is not, while keeping the rerouting transparent from the users point of view.

Object Granularity: How large should replicated chunks be? Should it be per-file, directory or complete sites? And, if needed, how to cluster them?

Bootstrapping: How to solve the bootstrapping problem for peers, a problem all P2P networks have to solve. Can this mechanism be combined with bootstrapping for publishers, creating only one mechanism for both issues?

Adaptive replication: DKS provide DOH with a basic degree of replication but during a flash crowd that might not be enough. Should caching be used and in that case, how do we maintain consistency?

Static vs Dynamic Content: The first prototype of DOH will only support static content, but how to serve dynamic content should be kept in mind during the design phase since future work probably will include dynamic content. Should an existing web server, like e.g. Apache, be used or should a new one be created?

Deployment: How are publishers supposed to deploy their content? If using e.g. the File Transfer Protocol[42] (FTP) should a regular FTP client be enough or should a new one be developed? When dynamic content is supported, easy deployment for publishers will become increasingly important.

Evaluation: How to evaluate the system result. What to compare it with? How can we claim that we have built something that actually works and has good performance?

In this section these questions will be reviewed and we look how they are solved in some of the previously described systems, mainly CoralCDN and Backslash since they are closest to the DOH system. Consider this section as a smooth transition phase between the background section and the analysis and design section.

Request Routing

To achieve load-balance, w.r.t the number of requests that each node handles, is a crucial point when creating a system like this. Unfortunately, from the author's point of view, there is not one solution that stands out from the rest as a clear candidate for using in the DOH system. CoralCDN[25] is the system that is closest to DOH so lets look in more detail how this is achieved in that system.

A CoralCDN node consists of three parts: An HTTP-proxy, a DNS server and the Coral[26] DSHT. To use CoralCDN, a content publisher (or anyone else for that matter), simply appends `.nyud.net:8090` to the hostname in a URL for it to become a part of the CoralCDN. The DNS part of a node contain entries for the `.nyud.net` domain and clients will be rerouted using DNS based rerouting. Coral uses a hierarchal clustering strategy based on RTT to keep

track of "close" nodes. Every node maintain three different clusters, where the RTT is used to determine which node that should be in which cluster. A HTTP-request is rerouted as described in [25]:

1. A client sends a DNS request for `www.x.com.nyud.net` to its local resolver. (A part of the browser)
2. The client's resolver attempts to resolve the hostname using some Coral DNS server(s), possibly starting at one of the few registered under the `.net` domain.
3. Upon receiving a query, a Coral DNS server probes the client to determine its RTT and last few network hops.
4. Based on the probe results, the DNS server checks Coral to see if there are any known nameservers and/or HTTP proxies near the client's resolver.
5. The DNS server replies, returning any servers found through Coral in the previous step; if none were found, it returns a random set of nameservers and proxies. In either case, if the DNS server is close to the client, it only returns nodes that are close to itself. (Because of the clustering)
6. The client's resolver returns the address of a Coral HTTP proxy for `www.x.com.nyud.net`.

The actual rerouting is thus done in two steps: the coralized URL directs the resolver to a node in the system and then the probing mechanism try to find a "close" Coral node to redirect it to. For DOH something similar has to be developed but whether it will be integrated with the system and part of every peer, or implemented as a stand-alone mechanism is yet to be decided.

Object Granularity
The size of the replicated[4] chunks are really important for the system performance. In general too small chunks creates much overhead in terms of lookups and entries in the hash table, whereas too big chunks is not very space-efficient. The two extremes are to replicate on a per-file or a per-site basis. When replicating each individual file itself the used physical space in the system will be optimal. However the overhead of maintaining the mappings and the lookup overhead will slow the system down considerably. When replicating on a per-site basis much more space is used then necessary. Consider the example when one document on a site becomes a "hot object", the size of a site is usually in the order of megabytes and the size of a document in kilobytes. Then it is easy to understand that it is not very space-efficient to replicate the whole site. Per-directory might seem like a good trade-off between the two other approaches, however: since web pages are not browsed sequentially, like e.g. a book, they show poor locality even within a directory. Furthermore, when using a DHT you are not dependent on the directory hierarchy. I.e. two pages that are in the same directory on an ordinary server could very well be hashed to different nodes.

[4]In this section no distinction is made between replicas and cached objects, the term replication is used to refer to both.

So how should you cluster content? There are solutions that suggests using access patterns[15] or server logs[53]. There also exist systems that replicate on a per-document basis, e.g. [40], and even though quite complex, this solution seem to provide good performance.

Bootstrapping
The problem addressed here is how a booting DOH node finds a node that already is in the network. An easy way to solve this problem would be to use some kind of cache of available nodes from the Internet, like e.g. Gnutella[29]. This means that when a node is booting it uses a URL to fetch information about known nodes that are already in the system. It then contacts one of these nodes and tells that node it want to join the system. Using for example XML to define a cache hierarchy, this could be easily achieved. One other advantage is that this web cache approach could be combined with "bootstrapping" for users and publishers, i.e. a non transparent request routing mechanism, where users/publishers get to choose manually from a list between different hosts.

Adaptive replication
DKS will provide DOH with a basic degree of replication depending on how the f-parameter is chosen. But since the system goal is to maintain low latency towards users even during a flash crowd, this might not be enough. Another strategy for surviving flash crowds has to be decided upon, to be prepared in case that the replication will not be enough to satisfy system requirements. This strategy will be caching. There already exist solutions for decentralized P2P web caches, namely Squirrel[32], which is built on top of Pastry[46]. The authors of Squirrel uses some assumptions that are not valid in our case, e.g. that all peers reside on a LAN and that they cooperate within that LAN to create one big cache of multiple client browser caches. But the algorithms described can still be used though slightly modified. In [32] a solution called the Directory scheme is described. In that scheme a client hashes the URL of an object and does a lookup to the node responsible, according to the routing algorithm in [46]. This node is called the home node. It stores a directory of pointers to other nodes that has the object cached, called delegates, and redirect requests to delegates in a round-robin fashion. If no node has a copy, the client fetches the object itself from the origin server and inserts the object in the cache. (In our system, the home node will be the node storing the object and therefore it can always provide a valid copy to the requesting node.) Inspiration also is taken from the P2P caching scheme described in Backslash[50]. As described earlier there are several different modes of operation that a Backslash node could be working in, where normal and overloaded mode are two of them. In normal mode the web server will reply to requests as usual, but when getting overloaded it will start to redirect client requests to other nodes that currently caches the requested object. This could be our starting point as well and combined with a modified version of the Directory scheme a caching mechanism could be created.

Static vs Dynamic Content
To create a web server that handles static content is pretty straightforward, it is when it comes to dynamic content that the servers become more complex. The choice here is simply between choosing an existing web server and adjust it

to fit our needs or to create a new one. As stated earlier, serving static content is the primary goal of this work. However using an already existing web server like e.g. Apache would mean that the system can be more easily adapted to handle dynamic content in the future, since the functionality is already there. An attempt to adapt an existing web server for our purposes will be made, but if it proves to be to slow a process, a new simple web server will be implemented instead. (For more information about handling dynamic content in CDNs look at e.g. [31, 48].)

Deployment
When using static content, deployment is not a big issue. (Compared to deploying dynamic content like e.g. JavaBeans where specialized software is needed.) DOH publishers could actually use their regular FTP-client to upload their content. The two things that needs to be decided upon is how to maintain resonable security and how the hashing of the content is done. On the server side the FTP operations has to be sligthly modified to allow the system to hash the content into place. How login information should be stored and handled is always important and might need its own structure in form of a web site.

Evaluation
The one metric that is really important to look at when evaluating this system, is the latency that the client experiences. One way of evaluating the system is to look at the ideal case, a stand-alone web server with low workload, and compare the latency of that system with DOH, under different workloads. An other way of doing evaluation, is to use one of the software developer's rules of thumb, that e.g. Windows uses extensively, which states that after X seconds something must happen or else the users think that something is wrong. Using this we can make sure that the system always respond within that limit of time, though of course the faster the better, and state that this is sufficient.

4 Analysis and Design

The aim of this section is to define the functionality required of the system and also how it could be implemented. The starting point of the analysis is that a system should be created which have users that wants to achieve their goals by using this system. (See Figure 4.1) The steps of the analysis will be as follows:

- Define sets of users.

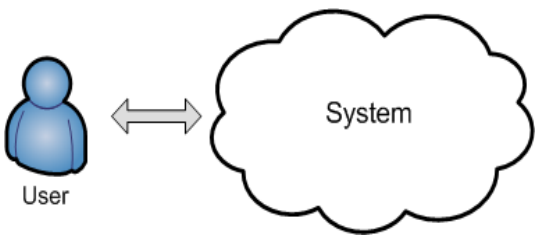

Figure 4.1: Starting point for the system analysis.

- Define subsystems.
- Determine functionality for the sets, from a high-level view to a low.
- Decide which subsystem that should implement the functionality.
- Look at how the subsystems must interact.
- Look at how to implement the functionality.

4.1 Terminology

Terminology used in the analysis section is mostly taken from UML[56]. System specific terminology and definitions will be explained in this section.

4.1.1 Actors

When describing how a system can be used and by whom, the term actor can be used to define different sets of users. In DOH there are several different sets of users, i.e. actors, as defined in Table 4.1. One physical person can of course belong to more than one of these sets, e.g. one might be both a User and a Publisher. In Table 4.1 the software that the actor is assumed to use when interacting with the system is also defined.

4.1.2 Subsystems

There are two major building blocks that will be used when modeling the system and when showing internal interaction patterns. These subsystems, and what they contain, are shown in Table 4.2. They will be explained in more detail in Section 4.4, where the interaction between them is described.

4.2 Actor scenarios

One way to capture the functionality that is required from a system is to start from a high-level view: what do the different actors want to achieve when using the system? Since the actors in DOH are now defined, scenarios of system usage can be created from their point of view, to determine what functionality DOH needs.

Name	Explanation
User	Someone browsing the internet. User Software: a regular browser.
Publisher	Someone that has a web site published in the DOH system. Publisher Software: an FTP-client.
Super user	Someone that creates new users. Super user Software: an FTP-client.
Administrator	Someone that manages a peer in DOH. Administrator Software: a DOH node.

Table 4.1: Actors, and software used for each actor, in the DOH system

Translator	Node
Rerouter	DKS
Web cache	FTP server
DNS server	Web server

Table 4.2: The main building blocks of the DOH system.

4.2.1 User

Joe is a regular guy, that has just typed in his favorite web page, foobar.com, in his browser. Without Joe knowing it, foobar.com has joined a DOH network. This however, since DOH rerouting is done in a transparent way, does not matter. The page will be fetched and presented to Joe just as before.

Later on, Joe find an interesting link to a fishing homepage via a huge news site. This homepage is experiencing its largest load ever. Luckily it is also a part of a DOH network and is replicated and cached so that Joe can see it, perceiving no more delay then usual.

I.e. from the user's perspective the ideal case is when the user is completely oblivious that he/she is using the DOH system.

4.2.2 Publisher

Bob is the web master of a small non-profit organization's web site. He is getting a bit anxious since he has heard that a big charity event is planned in a few days. He suspects that the site will be overloaded but he do not know what to do about it. That evening his wife Sue tells him about DOH and he realizes that this could be the answer to his problem. He registers an account, gets a list of DOH nodes, chooses one and uploads the site to it. He then puts a customized html page as the index page for his site. This html page will reroute the requests to a DOH Translator which in turn will reroute and distribute the requests for his site. The event is a big hit and even though the number of requests is at an all time high, thanks to DOH, no user is experiencing any degrade in performance.

4.2.3 Super user

Alice is a trusted Publisher in DOH who has agreed to help the system administrators. She checks if there are any requests for new Publisher accounts and if so, creates a new account and insert the account information in the DKS DHT.

4.2.4 Administrator

Dirk is a web server administrator for a small company. When he hears about DOH he decides that his company must become a part of this network and downloads the DOH-package from the Internet. He installs it on one of his machines and informs the employees of the company how they shall proceed to upload their content in the future. (As described in the Publisher scenario.)

4.3 Use Cases

The Use case method is well-known and widely used for capturing functionality and behavior of a system. A Use case describes the interaction between the system and an actor trying to achieve a goal while using the system. Taking information from the actor scenarios, Use cases are defined as seen in the use case diagram in Figure 4.2. The use cases that do not interact directly with an actor are usually called abstract use cases and it could be discussed whether or not they should be in the use case diagram. Since the focus here is the functionality of the system the choice is made to display them, for the sake of completeness.

4.3.1 User

When a user browses a web page the steps normally involved for fetching the page are seen in Figure 2.2. DOH will interact with the User on two levels. First when the DNS-lookup is done, it will eventually reach the DOH translator and be translated into an IP address of a DOH node. Secondly the browser will send a request to that node and ask for the page. The node will do a DKS lookup and retrieve the object from the DKS hash table and respond to the User. Thus dividing the process into two main Use cases, as seen in Figure 4.2, is reasonable:

DNS lookup: When a DNS lookup is made (i.e. transforming a URL to an IP address) it will eventually reach the DOH translator. The translator will take the URL and using the Rerouter it will find a node. This node's IP will be sent back to the User.

HTTP GET: When the http request arrives at a DOH node it will hash the URL and perform a DKS get, which will return the object from the hash table. It will then reply with a http response to the User.

Rerouting: The redirector facility has information about node's load and some proximity metrics. It is used to find a node within the DOH system that is not overloaded and close to the User.

DKS get: See Section 2.3.2, page 6.

4.3.2 Super User

The super user in DOH is responsible for creation and maintenance of Publisher accounts. Information that is required for a secure login procedure, is stored in the DKS DHT.

Account: Will store login and content information of a Publisher.

Create Account: Creates a Publisher account.

Delete Account: Deletes a Publisher account.

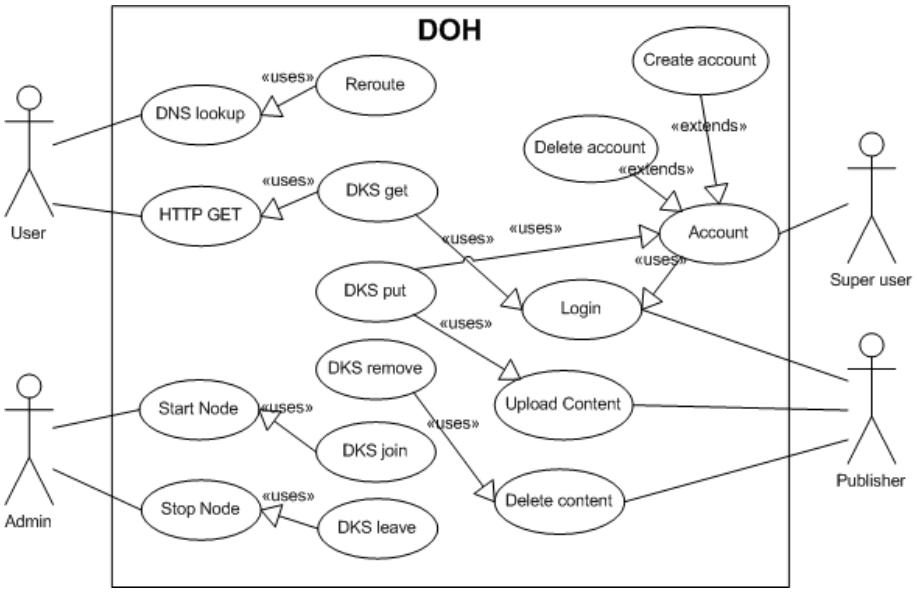

Figure 4.2: Use Cases in DOH.

4.3.3 Publisher

Before using the system an account has to be created for the Publisher. The account information, which will be stored in DKS, will be used for a secure login procedure. After logging in, the Publisher can upload, modify or delete his content. This results in the following Use cases for a publisher:

Login: Logs in to the Publisher's account.

Upload: Uploads content to one DOH node. The content will be inserted into the DKS DHT.

Delete: Remove content from the DOH system.

DKS put: See Section 2.3.2, page 6.

DKS remove: Removes an object from the DKS DHT.

4.3.4 Administrator

For now lets just assume that an Administrator can do two different things. Start and stop a node. The work before the node is started, maintenance work and crashes is out of scope for this analysis. Therefore, the Administrator Use cases looks like follows:

Start node: When the node boots it fetches a cache of known nodes from the web and performs DKS join. When the initialization phase is done it is a peer of the joined DOH network.

Stop node: Whenever a node needs to stop, DKS leave is performed and the node is gracefully removed from the DOH network.

DKS join: See Section 2.3.2, page 8.

DKS leave: See Section 2.3.2, page 8.

Notice that all DKS use cases are existing functionality in DKS, which is already implemented. So at this stage in the project it can be noticed where the DKS system will provide functionality for creating DOH and where functionality needs to be added to fulfill the requirements. Mainly functionality concerning account maintenance and rerouting is lacking and that should be kept in mind for the future sections.

4.4 Subsystem collaboration

In the previous section it has been stated how the different actors should interact with the subsystems. In this section the internal communication within the system for each use case will be shown. In the analysis view it is enough to divide the system into two main building blocks, as Table 4.2 shows. The Translator handles interaction between the user and the system before an HTTP request is sent to a Node. I.e. the last step in the DNS lookup process and the rerouting of a User to a Node in the system. It will also maintain a web cache that is used when Nodes wants to join the system and information about other Translators in the system. A Node, as shown in Figure 1.1, contains an FTP server, the DKS DHT, and a web server. It will serve User's http requests and confirm identity, insert, and remove content for Publishers.

In Figure 4.3 the interaction between the building blocks for all the use cases are shown as a sequence diagram. For simplicity reasons only one node is drawn in the figure representing all nodes in the system. So when Node calls itself in Figure 4.3, that call is actually to another node in the system.

As described in the beginning of this section, the functionality needed in the system has been captured and allotted to a subsystem. The picture of the system has thus evolved from the non-specific view in Figure 4.1 to the more detailed view in Figure 4.4. The question of who does what has now been answered and only the question of how is left.

4.5 Subsystem design view

The two subsystems from Figure 4.4 will now be looked at in further detail to get some ideas on how to actually implement the functionality assigned to them.

4.5.1 Translator

The internals of the DOH translator is shown in Table 4.2. The DOH DNS server will be a mini implementation of a DNS server which will be authoritative for a DOH node only. The Rerouter will get nodes that are up and running from the web cache and will probe them for load information and proximity. Thus when a DNS lookup is made, the DOH DNS will choose one or several DOH nodes that are not overloaded and return the IP of that node. Even though this is not easy to implement, the real challenge to get the system to work smoothly is another. Section 2.4 describe the details of how an actual

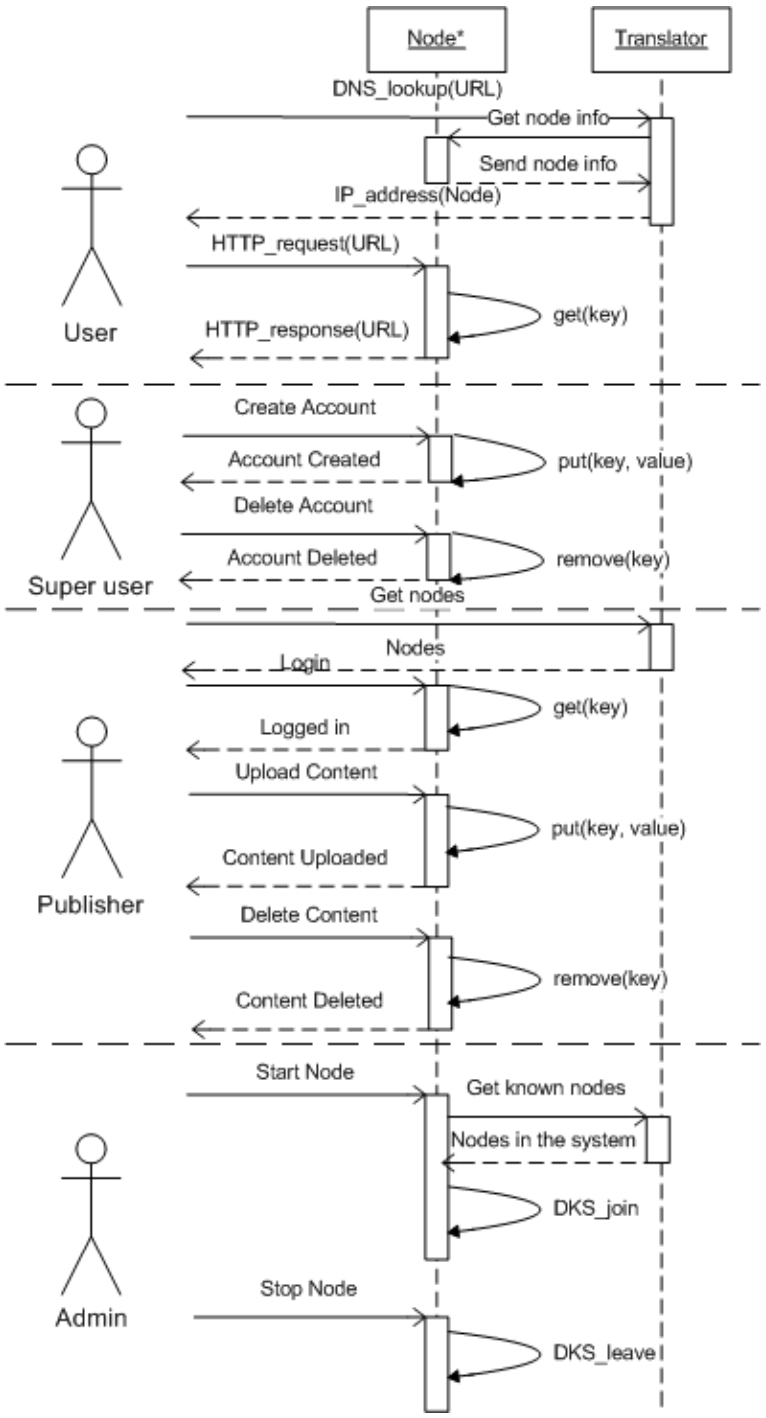

Figure 4.3: Interaction within the DOH system, shown for all use cases as a sequence diagram. (The star after Node is there to symbolize that there are usually several nodes involved in the process.)

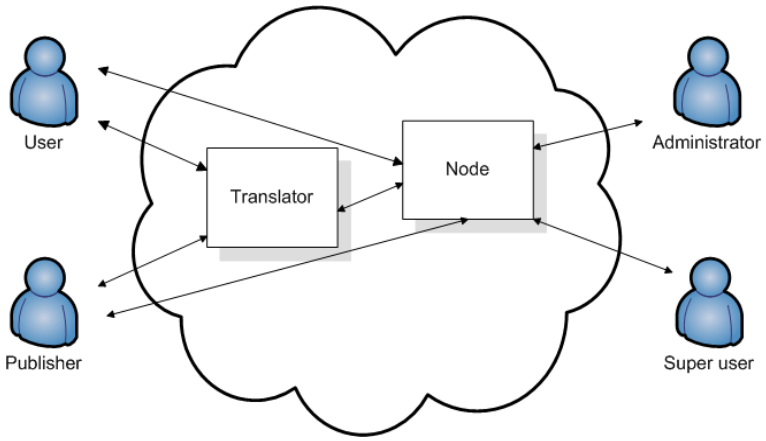

Figure 4.4: The system view after the analysis.

DNS lookup is made. But one DNS characteristic that is not mentioned, is the propagation speed when things change. The DNS system was not developed to handle frequent machine changes and therefore do not respond quickly to them. The normal propagation speed is in the order of 24 hours for an update to be publicly known. Implementing a system like DOH, this becomes a problem when doing the final resolution step and going from a regular DNS server to one of the DOH DNS servers. Consider the resolution process in Figure 2.3 once again, but this time imagine that the authoritative DNS server is part of the DOH system. Since the root server slowly adapts to changes this peer needs to be a part of the system for 24 hours before it is recognized. In a normal P2P system this is a very long time and therefore either some assumptions needs to be added about peers or else the Translator has to be implemented as a stand-alone device.

This fact is disregarded for the time being and the assumption is made that the requests are reaching the translator in some way. Then a closer look of how the actual rerouting mechanism should work and what information it will need, could now be done.

Rerouting in DOH
In DOH, DNS based request rerouting will be used as the primary redirection mechanism. Since the assumption is made that the Publishers should not have to change their URLs to participate in DOH, the first redirection step will be a CNAME redirection. This will redirect the request to a Translator, and more precise to the internal DNS that is in authority. The second step will be for the DOH DNS to choose an IP address to one or several DOH nodes which can serve this request "best", according to some metrics. The DOH nodes available to the DNS will be obtained, as stated earlier, from a web cache.

A secondary request rerouting mechanism is added to the system for those Publishers who do not have the possibility to make changes in their local

DNS server. The difference is only in how the Translator is found. In this approach, a customized HTML page is created and placed where the index page used to be. This page will redirect the users on the application level to the Translator instead of using DNS based rerouting.

The Rerouter part of the Translator handles the collection and analysis of the metrics and the node IPs. For this thesis, two metrics is considered to be enough for determining which DOH node that is "best" for a specific request. The Rerouter periodically probes the nodes in its cache, to get server load and RTT values. These metrics are used to calculate server load and network congestion and is calculated and compared over time. Thus, for this thesis the request routing mechanism only ensures that nodes will not get overloaded. Client to node proximity measurements is considered to be an optimization and will be left for future work.

Bootstrapping
As stated earlier the bootstrapping problem will be solved by using a web cache located at the Translator. It will contain nodes that are known to be up and running. There will be three levels of this caching structure, the first will keep a list of known caches, the second will be the cache and the third will be each individual cache entry, i.e. the actual nodes. When a node wants to join the system, it contacts the web cache it used last time or if that cache is down it queries a web cache list for online caches. Each cache contains nodes that are known to the system and when queried the cache will respond with several valid entries. The booting node will contact one of these nodes and tell that node it wants to join the system. The same mechanism will be used to find a node when a Publisher wants to upload or delete content.

The cache will be defined using XML and the syntax of the cache and a node entry is shown in Figure 4.5. The fields common for both the web cache and a node are several. The `ip` field contains the IP addresses of the cache and the nodes and the `url` and `dks_ref` fields will hold the URL to the cache and the node's DKS ID respectively. The `uptime` is used for both the web cache and the node entry and is a time value of how long the entity has been up and a part of the system. `version` is added for use in the future where different versions of DOH will have different features and a cache or a node can be chosen with respect to that. The `ping` and the two score entries will be used by the next level in the hierarchy for determining closeness and how reliable a cache or a node is. Finally the `ttl` value in the node entry will be used to determine when to update the entry.

4.5.2 Node

Two different approaches to implement the Node functionality and to combine its components will be described. The first will be used if the choice is made to use an existing web server and the second if a new one is created instead. The three components of a Node, as shown in Figure 1.1 is an application server, the DKS DHT and a web server. The application server will be listening to port 80 for HTTP-requests. When a client requests some URL, the application server will call DKS to do a lookup operation and retrieve the object requested. It is

```
<doh_webcache>
  <ip></ip>
  <url></url>
  <uptime></uptime>
  <version></version>
  <ping></ping>
  <cache_score></cache_score>
  <node></node>
</doh_webcache
```

```
<node>
  <ip></ip>
  <dks_ref></dks_ref>
  <uptime></uptime>
  <version></version>
  <ping></ping>
  <node_score></node_score>
  <ttl></ttl>
</node>
```

Figure 4.5: XML syntax for the DOH Web Cache. To the left: the web cache. To the right: a node entry.

now that the two approaches differ. In the first, when using an existing web server, the object is stored in the web server's root folder, with the directory structure matching the one of the URL. Then a modified HTTP-request is sent from the application server to the web server via the loopback interface, modified so that the web server responds to the client instead of the application server. In the other approach, when creating a new web server, the fetched object can be kept in some local data structure and sent to the client directly from the application server. The advantages of using the first approach is that the system could be easily extended to implement more of the existing web servers functionality. The disadvantage is that a garbage routine has to be created to remove old objects and also that this approach would probably be slower, since objects has to be written to disc before being returned. The advantages of creating a new web server is that objects does not have to be written to disc and the disadvantages is that adding more features to later versions of the DOH system will be harder.

Adative replication
Using the Directory scheme defined in [32] and combining that with the entry caching scheme of DNS, an adaptive caching algorithm has been devised.

The approach used is that whenever an object is requested it is likely to be requested soon again. Therefore it makes sense to cache it. What is not mentioned explicitly when describing the DKS lookup is that the return path, when returning the object to the node doing the lookup, is also recursive. As with the DNS, this could be used for caching the result along the lookup path. When a new request is arriving somewhere in the system, the node receiving the request will look in its cache if it already has a valid copy of the object. If it does it will return the object, if it does not it will perform a DKS lookup. When the object is found it will be cached along the respond path on its way back to the querying node. Consistency could be kept by using the if-modified-since field that is built-in to the header of the HTTP protocol. When caches are full, e.g. the Least Recently Used (LRU) algorithm, or some other caching policy, could be used for deciding which objects to evict.

A normal lookup in DKS is described in Section 2.3.2 on page 7. Assume once again that node 0's request arrives at node 5. When the object is

returned to node 0 it will cache it, as would any other node along the respond path. When a new request is made node 0 will check if it has a valid copy of the object and respond itself.

Object Granularity
This is one of the most important issues to get right for the system to show good performance. In DOH files will be hashed directory-wise. I.e. all files in the same directory will be hashed to the same node. E.g. as in Figure 4.6, where the files `banner.swf` and `index.html` in directory `www.url.com` will be hashed to one node and files `1.jpg`, `2.jpg`, and `index.html` in directory `b` will be hashed to another. This is done because the directory structure is easy to obtain and, more importantly, effectively can be used when transforming a URL to a key. E.g. consider the URL `http://www.url.com/a/b/index.html` When this request reaches a DOH node, it simply performs `dks_get(www.url.com/a/b/)`, where `www.url.com/a/b/` is the key.

Only because the files are hashed directory-wise it does not mean they have to be returned directory-wise. In the previous example, DKS allows `index.html` to be sent along with the request as the value to be retrieved, since this entry in the hash table might, and actually does, contain multiple values. Further optimizations could be done, such as calculating which the embedded objects of `index.html` are (`1.jpg` and `2.jpg`) and return them all on the same lookup, since the probability for a request concerning them in the near future is very high.

Login
For the login feature an existing Public Key Infrastructure (PKI) could be used. When a person wants to be a Publisher he will have to contact a Super user. The new Publisher needs to have a digital signature certified by some Certification Authority (CA). The Super user will add the new Publishers username as a key in the DKS DHT and as the value, the Publishers public key will be stored. Authentication is done by receiving the username of the Publisher that are trying to log in, encoding some random numbers with that username's public key and challenge the Publisher to encode it with his private key. If this string matches the original, the login has succeeded.

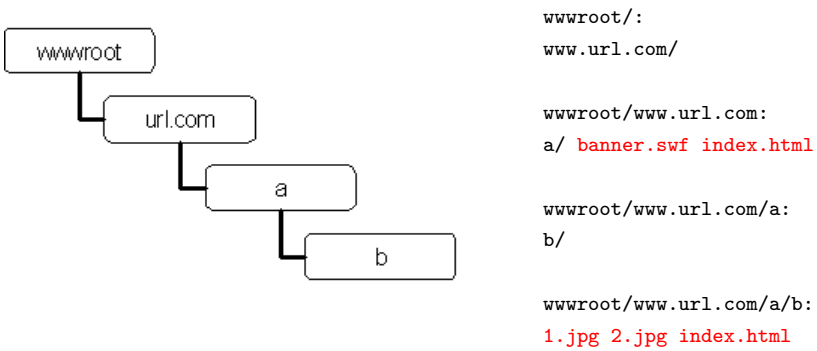

Figure 4.6: Example of a directory structure for the site www.url.com. To the left: the directory hierarchy. To the right: the files stored in each directory

5 Implementation

A prototype has been developed to check the validity of the design and to do some performance tests to see if the system will meet the performance demands required of a CDN.

This Section is divided into two major subsections, each describing a subsystem of DOH: the Node and the Translator. The implementation details will be described from the functionality perspective, using the specification created in Section 4. In this context it, hopefully, will be easier to understand the reasons for different implementation details.

5.1 Development platform

Since DKS was already implemented in Java, it became a natural choice when considering implementation language. The following version and edition was used: Java (TM) 2 SDK (Standard Edition) Version 1.4.2[54]. The code of the prototype was written in Borland JBuilder X Enterprise[13].

5.2 Node

The node functionality was decided by the actors in the design phase and could be summed up like this:

1. Should answer to HTTP requests with content retrieved locally or via lookup in DKS.
2. Should support uploading and deletion of content for publishers.
3. Should support management of publisher accounts.
4. Should join and leave DKS gracefully.

In the prototype of the Node, the web server functionality required by the User actor was implemented by modifying the Jetty[27] web server, which is licensed under the Apache license[7]. The functionality required by the Publisher was realized using the FTP standard and implemented by modifying the JavaFTP server[11] package, which also is licensed under the Apache license. The Node prototype also is a peer in the DKS network and uses the DKS DHT API to store and retrieve data.

5.2.1 Web Server

Jetty 5[27] is used in DOH to implement the web server functionality that is required. An overview of Jetty's internal structure can be seen in Figure 5.1. There is a set of HTTPListeners that listens for requests and then notifies their server on incoming requests. The HTTPServer is responsible for matching a request to an HTTPContext, usually by using the directory part of the URL. The context will pass the request on to its registered HTTPHandlers until the request is recognized and handled, where handled usually means that a response has been sent back to the requestor. The most common listeners and handlers, e.g. listeners for HTTP and SSL and handlers for "page not found" and static content, are already provided by the Jetty developers.

The fact that Jetty is modular in this way, will allow for a change in the DOH design. Since it is fairly easy to add new behavior to the web server, there is no need for an application server listening to port 80 (as described in Section 4.5.2). All that is needed is to create your own, customized, handler. In our case, the functionality that needs to be added to make Jetty "DOH compliant" is to create a handler that searches DKS if the page not is found locally. In the created DOHHandler, the requested page is searched for, first locally in the web server cache and if not found the page is retrieved from DKS and written to the disc. Then it can be displayed using the regular handler functionality provided by Jetty. How the DKS lookup is done will be described later, in Section 5.2.3.

5.2.2 FTP Server

JavaFTP server[11] is a package implementing the FTP standard and was a part of the now closed Apache Avalon project[9]. It has been modified so that

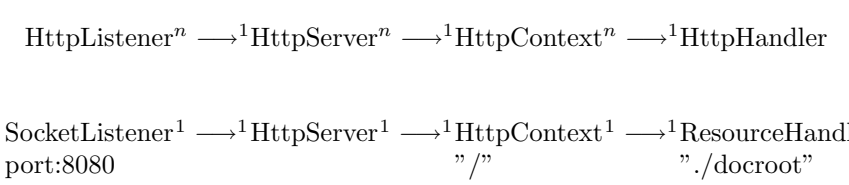

Figure 5.1: *Top:* a generic overview of the internal Jetty structure. *Bottom:* An example. This depicts a single listener on port 8080 passing requests to a single server, which in turn passes them to a single context "/" with a single handler which returns static content from the directory "./docroot".[27]

when connected to a host, instead of uploading files to the host's local file system, the files are uploaded and stored in DKS.

Since there should be no difference in the look and feel between the FTP server in DOH and a usual FTP server, information about the user and his uploaded files needs to be stored somehow between sessions. This is done by storing user information as an XML scheme in DKS, retrieve it during the login procedure, record changes locally during the session, and store changes in DKS when quitting. There are two reasons for having user information encoded as XML. First of all, since a directory might contain both other directories and files recursively, no simpler structure would do. And second, using XML let us use already existing, efficient, XML parsers for parsing the User information.

When a Publisher makes an attempt to log in, a lookup in the DHT is done for that username. If there are no hits, or if the password is wrong, login will fail. If there is a hit and the password entered is matched against the password stored, DOH will parse the user information and display the previously uploaded content, if any. The complete structure of the user information XML scheme can be found in Figure 5.2.

The user information consists of three types of directory elements: domains, directories and files. The file element keeps information about the file, such as: file name, size, and time of last modification. A domain and a directory are the same thing, except that a domain will only be present at the top level of the directory structure and has the imposed restriction that it should be named to the domain name of the web site it is storing. Remember that objects should be stored in DKS so that they could be found using the URL, as described in Section 4.5.2. If the top directory is named after the domain when uploading content, getting the keys for storing objects will be simple. Assume e.g. that someone wants to publish his site that has the domain name `www.url.com` with files as shown in Figure 4.6. If the top directory, the domain, is named `www.url.com`, all files will be stored under the correct hash index just by hashing the directory path.

```
<user_info>
  <user_name></user_name>
  <password></password>
  <domain>
    <name></name>
    <dir></dir>*
    <file></file>*
  </domain>*
</user_info>
```

```
<dir>
  <name></name>
  <dir></dir>*
  <file>
    <name></name>
    <modified></modified>
    <size></size>
  </file>*
</dir>
```

Figure 5.2: User information XML structure. A star means zero or more occurrences of that label. E.g. a dir can contain multiple other dirs. Labels without stars is mandatory and should be in the structure only once.

5.2.3 DKS Peer

To be able to join the DKS network, a node already in the network must be contacted. In DOH a booting node must have a URL to a Translator and the booting node will get information about existing nodes via his Translator. A small GUI has been developed, containing a start and a stop button. Pressing start will boot the node and make it a part of the DOH system. Pressing stop will make sure that the node leaves DOH, and especially the DKS network, gracefully. I.e. the leaving node will leave DKS as described in Section 2.3.2.

The current implementation of the DKS API does not provide a hash function. It just states that keys, or more accurate indexes, used should be 64 bits[5]. So when using the distributed `put(key, value)` method of DKS, the `key` should be the already hashed index. The `value` should be a DKSObject, where the DKSObject class is a wrapper around an array of bytes. The `lookup(key)` operation, where `key` once again is the already hashed index, returns a DKSObject array containing all the objects stored under that index. The internal order of this array is unspecified.

Since the decision was made in DOH to use URLs as the keys in the hash table, some special arrangements had to be made to handle the practice of not referring to the index-page when typing URLs. When receiving a URL that ends with a front-slash DOH will therefore first perform a check to see if the specified directory contains an index-file. E.g. assume that the URL `http://www.url.com/` is received by a Node. Then the URL will be hashed and since it ends with a front-slash, the Node will first perform a search within the directory for an index-file. If an index-file is found, it will be sent back. If not, a listing of all the files in the directory will be returned instead. However the issue with multiple URLs pointing to the same content has not been addressed so far in DOH, i.e. the URLs `http://url.com` and `http://www.url.com` will not lead to the same page.

In order for the DOH system to handle large objects, a way of fragmenting objects is needed for mainly one reason: to avoid running out of memory. Without fragmentation, consider the scenario when e.g. a Publisher is uploading a large file, then all of the file must be kept in memory before it is being stored in DKS. Fragmenting the file means that the system has better control over memory usage and can avoid running out of memory.

The information that is needed for fragmenting objects and then merge them correctly again are the following: type of object, an optional file name (if the type specifies a file), number of fragments, the number of this fragment, and finally the data. The different types of objects in DOH are two: files and users. A syntax example: if the delimiter used is an & and the stored object is a file with the name index.html, divided into 5 fragments where this is the first, then this is how it would be stored in the DKS DHT: FILE&index.html&5&1&<`data`>

A class called DOHObject has been implemented, keeping an array of

[5]DOH uses SHA-1[22], shortened to 64 bits, to create a hash table index from a key.

these fragments. When a lookup operation is done from e.g. the web server, the returned DKSObject array is searched for the fragments of the requested file and a DOHObject containing all the fragments of the requested object is returned. Figure 5.3 graphically shows the steps performed by DOH when adding a file to DKS and Figure 5.4 shows the steps when retrieving it again.

For the prototype a straightforward cache algorithm has been implemented as default: Each node will cache the files that it receives from the `lookup` on disc. After that the Jetty web server will itself choose which files to cache in memory. The cached files are said to be stale when a certain amount of time has passed, currently 5 minutes, and a new copy will be retrieved from the DHT. No garbage collection has currently been implemented. The reason for abandoning the cache algorithm described in Section 4.5.2 is because the current version of DKS does not support the features needed for it to work.

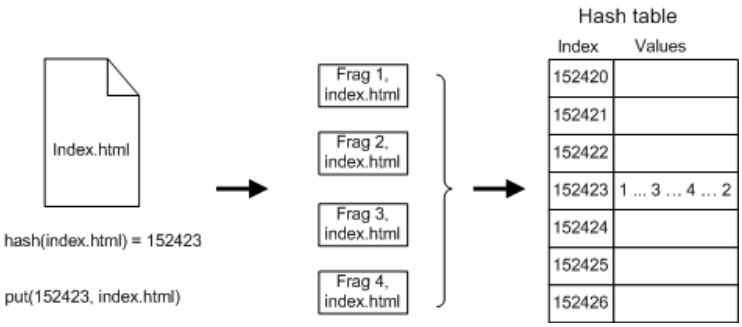

Figure 5.3: Adding files to DKS in DOH: first the key is hashed to get an index; if needed, the file is fragmented; each fragment is then added to the calculated index in the hash table.

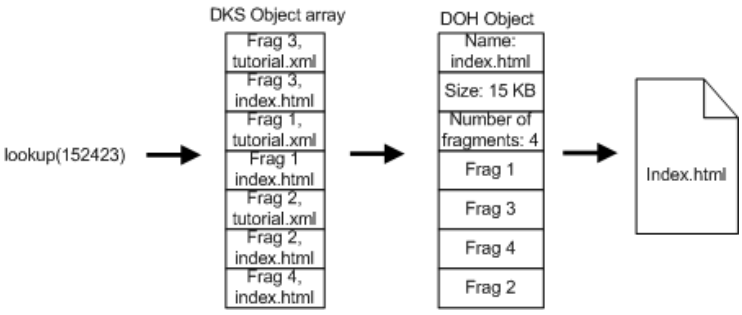

Figure 5.4: Retrieving files from DKS in DOH: once again hash the key to get the index; lookup returns a DKSObject array containing all the files added to the index; filter the fragments for the requested file and put them in a DOHObject. The file can now be assembled.

5.3 Translator

In the prototype the Translator will be a stand-alone part with the following functionality:

1. Keep a cache of existing DOH nodes.
2. Re-route HTTP requests to a node in the system.
3. Calculate which node is best to re-route to.
4. Display node information to Publishers in a human-readable format.

To achieve this functionality two packages was implemented: the Web cache and the Rerouter package. These will now be described in more detail.

5.3.1 Web cache

As for the prototype a small protocol has been created between the Node and the web cache part of the Translator. (See Table 5.1 for the complete protocol specification.) The web cache keeps track of two lists of nodes: the now active nodes and old, inactive, nodes. This is to be able to calculate the node score more accurately, a node that jumps in and out will get a lower score since it is less trustworthy than a stable node. (The structure of the web cache can be seen in Figure 4.5)

For a node to be able to boot and become a part of DOH, it needs a URL to a Translator's web cache. When booting, it queries the cache for known nodes and the cache responds with DKS references to all nodes it knows. A DKS reference contains, among other things, the node identifier and the node IP. After receiving the DKS references the node can start the process of joining the DKS network. When the node has joined, it sends an INFO-message containing its node information to inform the cache that a new node now exists. The node information is shown in Figure 4.5. When a node is leaving, it sends a LEAVE-message to the web cache, which removes the node from the active list.

To be able to calculate which node is the "best" to re-route to, the nodes also inform the web cache about their load. There are two different types of load messages, the regular load message, UPDATEC, and the OVERLOAD-message. If the web cache receives an OVERLOAD message it will decrease the score

Initiator	**Message**	**Msg Content**	**Replay**	**Replay Content**
Node	BOOTING	-	CACHES	A web cache
Node	INFO	A Node entry	ACK	-
Translator	PING	-	PONG	-
Node	LEAVING	Node ip	-	-
Node	OVERLOAD	Node ip	-	-
Node	UPDATEC	Node load info and IP	ACK	-

Table 5.1: Protocol between a DOH node and the Translator web cache.

of that node by 50% to assure that the load of the node will be decreased instantly. UPDATEC is sent regularly when the node has received a fixed number of requests, currently 1000. The content of the message is information of the node's load. The load is calculated by using the undocumented class `sun.misc.Perf`, to get precise timing and also to be able to get the CPU frequency. The class, included in Java SDK since version 1.4.2, allows for accessing the high performance timer of the CPU. The formula for calculating load in the prototype is like follows: time in ms that it takes to handle 1000 requests divided by the CPU frequency times a constant k, where k = 1000 to get values between 1 - 1000 on the testbed. There are more elaborate ways of calculating load, but this is sufficient for the evaluation of the prototype.

When the cache receives an UPDATEC-message, it calculates a new score for that node. The node score is based on the inverse load of the node, inverse number of incarnations of the node, and also on the RTT between the node and the cache. (Hence the PING and PONG messages in Table 5.1.) The node score will be a number between 1 - 1000, where 1000 is the best, and the node with the currently best node score will get to answer the incoming requests. The node score is decreased with one on every rerouted request, to ensure that fairness is achieved.

5.3.2 Rerouter

The Rerouter prototype is implemented as a Listener that listens for incoming requests on the standard HTTP port and handles the incoming HTTP requests in one of two different ways. Usually it will get the "best" DOH node from the web cache and send an HTTP 302 response message back to the requestor, adding the "best" DOH node's IP to the old URL. The HTTP standard[24] states that the 302 message is used for pages that is temporarily moved and therefore it can be used for e.g redirecting. (See Figure 5.5 for a complete example of an HTTP 302 answer.) When having received the 302 response the requestor will try to get the requested page from the new URL, thus contacting a DOH node, and the request will be handled as described in Section 5.2.1.
The special case is when the requested URL contains the word `doh_webcache.xml`, then the Rerouter will assume that the request is coming from a Publisher wanting to find an IP address to a node. In this case the Rerouter will create an HTTP response, the content being the nodes in the

```
HTTP 302 Found
Date: 13.37 May 31:th 2005 GMT
Server: DOH Server 0.3b
Location: http://192.168.2.23/www.url.com/a/b/index.html
Content-Length: 0
Connection: close
Content-Type: text/plain
```

Figure 5.5: An example reroute message. The HTTP 302 answer from the Rerouter to a request for `http://www.url.com/a/b/index.html`, where `192.168.2.23` was currently the "best" DOH node's IP.

web cache transformed to the XML scheme described in Figure 4.5, and thereafter reply to the request itself. From this XML page the Publisher can choose whatever node he wants and connect to it.

6 Validation

This Section will cover the most important parts of the validation of the implemented prototype. One can say that validation means testing whether the implementation fulfills the proposed design. A system like DOH is rather straightforward to validate, since the use cases are specified and most of the system functionality is towards a human user. E.g. it is easy to validate that a User is rerouted and that a Publisher is connected and able to upload and delete content, using the mere eye.

However there exist other parts of the system that needs to be validated as well, first of all the Rerouter needs to be validated in terms of how fair it really is. Is it dividing load at all? This will be the first validation scenario and then the system will be validated to the use cases described in Section 4.3.

6.1 Fairness of the Rerouter

In the validation of the Rerouter the simulated nodes will be assumed to be equal according to performance. I.e they can handle the same number of requests in a given amount of time. This way it can be verified if the Rerouter is fair, because then the nodes should get the same amount of requests.

A test program is developed that will initiate a Rerouter, spawn off and attach 5 simulated nodes to it, and then perform a given number of requests. All the nodes will start off with an equal node score, in this case it is the max node score of 1000. At first a run is done that performs 5000 rerouting requests and, as the results in Table 6.1 shows, the Rerouter so far seems to be fair.

The nodes are periodically assumed to inform the Rerouter of their load, and for the simulation this is done every thousand request. To further verify the Rerouter's fairness the number of requests is increased to 100000 and this test setting is iterated 1, 10, 100, and 1000 times to check for tendencies. (See Table 6.2.) What can be noticed is that the median is close to 20000 requests already with only one iteration, and that the range between maximum and

Node	**Number of requests**
3	1001
2	999
1	999
0	1000
4	1001

Table 6.1: Rerouter simulation of 5000 requests with 5 nodes, where all nodes start with an equal node score.

minimum values are decreased from 209 to 3 when you go from 1 to 1000 iterations. Thus the Rerouter, under perfect conditions, could be considered fair. (See Appendix B for the exact numbers of the runs.)

6.1.1 Asymmetric node scores

Now when it has been shown that the Rerouter is fair using symmetric node scores a test is done to check for another important characteristic of the Rerouter, namely to see what happens when the node scores are asymmetric. In this validation test, four nodes are simulated: Node 1 has a node score of 1000, Node 2 has a node score of 500, and Node 3 and 4 has a node score of 250 each. 2000 requests are then executed and the result, as shown in Table 6.3, is good: the Rerouter divides the load according to the node scores. An increase to 3000 requests shows the same pattern as the symmetric case: after a few iterations all nodes has roughly got 250 more requests each. (See Appendix B.)

6.2 Use Case validation

The use cases, see Section 4.3 and Figure 4.2, is now to be validated. To capture all the functionality described in the use cases and to validate them, the following scenario will be used: find a Node to login on, login to a Node's FTP server with username Bob and password foo, upload some content, browse that content, delete the content, and try to browse it again. After that Bob's Publisher account will be deleted and another login will be attempted. That will show that one can:

1. Start a Translator.
2. Start a DOH Node.
3. Add a user.
4. Find nodes using the translator.
5. Login.
6. Upload a web page.
7. Browse a web page that are uploaded.
8. Delete a web page.
9. Delete a user.

# of iter	**Max**	**Min**	**Median**
1	20 106	19 897	19 992
10	20 065	19 947	19 995
100	20 018	19 975	20 000
1 000	20 001	19 998	19 999

Table 6.2: Rerouter simulation with 5 nodes and 100000 requests, iterated from 1 - 1000 times.

Node	Node score at the start	Number of received requests
1	1000	1000
2	500	500
3	250	250
4	250	250

Table 6.3: Rerouter simulation with 4 nodes and 2000 requests using asymmetric node scores.

Contacting the Translator to get a Node to upload too might look like in Figure 6.1 where the web cache information is displayed. Figure 6.2 shows the login dialog between then FTP client and the FTP server and Figure 6.3 shows when a Publisher (Bob) is uploading his content to a DOH node. Figure 6.4 shows a User browsing Bob's page and Figure 6.5 shows a User trying to browse the same page after Bob has removed it, and the time for updates has expired. Finally, Figure 6.6 shows Bob attempting to login after his Publisher account has been deleted. The use cases "Stop Node" and "Stop Translator" have been validated but is excluded from this presentation. (For more instructions on how to use DOH see Appendix C.)

6.3 Portability

Since DOH is written in Java it would be a nice feature for the system to be portable between different operating systems. Under the implementation of DOH this has been a goal and the system has been tried on Linux (Mandrake 9.2) and Windows XP and at least runs on both these platforms.

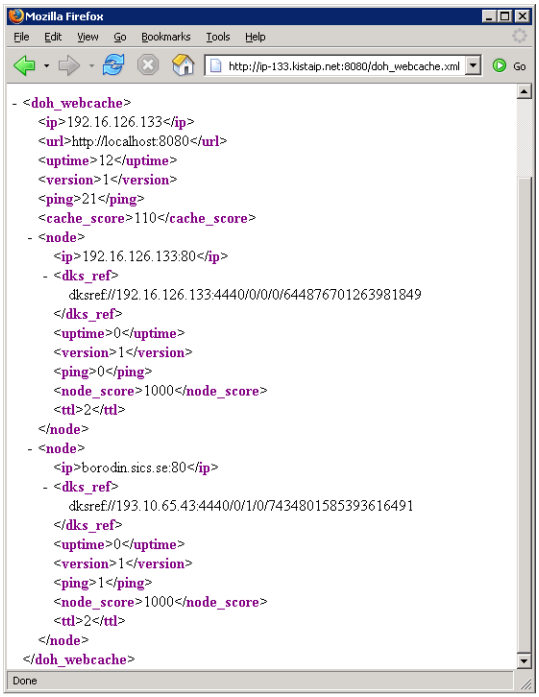

Figure 6.1: An excerpt from a DOH webcache showing how a Publisher can find a Node to upload content to.

```
Connect to: (2005-09-09 10:33:35)
hostname=ip-133.kistaip.net
username=
startdir=
ip-133.kistaip.net=192.16.126.133
220 Service ready for new user
USER bob
331 User name okay, need password for bob
PASS **********
230 User logged in, proceed
SYST
215 UNIX Type: FtpServer
FEAT
502 Command FEAT not implemented
Connect ok!
PWD
257 "/" is current directory
```

Figure 6.2: The Login dialog between the DOH FTP server and the client.

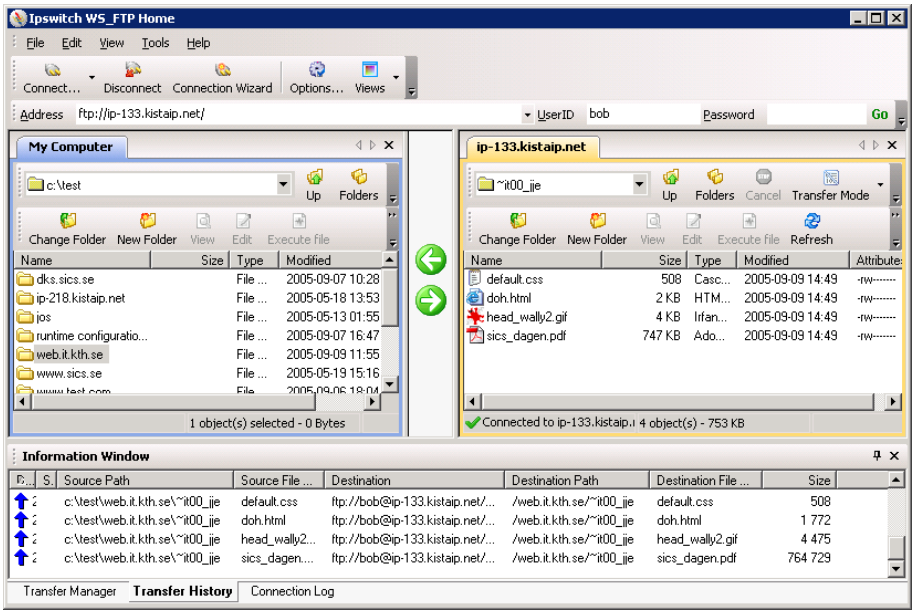

Figure 6.3: Uploading content to a Node's FTP server.

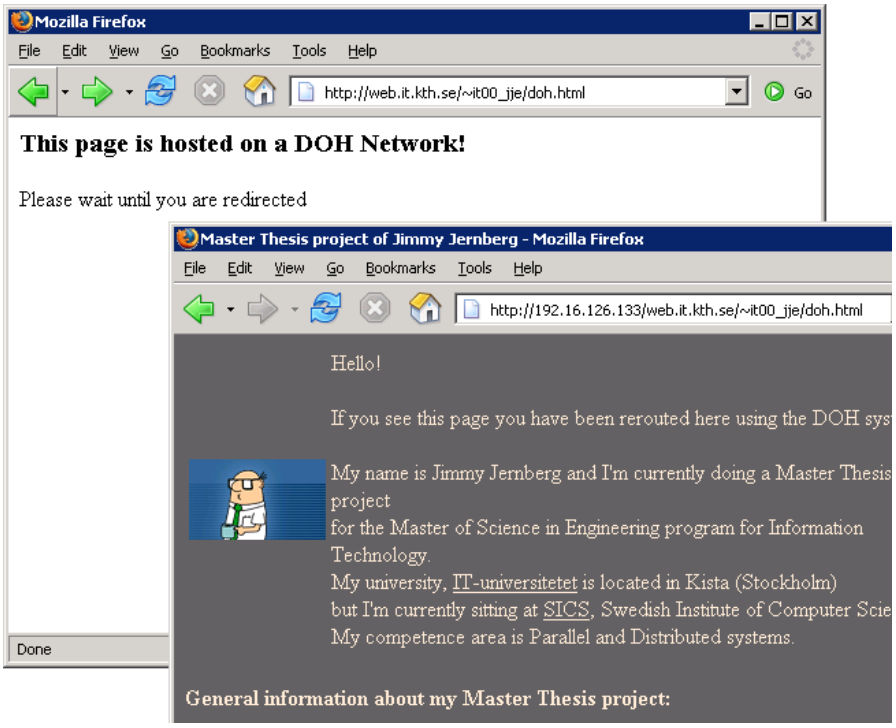

Figure 6.4: An example of rerouting in DOH. Browsing the newly uploaded content.

Figure 6.5: Trying to browse the same page, after it has been removed.

```
Connect to: (2005-09-09 15:20:03)
hostname=ip-133.kistaip.net
username=
startdir=
ip-133.kistaip.net=192.16.126.133
220 Service ready for new user
USER bob
331 User name okay, need password for bob
PASS ***********
530 Access denied
```

Figure 6.6: Failed DOH FTP server login attempt. (User bob has been deleted.)

7 Evaluation

In this Section the performance of the system will be tested. The crucial performance factor to get right in DOH is the User perceived latency and it will therefore be the focus of the evaluation. At first a test is designed that will examine some of the design properties of DOH with respect to this latency, namely the object granularity of the system and the use of caching. After that the prototype will be evaluated with respect to its performance and furthermore a simulation of the system will be presented and used to perform more thorough tests on a larger scale.

7.1 Test-bed platform

The computer running the first set of tests is an AMD Athlon 64 3000+ with 512 MB RAM. The operating system is Microsoft Windows XP SP 2.

To run the different parts of the system Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2_04-b05) was used.

In addition Borland Optimizeit Enterprise Edition 6[14] is used to run some fine-grained evaluation tests. Opimizeit is a tool for isolating and resolving performance of Java (J2EE) applications, it can provide information on a level of functions.

7.2 Critical Path

The critical path of any system is the path of events that will be worst, i.e critical to get right, with respect to the performance measurement used. In the case of DOH the performance measurement is response times and the critical path for a request is as follows:

1. Redirection from old home to a Translator.
2. Redirection from a Translator to a Node.
3. Retrieve requested file from the DHT.
4. Unwrap files and write them to disc.
5. Send data to User.

The redirection from the old home to a Translator could be done in two ways, as explained earlier. The DNS entry for the page might be updated, or a simple JavaScript-file could be used. This redirection step is not considered when evaluating the system, since it strictly speaking not is a part of the system.

To analyze the critical path, files of increasing sizes where added to the system and then extracted again, using Optimizeit[14] to get the timing values shown in Table 7.1. Also it can be noted that when it comes to bigger files, the performance of DOH is the performance of DKS since over 90% of the time used by a request of a file with a large file size is spent in DKS.

File size	findInDKS	Lookup	Unwrapping	Fragments	Total time
5 Kb	41 ms	12 ms	6 ms	1	15 ms
15 Kb	47 ms	24 ms	5 ms	1	50 ms
17 Kb	35 ms	18 ms	6 ms	2	40 ms
30 Kb	58 ms	41 ms	6 ms	2	65 ms
60 Kb	111 ms	90 ms	18 ms	4	125 ms
120 Kb	160 ms	113 ms	29 ms	8	250 ms
240 Kb	293 ms	252 ms	30 ms	15	512 ms
480 Kb	521 ms	445 ms	47 ms	30	1010 ms
960 Kb	872 ms	784 ms	71 ms	60	2003 ms
1920 Kb	1650 ms	1504 ms	117 ms	120	4050 ms
3840 Kb	3115 ms	2868 ms	206 ms	240	7121 ms

Table 7.1: The time used by DOH for retrieving files of different sizes. findInDKS consists of two stages, first the lookup and then the unwrapping of DKSObjects into files on the disc. The difference of total time from findInDKS is mostly due to DKS related activities on lower levels of the DKS overlay network.

7.3 Design choice evaluation

As described in Sections 3 and 4.5.2, the object granularity of the system is important for good performance. DOH is designed to hash and retrieve objects on a per directory basis, i.e if one file from a directory is requested all files from that directory is currently retrieved and cached. Here this approach will be compared to hashing and retrieving objects on a per file and per site basis, with and without using a cache.

The test is performed using just one Node, and the Translator and the Node is residing on the same computer. Approximately 50 files with a mean file size of 10Kb, in 6 domains, and 16 directories are then uploaded to the Node. The test program will act as a web browser. It will randomly pick one of the domains and then request some of the pages in that domain. The time for each request will be recorded using the undocumented class `sun.misc.Perf`, which can be used to create a very precise timer. The system is restarted before every run, i.e the node does not have any files cached at the beginning of a run. Runs with 100, 1000, and 10000 requests where made and the results are shown in Figure 7.1, which shows the average response times for each strategy. What can be noted is that the impact of a "cold" system is big as expected: the average response time for all strategies in the 100 request run is significantly higher then the response time in the 10000 request run. This suggests that the system should benefit from the use of caches, which is also supported by looking at the strategy in the figure that do not use a cache at all. (Using the site- and directory-wise approaches without caching was omitted since they make no sense. What is the point in retrieving more objects from the DHT and not cache them?)

When reviewing the strategies using this test as a base, there are very small differences between the approaches: in the long run all strategies that

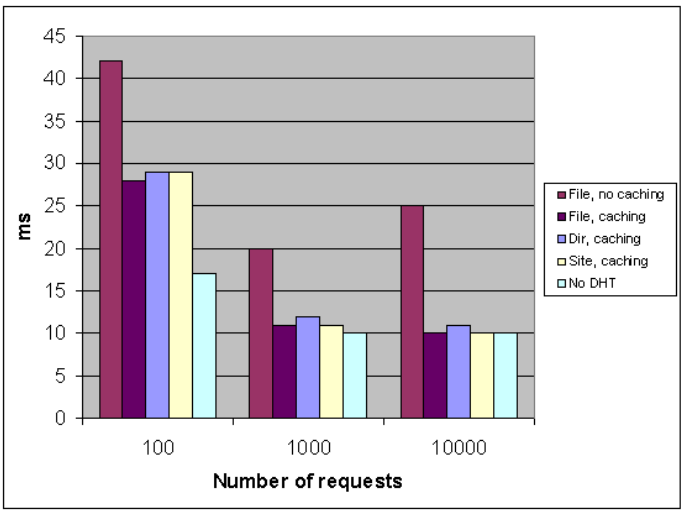

Figure 7.1: Average request response times for different object granularities in DOH.

uses caches differs very little from the ideal case (No DHT), which is as one should expect.

Two other important performance measures is the amount of memory and CPU used. During the tests of the strategies using a cache, the node used about 30Mb of memory and the processor of the test machine never exceeded 30%. During the file-wise test with out caching, the amount of memory used was 40Mb and the processor worked in between 50-100% all the time. To further look at the differences between caching and not caching objects Optimizeit was used get the timing characteristics of the two file-wise approaches. Two new runs where made, with 1000 requests, and the result is found in Table 7.2. Thus it is relatively easy to understand that how to cache objects and for how long is very important for system performance.

One of the reasons that there are very small differences between the directory- and site-wise approaches is of course the design of the tests. In total the size of the uploaded files are about 1 Mb, which is to small to see the differences between these approaches. But as explained in Section 3 downloading the complete site every time a page is requested is not very efficient, which also can be realized by looking at Figure 7.2 (extracted from Table 7.1) where the

	File search (%)	Total time (s)
Cache	26	8
No cache	86	35

Table 7.2: Using a cache or or not. Procentage spent looking for files and total run time to execute 1000 requests.

retrieval time for files of different sizes in DOH is displayed. The file-wise approach is optimal when it comes to retrieval-overhead, since it has none. However it shows a smaller cache hit ratio then e.g. the directory-wise approach. In conclusion it can be said that if picking one of the approaches, the directory-wise seems to give the best tradeoff between downloading overhead and cache hit ratio when the load of the system is reasonable.

7.4 Performance Evaluation

The testbed used for the performance evaluation is several Pentium 3, 500Mhz computers with 256 MB RAM running Linux (Red Hat 9.3). In this section the term workload refers to the grade of parallelism used in that particular run: the number of parallel, independent, clients that are used at any given run is the workload of the object being evaluated.

First scenario
The first testing scenario was to evaluate a stand-alone Jetty web server, to use as a baseline for comparisons. This scenario is very similar to the one used when validating the system: approximately 50 files, with a mean size of 10Kb was uploaded to the system and the test program was basically a browser that requested a random page and recorded how long time each request took. This was repeated for several workloads and the result is shown in Figure 7.3. After that, the first evaluation test of DOH was made. Choosing to use the workload of 100 parallel requests, 4 test runs where made with an increasing number of nodes. The results can be found in Table 7.3. What can be noticed is that at this workload, Jetty still is faster. However when increasing the workload to 200 parallel requests with 4 DOH nodes, DOH has an average response time of 135ms compared to Jetty's 171.

Second scenario
The second test scenario is larger in scale: 18 domains, 47 folders, and 503 files (mean size is still 10Kb). The idea here is to evaluate how DOH would perform in a real life scenario. Runs where made with 1 - 5 nodes and with workloads of 10, 50, and 100 parallel requests. Also, for reference, Jetty was run with this new file set on workloads of 10, 50, and 70. Figure 7.4 shows the response times for 1 - 5 nodes under different workloads and compares it with Jetty. As can be seen, DOH is slower then Jetty for all the workloads that was tested. Due to a bug, the DKS version had to be changed between the tests.

Nodes	**Response times**
1	120
2	120
3	110
4	110

Table 7.3: The average response time for DOH, when the workload is 100 parallel requests.

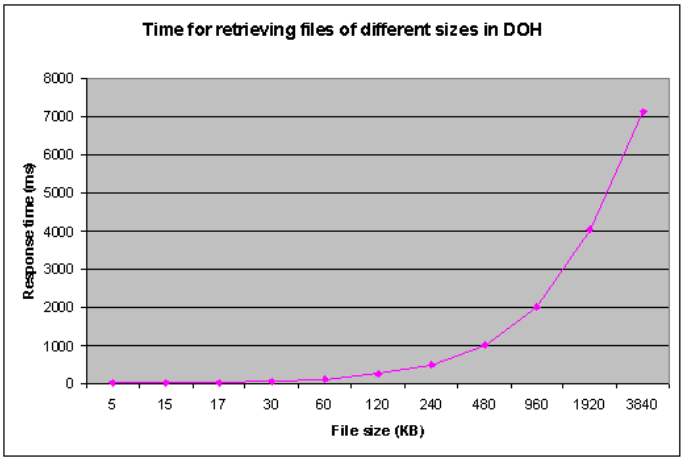

Figure 7.2: Time to retrieve files of different sizes in DOH.

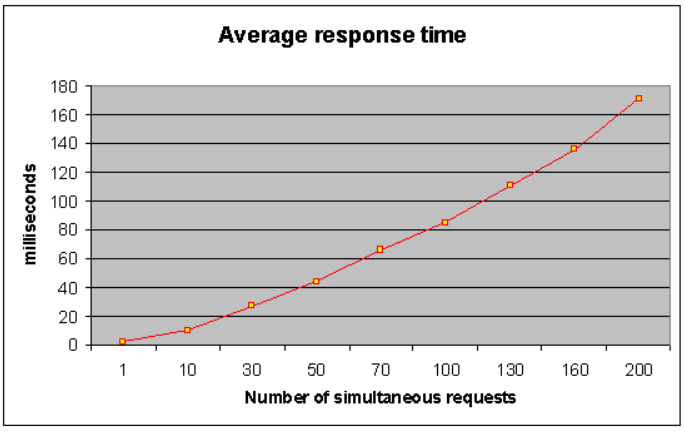

Figure 7.3: The performance of Jetty under different workloads.

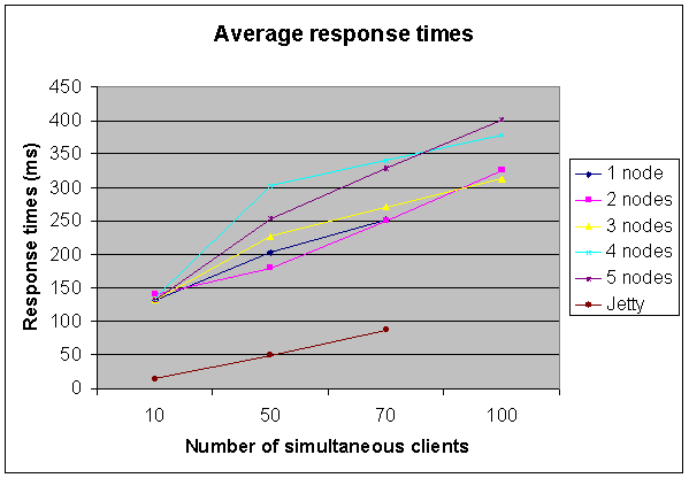

Figure 7.4: The average response time for DOH under different workloads, for a number of nodes. Jetty is included for comparison reasons.

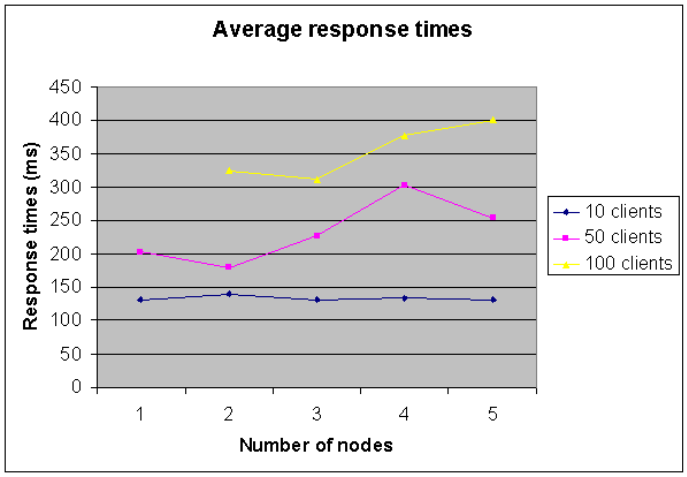

Figure 7.5: The average response time for DOH under different workloads, for a number of nodes.

This is very unfortunate since no real comparisons can be made between the results in the different scenarios. For instance, in this scenario "Java out of memory"-errors where generated when workloads exceeded 70 parallel requests. The problem is that it can not be determined if this is because of the larger working set or because this version of DKS uses more memory. Furthermore, it can not be stated that the slower response times is due to the larger file set, but might be the because the new "beta" DKS version extensively writes to System.out. However, an excerpt of the results from this testing scenario is shown in Figure 7.5. What could be a pattern, is that all the graphs in the figure seems to have a minima, i.e. for all tested workloads there is an "ideal" number of nodes where the response time is smallest. During these testing scenarios the need for a less complex and more controllable environment has grown stronger. There are simply to many other factors that interferes with what we want to measure. It also would be of interest to control the number of requests per second (request rate), instead of the number of parallel requests (workload). Then further examination could be done regarding the results that suggests that there exist an "ideal" number of nodes for each workload. For these reasons, a model was developed and a simulator implemented to continue the testing in a more fine-grained and controlled manner.

7.4.1 Model description

A simulation model of DOH was implemented as well, using the lessons from the earlier described tests as a base. The model is supposed to be an accurate description of the DOH system, with respect to the user-perceived latency. Indata will be requests and outdata will be system statistics such as number of requests, average load, total time, cache hit ratio, and most important average service time i.e the user-perceived latency.
When creating a model of the system, some assumptions of system behavior is often needed and the DOH Simulation model is no exception. The first assumption is that all nodes have the same performance and capacity. The arrival rate of the requests is then assumed to be exponentially distributed, and furthermore the generated requests are assumed to be equally distributed between the nodes (because of the Translator). For determining which file that is requested it is shown in [19] that a heavy-tailed distribution should be used, and therefore the Zipf distribution is used in the model for that purpose. File sizes in the model will be roughly uniformly distributed, with a few files that will be much larger than the mean while most of the files are smaller. This is not entirely true to reality but will suffice for our evaluation needs.

Since the earlier tests shows that caching is an important feature for the system performance, the simulation model needs to have that feature as well.

The three factors that determines the service time of a request are the current load of the server, the size of the requested file and if the file is cached or not. To get accurate timing characteristics, the results from Table 7.1 and the results from the Jetty performance test, Figure 7.3, was used.

Both these tests produces linear functions where the time in milliseconds can be obtained from the size of the file or the workload. Since the simulator implements a cache a function for calculating service times for requests under

different work loads and file sizes could now be created:

$$ST = load * 0.85 + 2 + cM * (2 * fileSize + 15 + H * (\log_2(N) * 100))$$

where **ST** is the service time in milliseconds, **load** is the number of parallel requests served, **cM** $\in \{0, 1\}$ is a cache miss or hit, **fileSize** is the size of the file in kilobytes, **H** $\in \{0, 1\}$ is used to determine whether the file is stored locally or if a number of hops is needed to find it. The probability that H = 1 is $\frac{f}{N}$, where **f** is the fault tolerance parameter, **N** is the number of nodes in the system, and the average number of hops is used. (See Section 2.3.2 for further explanations of the DKS parameters.)

7.4.2 Simulator description

From the model a simulator was created that implements the model. The simulator has two important classes: the Dispatcher class and the Node class. The Dispatcher class contains the main method, a list of Node objects, and the global clock. Each Node object have a list of events and furthermore all nodes have the same list of generated pseudo files. Each simulated time step is assumed to be one millisecond, and follows the following algorithm:

1. Generate a number of requests.
2. Distribute them among the nodes.
3. For all nodes: evaluate the events for this time step.
   (a) For all events:
      i. Calculate which file that should be requested.
      ii. Calculate service time.
      iii. If not cached ⇒ cache.
4. Increase time with one.
5. When simulation clock equals end time ⇒ finish the requests that are in the system and then calculate statistics.

Of course the load will be increased with one for all incoming requests and decreased with one every time the request is served. All requests are assumed to be taken care of as soon as they reach the node. Therefore there will be no waiting time, but the service time however, will increase with the load as explained earlier. Regarding the caching, two different simulators actually where implemented: one for file-wise retrieval and caching and one for both directory- and site-wise. Thus the different approaches can be compared in the simulator as well.

7.4.3 Results

Since Jetty has been tested under different workloads, it will be the baseline when evaluating our results. As shown in Figure 7.3 the response time is describing a linear function during different loads. This is used in the simulator to improve the accuracy of the calculated service time, as described earlier, and also it will be used for comparison with DOH.

In the first test scenario, when performing 200 parallel requests, the average service time for Jetty was 171 milliseconds, which equals a rate of 1170 served requests per second. (It can also be noted that the CPU of the machine running the server had almost 100% load starting from 70 parallel requests, but that the server still only shows linear degrade in performance from that point and up to the 200 requests tested.) This was the starting point for the first round of simulation: the request rate was chosen to be 1170 requests per second and then the number of nodes was increased.

The in-parameters for the simulation is chosen as follows: the request rate is 1170 requests/s as stated earlier; the number of files in the system is 1000 (in the case of the directory-wise approach: 100 directories containing 10 files each); the average file size is 30Kb (concurring with the average file size of the web, as showed in [8]); simulation time is 100 seconds and TTL in the cache is 3 seconds.

The results are promising and are shown in Figure 7.6 for the directory-wise approach and in Figure 7.7 for the file-wise approach. In fact, at this rate, using two DOH nodes gives the Users better performance results, then using a regular Jetty server. After performing these tests and a number of others with request rates in the interval between 1000 - 3000 requests per second, it can be determined that for each request rate there is a "minimum" on the graph representing the ideal number of nodes for just that rate. This minimum represents the point where the average cache hit ratio for the whole network is the highest, for request rates that is in the range of 500-750 requests per node and second, see Table 7.4. When increasing the TTL in the cache it shows that the directory-wise approach benefits more then the file-wise, and with TTLs over 30 seconds, the directory-wise is faster. Also, as can be seen in Table 7.4, changing the TTL in most cases wont change the ideal number of nodes. Furthermore it can be noted that when the network increases in size, for the tested ratios, the directory-wise approach outranks the file-wise even when using small TTL values. Increasing the number of files in the system will not change the ideal number of nodes, it will however increase the service times.

After these tests, the main conclusion is the same as the one in [41]: there is not one single caching strategy that is best for every situation.

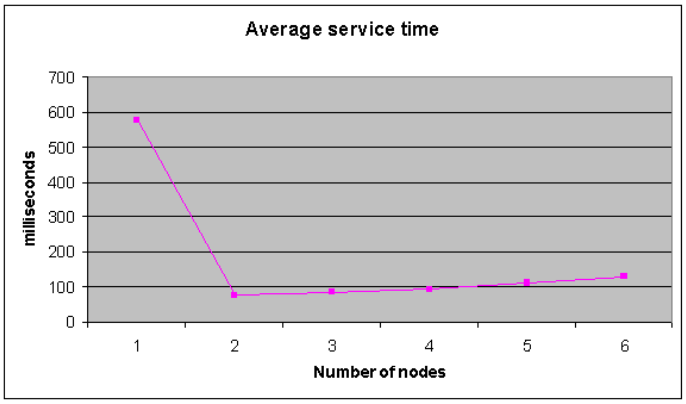

Figure 7.6: The average service time in DOH, with 1170 requests per second, retrieving files directory-wise

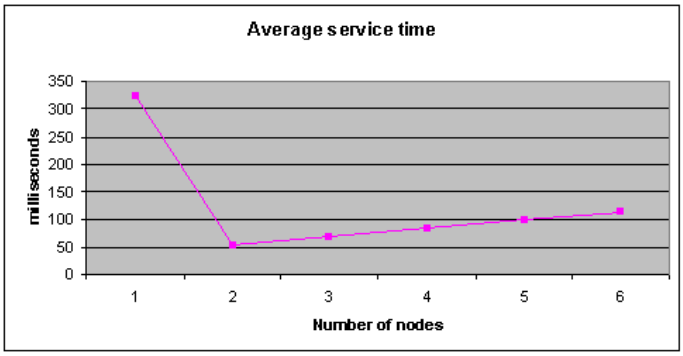

Figure 7.7: The average service time in DOH, with 1170 requests per second, retrieving files with the file-wise approach.

Directory-wise approach					
RR	**TTL**	**N**	**ST**	**CR**	**R/N**
1000	3%	2	80	94%	500
1000	30%	2	14	99%	500
1000	60%	2	9	100%	500
1000	100%	2	5	100%	500
1500	3%	3	84	94%	500
1500	30%	2	14	99%	750
1500	60%	3	8	100%	500
1500	100%	3	6	100%	500
2000	3%	3	83	95%	667
2000	30%	4	15	99%	500
2000	60%	4	9	100%	500
2000	100%	4	6	100%	500
2500	3%	5	88	94%	500
2500	30%	4	15	99%	625
2500	60%	5	9	100%	500
2500	100%	5	6	100%	500
3000	3%	5	93	95%	600
3000	30%	5	15	99%	600
3000	60%	5	9	100%	600
3000	100%	6	6	100%	500

File-wise approach					
RR	**TTL**	**N**	**ST**	**CR**	**R/N**
1000	3%	2	50	70%	500
1000	30%	2	14	93%	500
1000	60%	2	10	96%	500
1000	100%	2	6	98%	500
1500	3%	2	63	76%	750
1500	30%	2	16	95%	750
1500	60%	2	11	97%	750
1500	100%	2	8	99%	750
2000	3%	3	80	75%	667
2000	30%	3	19	95%	667
2000	60%	3	13	97%	667
2000	100%	3	8	99%	667
2500	3%	4	94	73%	625
2500	30%	4	23	95%	625
2500	60%	4	15	97%	625
2500	100%	4	9	98%	625
3000	3%	4	106	76%	750
3000	30%	5	25	94%	600
3000	60%	5	17	97%	600
3000	100%	5	10	98%	600

Table 7.4: The ideal number of nodes for different request rates. Where **RR** is the request ratio per second, **N** is the ideal number of nodes, **ST** the service time, **CR** the cache hit ratio, and **RR/N** the request ratio per node and second. The number of files in the system are 1000 for both approaches, however in the case of the directory-wise they are divided to 100 directories containing 10 files each).

8 Conclusions

In this report DOH has been presented as a way of creating a CDP2PN, i.e. an inexpensive way of protecting your web server against a flash crowd[1]. The system is based on DKS[4], which is an overlay network developed at KTH[33]. Each node in the DOH network consists of a web server, an FTP server, and is also part of the DKS overlay network. Furthermore a Translator has been introduced that translates and reroutes HTTP-requests to DOH nodes. It also keeps a cache of nodes that are known to be in the system, for solving the bootstrapping issues that comes with P2P systems and for Publishers to find nodes to login to, when uploading content. The idea of DOH is to store files in a DHT so each node is able to retrieve the files requested and cache them locally for future requests. Thus when a sudden traffic surge occurs, there will not only be one server serving all the requests but a network of cooperating web servers helping each other to divide the load.

In order to test these ideas a prototype, realizing the design (except for the DNS-server part of the Translator), was implemented in Java[54]. Several tests has been run to validate the prototype against the proposed design, including: validation of the fairness of the Translator's rerouter; adding Publishers to the system; uploading content into the DHT using the node's FTP server; browsing the uploaded content with a regular browser; deleting the content; and deleting Publishers. According to these validation tests the prototype fulfills the requirements stated in the use cases of the proposed design.

The prototype was also evaluated performance-wise, to see if DOH has the performance to function in a real life scenario. The first test looked at retrieval time versus file sizes, and it shows that in that aspect, the performance of DOH depends heavily on the performance of DKS: over 90% of the time used by the system comes from DKS-related activities when file sizes is increased to 4Mb. Furthermore it can be stated that while DOH is not fast, it is neither to slow to be used. When workloads (number of parallel requests) are low Jetty (the stand-alone web server used for comparison) outperforms DOH, but when the traffic increases DOH's response times will not increase as fast as Jetty's. E.g. at 100 parallel requests, the average response time for Jetty is 87ms compared to 110ms for DOH. However at 200 parallel requests, the average response time for Jetty is 171ms compared to 135ms for DOH.

The simulator suggests that DOH will perform well, with response times smaller then 300ms, at request rates smaller than approximately 1200 requests per second and node. Assume that 100 000 users decide to visit a site in a period of 5 minutes. Further assume that they will produce an average of 10 requests each. This will give the system 1 million requests in 300 seconds, or a request rate of 3333 requests per second. Then 3 DOH nodes should be able to handle that traffic surge, an increase in traffic that would cause a huge degeneration of performance in Jetty. DOH would be able to handle a flash crowd, however the price the users of the system would have to pay is that the page retrieval under normal workloads would be slower.

Three different approaches for file storage and retrieval has also been

evaluated, to see which is the best: file-, directory-, and site-wise. Unfortunately, the results show that there is no clear candidate to use in all cases. If web pages are changing rapidly or if the load of the network is small, then the file-wise approach yields the best results. If there seldom are changes in the stored sites or the load is higher, then the directory-wise or in some cases even the directory-wise approach is the best to use. This suggests that the system would perform even better if there was a way of using at least two of the approaches at the same time, and DKS indeed supports this kind of flexibility (see Section 9 for more information on this.)

Finally a comparison of DOH against a stand-alone web server was performed. Or rather, the web server was tested under different workloads and when it started to degrade in performance the response times at that ratio (1170 requests per second) was compared with the response times of DOH with different number of nodes. It was found that there exist an ideal number of nodes for each request ratio, where there is a minimum on the performance graph. E.g. a request rate of 1500 requests per second will yield its best performance with 3 nodes in the system (with an average of 14ms in response time), while a request rate of 3000 requests per second has its minima with 5 nodes in the system (15ms). This minimum occurs when the cache hit ratio is largest, which suggests that even more effort should be used to find an even better caching strategy. Since DOH already has a virtual file system, it should be fairly easy to add information that can be used for caching decisions, such as frequency of modification and version of the file.

When it comes to memory and CPU used, the single most important factor to decrease these factors is the cache. When not using a cache, the CPU will have to work very hard even at rather small request rates and the memory consumed is 25% higher. However, when using a cache and fragmenting files when inserting them into the DHT, the memory required is not a big issue for most of todays web servers. The CPU on the other hand will have quite a high workload when the web server starts to experience higher loads, but that is inevitable and depends heavily on the web server chosen, i.e Jetty[27].

9 Future work

There are two different types of future work for the DOH system. The first concerns enhancing the existing functionality with the knowledge gained from this project, and the second is to add functionality to the system.

Enhancing existing functionality
The functionality that the system should benefit the most from is creating a dynamic approach for choosing how to retrieve files. DKS supports a lookup that will only retrieve the specified object from the DHT. Thus one can hash on directories and when the load is small, instead of downloading the whole directory one can choose to only download the requested file. Furthermore the caching scheme could be extended to take into account how often files change when deciding TTL, and nodes might also probe the DHT for finding out if they have the same object instead of downloading it blindly.

As stated before, the caching algorithm should also be improved for the system to enhance its performance further. When looking to improve the caching scheme, maybe the use of an invalidation scheme should be considered as well. DKS supports the use of broadcasting messages, which might be used for this reason. On the other hand, Akamai[2] uses a versioning scheme in the URLs to determine which is the latest version of a page. Since the URLs already plays an important role when finding objects in DOH, this might be added to our system as well, without the need of changing the design drastically.

The next thing to improve for the system to be able to work in a real environment is the Translator. It should also be implemented as a node in DKS, then it could use the information of DKS's routing tables to see which nodes that exist, and no communication needs to be done between the Translator and the DOH-nodes except for load information. Also implementing the DNS part of the Translator would provide the system with the benefits of the DNS rerouting approach described in Section 2.4.1.

Adding new functionality
Research is now being done on how to use CDNs with dynamic content, e.g. [48], and that could be the next step for DOH as well. As stated earlier, that was one of the reasons for using an already existing web server: to facilitate the use of dynamic content in later versions of the system. The DKS DHT could still be used for storage, if a middle-layer is created that has some support for SQL syntax (or something similar). For such a system, questions about transactions, states and how to handle failures becomes very important. Take the example of someone wanting to purchase something from an e-commerce web site that uses DOH 2.0 which supports dynamic content. What will happen if a node crashes during a session? How do you store the state to achieve failover? Making sure that the system supports the ACID properties of a transaction is a must, but how should that be achieved? Logging the user's activity, and implementing a two-phase commit protocol are ways of dealing with these issues, but how do they effect the performance?

Furthermore, when deploying your application into such a system, what scheme should you use to avoid namespace collisions, to know what to run, and making sure that the applications not is harmful for the node running them. Meta-info about the application has to be provided to the system, which can be done using e.g. an XML-scheme. Looking at e.g. JavaBeans[55] might be a good starting point to see how deployment of applications is done.

Also security issues needs to be addressed. As DOH now is implemented, there is no way for the owner of a node to limit the resources that the system uses. This concerns both CPU, memory and disc space. The next step would be to at least have the functionality for node owners to be able to assign a disc quota that can be used by DOH. Also one should consider to protect the cached content as well and try to find a tradeoff between how secure the replicated objects in the node cache needs to be against the fact that they need to be quickly accessed by the system. Furthermore the user management in the

prototype needs to be reviewed, and how to get it secure is quite a big challenge as with all distributed systems. Who should be responsible for adding and deleting users, and how should they be authenticated? Remember that anyone can start a node that joins the DOH network, and as it is now the person that starts the first node has all the administrator privileges.

10 References

[1] S. Adler, "The Slashdot Effect: An Analysis of Three Internet Publications" [Online] http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html, 2005.

[2] Akamai Technologies, Inc. [Online] http://www.akamai.com/, 2005.

[3] L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi, "A framework for peer-to-peer lookup services based on k-ary search." Technical Report TR-2002-06, (SICS), May, 2000.

[4] L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi, "DKS(N, k, f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications", In *The 3rd International workshop CCGRID2003* (Tokyo, Japan), May 2003.

[5] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris, "Resilient Overlay Networks," In *18th ACM Symposium on Operating Systems Principles (SOSP)*, **p. 131-145**, October, 2001.

[6] [Online] http://www.apache.org/, 2004.

[7] [Online] http://www.apache.org/licenses/LICENSE-2.0, May, 2005

[8] M. F. Arlitt and C. L. Williamson. "Internet web servers: Workload characterization and performance implications." In *IEEE/ACM Transactions on Networking*, **5(5):631-645**, October 1997.

[9] [Online] http://jakarta.apache.org/avalon, May, 2005

[10] A. Barbir, B. Cain, F. Douglis, M. Green, M. Ho.man, R. Nair, D. Potter, and O. Spatscheck, "Known CN Request-Routing Mechanisms", RFC3568, IETF, July, 2003. [Online] http://www.ietf.org/rfc/rfc3568.txt

[11] R. Bhattacharyya [Online] http://www.mycgiserver.com/řanab/ftp/, May, 2005.

[12] BitTorrent, Inc. [Online] http://www.bittorrent.com/

[13] Borland, JBuilder X Enterprise. [Online] http://www.borland.com/jbuilder/, May 2005.

[14] Borland, Borland® Optimizeit™ Enterprise Suite 6. [Online] http://www.borland.com/optimizeit, Sept 2005.

[15] Y. Chen, L. Qiu, W. Chen, L. Nguyen, and R. H. Katz, "Clustering Web Content for Efficient Replication", In *10th International Conference on Network Protocols, IEEE Computer Society Press,* (Los Alamitos, CA.), November, 2002.

[16] Y. Chen, R. Katz, and J. Kubiatowicz. "SCAN: A dynamic, scalable, and efficient content distribution network." In *Proceedings of the International Conference on Pervasive Computing* (Zurich, Switzerland), August, 2002.

[17] M. Colajanni, P. S. Yu, and D. M. Dias. "Analysis of task assignment policies in scalable distributed Web-server systems." In *IEEE Transactions on Parallel and Distributed Systems*, **9(6):585-699**, 1998.

[18] A. Crespo and H. Garcia-Molina. "Semantic Overlay Networks for P2P Systems." Technical report, (Stanford University), October, 2002.

[19] M. E. Crovella, M. S. Taqqu, and A. Bestavros. "Heavy-tailed probability distributions in the World Wide Web." In *A Practical Guide To Heavy Tails*, **p.3-26**, Chapman & Hall, New York, 1998.

[20] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and Ion Stoica. "Wide-area cooperative storage with CFS". In *SOSP* (Banff, Alberta, Canada), October 21-24 2001.

[21] J. Dilley, Maggs, B., Parikh, J., Prokop, H., Sitaraman, R., AND Weihl, B. "Globally Distributed Content Delivery." *IEEE Internet Computing 6 5* **p.50-58**, September, 2001.

[22] D. Eastlake and P. Jones. "US Secure Hash Algorithm 1 (SHA1)", RFC 3174, 2001. [Online] http://www.ietf.org/rfc/rfc3174.txt

[23] H. Eriksson, "MBone: The Multicast BackBone," Communications of ACM, vol.37, no. 8, **p.54-60**, August, 1994.

[24] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. "Hypertext Transfer Protocol - HTTP 1.1", RFC2616, Network Working Group, June 1999. [Online] http://www.ietf.org/rfc/rfc2616.txt

[25] M. J. Freedman, Freudenthal, E., and Mazi'eres, D. "Democratizing Content Publication with Coral." In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI 2004)* (San Francisco), March, 2004.

[26] Michael Freedman and David Mazi'eres. "Sloppy hashing and self-organizing clusters." In *2nd International Peer To Peer Systems Workshop* (Berkeley, CA, USA), February, 2003.

[27] Mortbay Consulting. [Online] http://jetty.mortbay.org/jetty/index.html, May, 2005.

[28] Ali Ghodsi, Luc Onana Alima, Seif Haridi. "Symmetric Replication for Structured Peer-to-Peer Systems",In *The 3rd International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, (Trondheim, Norway), August, 2005.

[29] Gnutella, [Online] http://www.gnutella.com, May, 2005.

[30] N.J.A. Harvey, M. Jones, S. Saroiu, M. Theimer, and A.Wolman. "Skipnet: A scalable overlay network with practical locality properties." In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, March, 2003.

[31] C. Huang, S. Sebastine and T. Abdelzaher, "An Architecture for On-Demand Active Web Content Replication", In *16th Euromicro Conference on Real-Time Systems*, (Catania, Italy), July, 2004.

[32] S. Iyer, A. Rowstron, and P. Druschel. "Squirrel: A decentralized, peer-to-peer web cache." In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing.* ACM, July, 2002.

[33] [Online] http://www.kth.se/, 2005.

[34] A. D. Keromytis, V. Misra, and D. Rubenstein. "SOS: Secure Overlay Services" In *Proceedings of ACM SIGCOMM*, **p.61-72**, August, 2002.

[35] I. Lazar and W. Terrill. "Exploring content delivery networking." In *IT Professional*, **3:47-49**, July-August, 2001.

[36] P. Maymounkov and D. Mazi‘eres. ”Kademlia: A peerto- peer information system based on the xor metric.” In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS02)* (Cambridge, MA), March 2002.

[37] MySQL AB, [Online] http://www.mysql.com, 2005.

[38] Open Content Network, [Online]http://open-content.net/, 2005

[39] The PHP Group, [Online] http://www.php.net, 2005.

[40] G. Pierre, and M. van Steen ”Design and implementation of a user-centered content delivery network.” In *Proc. 3rd Workshop on Internet Applications*, (San Jose, CA), 2003.

[41] G. Pierre, M. van Steen, and A. S. Tanenbaum. ”Dynamically selecting optimal distribution strategies for Web documents.” *IEEE Transactions on Computers*, **51(6):637–651**, June 2002.

[42] J. Postel and J.K. Reynolds. ”File Transfer Protocol”, RFC959, October 1985. [Online] http://www.ietf.org/rfc/rfc959.txt

[43] M. Rabinovich and A. Aggarwal ”RaDaR: A scalable architecture for a global Web hosting service.” In *The 8th Int. World Wide Web Conf*, May, 1999.

[44] S. Ratnasamy, P. Francis , M. Handley , R. Karp , S. Schenker, ”A scalable content-addressable network”, In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, **p.161-172** (San Diego, California, United States), August 2001.

[45] P. Rodriguez and S. Sibal, ”SPREAD: scalable platform for reliable and efficient automated distribution”, In *Proceedings of the 9th international WWW conference on Computer networks : the international journal of computer and telecommunications networking* **p.33-49**, (Amsterdam, Holland), June, 2000.

[46] A. Rowstron and P. Druschel, ”Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems”. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)* **p.329-350** (Heidelberg, Germany), November, 2001

[47] A. Rowstron and P. Druschel, ”Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility”, In *Proceedings of the eighteenth ACM symposium on Operating systems principles* (Banff, Alberta, Canada), October 21-24, 2001.

[48] S. Sivasubramanian, G. Alonso, G. Pierre, and M. van Steen. ”GlobeDB: Autonomic Data Replication for Web Applications.” In *Proceedings of the 14th International World-Wide Web Conference*, May, 2005.

[49] S. Sivasubramanian, G. Alonso, G. Pierre, and M. van Steen. ”Web Replica Hosting Systems.” In *ACM Computing Surveys* **36(3)**, September, 2004.

[50] T. Stading, P. Maniatis, and M. Baker, ”Peer-to-peer caching schemes to address flash crowds,” In *Proceedings of IPTPS'02* (Cambridge, MA), March, 2002.

[51] SICS, [Online] http://www.sics.se/, 2005.

[52] I. Stoica , R. Morris , D. Karger , M. F. Kaashoek , H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications", In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications* **p.149-160**, (San Diego, California, United States), August 2001.

[53] Z. Su et al, "Correlation-based Document Clustering using Web Logs", In *Proc. of the 34th Hawaii International Conference On System Sciences (HICSS-34)*, 2001.

[54] Sun Microsystems. Java 2 Platform, Standard Edition (J2SE). [Online] http://java.sun.com/j2se/, May 2005.

[55] Sun Microsystems. JavaBeans [Online] http://java.sun.com/products/javabeans/reference/index.html, November 2005.

[56] UML, OMG, [Online] http://www.uml.org, 2005.

[57] A. Vakali and G. Pallis. "Content delivery networks: Status and trends." In *IEEE Internet Computing* **7(6):68-74**, December, 2003.

[58] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. "Tapestry: An infrastructure for fault-tolerant wide-area location and routing." Technical Report UCB/CSD-01-1141 (UC Berkeley) April, 2001.

[59] W. Zhao and H. Schulzrinne. "DotSlash: A selfconfiguring and scalable rescue system for handling web hotspots effectively." In *International Workshop on Web Caching and Content Distribution (WCW)*, (Beijing, China), October 2004.

A Acronyms

List of used acronyms in thesis.

ACID Atomicity, Consistency, Isolation, and Durability
API Application Programming Interface
AS Autonomous Systems
BGP Border Gateway Protocol
CDN Content Delivery/Distribution Network
CDP2PN Content Delivery Peer-to-Peer Network
CLI Command Line Interface
CPU Central Processing Unit
DHT Distributed Hash Table
DNS Domain Name Service
DOH DKS Organized Hosting
DKS Distributed K-ary Search
DSHT Distributed Sloppy Hash Table
GUI Graphical User Interface
HTML Hyper Text Markup Language
HTTP Hyper Text Transfer Protocol
ID IDentifier
IP Internet Protocol
ISP Internet Service Provider
LAN Local Area Network
LRU Least Recently Used
P2P Peer-to-Peer
PHP PHP Hypertext Preprocessor
PKI Public Key Infrastructure
RTT Round-Trip Time
SHA-1 Secure Hashing Algorithm 1
SICS Swedish Institute of Computer Science
SQL Structured Query Language
TCP Transmission Control Protocol
TTL Time To Live
URL Uniform Resource Locator
XML eXtensible Markup Language
WWW World Wide Web

B Rerouter validation

Symmetric runs

```
5 nodes, 100 000 requests, 1 iteration:

Node: 0.0.0.3, number of hits: 19992 max: 20106
Node: 0.0.0.2, number of hits: 20034 min: 19897
Node: 0.0.0.1, number of hits: 20106 median: 19992
Node: 0.0.0.0, number of hits: 19897 max-min interval: 209
Node: 0.0.0.4, number of hits: 19971


5 nodes, 100 000 requests, 10 iterations:

Node: 0.0.0.3, number of hits: 19987  max: 20065
Node: 0.0.0.2, number of hits: 20065 min: 19947
Node: 0.0.0.1, number of hits: 20004 median: 19995
Node: 0.0.0.0, number of hits: 19995 max-min interval: 118
Node: 0.0.0.4, number of hits: 19947


5 nodes, 100 000 requests, 100 iterations:

Node: 0.0.0.3, number of hits: 20014  max: 20018
Node: 0.0.0.2, number of hits: 19991  min: 19975
Node: 0.0.0.1, number of hits: 20000  median:20000
Node: 0.0.0.0, number of hits: 20018 max-min interval: 47
Node: 0.0.0.4, number of hits: 19975


5 nodes, 100 000 requests, 1 000 iterations:

Node: 0.0.0.3, number of hits: 20001 max: 20001
Node: 0.0.0.2, number of hits: 20001 min: 19998
Node: 0.0.0.1, number of hits: 19998 median: 19999
Node: 0.0.0.0, number of hits: 19999 max-min interval: 3
Node: 0.0.0.4, number of hits: 19999
```

Asymmetric runs

```
Start node scores:

Node 1: 1000
Node 2:  500
Node 3:  250
Node 4:  250

4 nodes, 2000 requests, 1 iteration:

IP: 4, number of hits: 250
IP: 3, number of hits: 250
IP: 2, number of hits: 500
IP: 1, number of hits: 1000

4 nodes, 3000 requests, 1 iteration:

IP: 4, number of hits: 498
IP: 3, number of hits: 497
IP: 2, number of hits: 746
IP: 1, number of hits: 1259


4 nodes, 3000 requests, 10 iterations:

IP: 4, number of hits: 505
IP: 3, number of hits: 495
IP: 2, number of hits: 747
IP: 1, number of hits: 1252


4 nodes, 3000 requests, 100 iterations:

IP: 4, number of hits: 501
IP: 3, number of hits: 501
IP: 2, number of hits: 748
IP: 1, number of hits: 1248


4 nodes, 3000 requests, 1000 iterations:

IP: 4, number of hits: 499
IP: 3, number of hits: 499
IP: 2, number of hits: 750
IP: 1, number of hits: 1250
```

C DOH Manual

C.1 Starting a Translator

The translator can be started in two different ways. First if you do not have an old cache file, you might start by giving it parameters or use the default parameters. If an old cache file exist, the translator will use that file when it boots. Syntax:

java -jar DOHTranslator.jar [-h (help) | -url (url) -tp (port of translator) -uptime (uptime) -v (version) -p (ping) -score (cache score) -t (list of other translators) -wp (port of web cache)]

The Translator will start two listeners on ports 4711 (the default port for the web cache) and 8080 (the default port of the translator, which can be changed using the e.g. the `url` parameter) When started and ready, the Translator window will look something like this:

```
>java -jar DOHTranslator.jar
INFO: Could not find any old cache file...
initializing new cache using default values!!
Translator ip: 192.168.50.182
Translator url: http://localhost:8080
Uptime: 12
Version: 1
Score: 110
DOH Web Cache is running!
```

There is a shutdown hook added to the Translator, so when the program is exited by the user, using e.g. Control-C, it will save the current cache in a file called `doh_webcache.xml`. This file will be used at the next boot, if it is not removed. However, all the cache entries and the IP-address of the Translator are discarded since they might no longer be correct.

C.2 Starting a DOH Node

Now when there is a Translator to connect to, it is time to start the first DOH Node. At first, some configuration is needed. The configuration file has the following parameters:

host The DNS name if the computer hosting the node.

port The web server's port, if not set default (80) is used.

home The web server's root directory.

dks_port The DKS node port, if not set default (4440) is used.

ftp_conf The path to the FTP configuration file.

And it might look something like this:

```
host=computer.company.com
port=80
home=./wwwroot/
dks_port=4440
webcache_url=http://doh.translator.org:4711
ftp_conf=./ftpd.conf
```

NB: the separator should be an equals sign, or the startup might fail.
The FTP configuration file, `ftpd.conf`, must also be updated with host ip and home directory. How is explained in the file itself and therefore not included here.

When the configuration is done, it is time to start the Node itself. This can be done using the `DOHNode.jar`, or the `DOHNodeCLI.jar`. They both have the same syntax, the only difference is that `DOHNode.jar` has a minimal GUI. Syntax:

java -jar DOHNodeCLI.jar <configuration file>

If this is the first node that enters the system, you will be asked to provide an administrator password for the FTPs. When started and ready, the Node window will look like this:

```
INFO: Node details sent to Cache.
2005-sep-08 17:05:37 org.mortbay.util.FileResource <clinit>
INFO: Checking Resource aliases
2005-sep-08 17:05:37 org.mortbay.http.HttpContext setStatsOn
INFO: setStatsOn true for HttpContext[/,/]
FtpServer.server.config.user.manager =
ranab.server.ftp.usermanager.PropertiesUserManager
FtpServer.server.config.log.flush = true
FtpServer.server.config.ip.allow = false
FtpServer.server.config.data = ./wwwroot/apps/ftp/data/
FtpServer.server.config.prop.encrypt = true
FtpServer.server.config.anonymous.login = 10
FtpServer.server.config.anonymous = true
FtpServer.server.config.log.level = 1
FtpServer.server.config.self.host = computer.company.com
FtpServer.server.config.root.dir = ./wwwroot/
FtpServer.server.config.port = 21
FtpServer.server.config.home.create = false
FtpServer.server.config.admin = admin
FtpServer.server.config.idle.time = 60
FtpServer.server.config.poll.inverval = 60
FtpServer.server.config.login = 20
FtpServer.server.config.log.size = 1024
FtpServer.server.config.server.host = 198.162.5.33
Started FTP
2005-sep-08 17:05:38 org.mortbay.http.HttpServer doStart
INFO: Version Jetty/5.1.3rc4
2005-sep-08 17:05:38 org.mortbay.util.Container start
INFO: Started HttpContext[/,/]
2005-sep-08 17:05:38 org.mortbay.http.SocketListener start
INFO: Started SocketListener on 0.0.0.0:80
2005-sep-08 17:05:38 org.mortbay.util.Container start
INFO: Started org.mortbay.jetty.Server@110b053
********* NODE IS STARTED! **********
Write stop to stop it!
```

The only command that is supported by the Node CLI is stop, which will try to shutdown the node gracefully by informing the DKS network and the Translator that the node is shutting down.

C.3 User Management

When we have a node up and running it is time to add some users to the system so that content, eventually, can be uploaded. Currently, the FTP admin account is the only account that is allowed to add and delete users. This is also the only thing that can be done with the admin account, it can not be used to store files and folders.

This is how to add a new user: first create a file on your computer that is called `new_user&username&password` (where username is the username and password is the password of the new user); connect and login as admin on the node's FTP server; upload the file. (The content of the file does not matter, the system will look at the file name.) The FTP server will say that the file not has been uploaded, but instead there should be a new user added. To delete a user you do the same thing: create a file called `delete_user&username` (where username is the username of the user that should be deleted); login as admin; upload the file. When a user is deleted all of his stored files are deleted as well.

This might not be the best and most secure way of adding and deleting user accounts, but it is sufficient enough to be used with the prototype.

C.4 To publish content

The implementation of the DOH FTP server and the fact that uploaded files are stored in a DHT instead of a regular file system requires two things from the FTP clients used. The DOH FTP server is session oriented and a copy of the user's virtual files are retrieved from the DHT during the login, which marks the session start. During the session any changes of the user's files are recorded locally and the virtual files in the DHT are not updated until the session ends. Since the DOH FTP server is session oriented in this way, it will not work well with FTP clients using multiple connections. Furthermore it will not work well with FTP clients that omit the FTP command "QUIT" when disconnecting from a server, since that command is being used by the server to mark the end of a session. (See Section 5.2.2 for all the implementation details of the DOH FTP server.)

When you have found an FTP client that supports this, and have a registered account, then you should upload the content that are to be published on any of the DOH Node's FTP server. (Node IP's can be found be contacting a translator and requesting the file `doh_webcache.xml`, where all the Translators active nodes are displayed.) After that you make sure that there is a redirection from the page's old location to the Translator. Using e.g. JavaScript or by adding a CNAME record to the authorative DNS server. The JavaScript-based redirect file might look like this:

```
<HTML>
<HEAD>
<SCRIPT type="text/javascript">
<!- -
function delayer(){
var oldURL = document.location.href

// using substring to remove http:// from original url
document.location = "http://my.translator.com/" + oldURL.substring(7,oldURL.length)
}
//- ->
</SCRIPT>
</HEAD>

<!-  - 3000 is the number of millisecs to wait before redirecting -  ->
<BODY onLoad="setTimeout('delayer()', 3000)">
<H3 >This page is hosted on a DOH Network!</H3>
<P>Please wait until you are redirected</P>
</BODY>
</HTML>
```

(That script can be found in the file `doh.html`, which comes with the DOH package. If you do so, do not forget to change the URL of the Translator.)

One more restriction is imposed on the Publisher from the system. The top level directory of the published site must be named after the domain name. E.g. if the site's name is `www.url.com` then the top level directory should also be named www.url.com. (See Section 5.2.2 for details.)

Publisher checklist:

1. Get an account.
2. Rename your top level directory to the domain name of the site you are about to upload, e.g `www.url.com`.
3. Find a Node to upload to, by e.g contacting a Translator and requesting the file `doh_webcache.xml` (E.g. http://a.translator.com/doh_webcache.xml)
4. Upload content.
5. Add a redirection from your page's old destination to a Translator. Use e.g. the JavaScript found in `doh.html`.