Free University Amsterdam
Faculty of Sciences
Warsaw University
Faculty of Mathematics, Informatics and Mechanics

Michał Szymaniak
Student number: 1289004

A DNS-based Client Redirector for the Apache HTTP Server

Master's Thesis
Specialization: COMPUTER SCIENCE

Supervisors (Free University):
Prof. Maarten van Steen
Dr. Guillaume Pierre

Supervisor (Warsaw University):
Dr. Janina Mincer-Daszkiewicz

June 2002

This thesis is ready to be marked.

Date: Author's signature:

This thesis is ready to be verified by the second reader.

Date: Supervisor's signature:

Abstract

The client redirection problem is one of the most fundamental issues when developing a Content Delivery Network (CDN). This thesis discusses three known solutions and describes the implementation of one of them, which exploits a special DNS protocol implementation. The system consists of a special module for the Apache HTTP Server, which supports the basic functions of a typical DNS server. The module generates custom DNS responses based on network distance calculations according to the AS-path length metric. The Internet topology information required to compute the AS-path length metric is derived from BGP routing tables. Performance measurements prove that the overhead introduced by the network distance calculations is negligible in comparison to the overall DNS request-reply delay.

Keywords:

Redirection, CDN, DNS, Apache, Autonomous Systems, BGP.

Contents

1. Problem description . . . 5

2. Redirection Mechanisms . . . 9
- 2.1. HTTP-based Redirection . . . 10
- 2.2. TCP Handoff . . . 12
- 2.3. DNS-based Redirection . . . 14
- 2.4. Discussion . . . 18

3. Implementation Details . . . 21
- 3.1. The Apache HTTP Server . . . 21
  - 3.1.1. Request Processing Model . . . 21
  - 3.1.2. Modules . . . 23
  - 3.1.3. Configuration . . . 23
  - 3.1.4. Apache Portable Runtime . . . 24
  - 3.1.5. Multi-Processing Modules . . . 24
  - 3.1.6. Internal Data Structures . . . 24
- 3.2. Redirector Design . . . 25
- 3.3. The Redirector as an Apache Module . . . 27
  - 3.3.1. Replica Database . . . 27
  - 3.3.2. Redirection Policy Implementation . . . 28
  - 3.3.3. DNS Protocol Implementation . . . 28
  - 3.3.4. Configuration Routines . . . 28
  - 3.3.5. DNS/TCP Support . . . 29
  - 3.3.6. DNS/UDP Support . . . 29

4. Redirection Policies . . . 33
- 4.1. Introduction . . . 33
  - 4.1.1. Policy Model . . . 33
  - 4.1.2. Static Name-to-Address Policy . . . 33
  - 4.1.3. Round-robin Policy . . . 34
- 4.2. AS-path Length Policy . . . 35
  - 4.2.1. Routing Basics . . . 35

4.2.2. Autonomous Systems . . . 36
4.2.3. Border Gateway Protocol . . . 37
4.2.4. Constructing a Map of the Internet . . . 39
4.2.5. Policy-based Routing . . . 40
4.2.6. IP-to-ASN Translation . . . 41
4.2.7. Final Algorithm of the AS-path Length Policy . . . 42
4.3. Policy Implementation . . . 43
4.3.1. General Concepts . . . 43
4.3.2. Simple Policies . . . 43
4.3.3. The AS-path Length Policy . . . 44

5. Performance Evaluation . . . 47
5.1. Performance of the DNS Transport Layer . . . 47
5.2. Micro-Benchmarks of the AS-path Length Policy . . . 48

6. Conclusion . . . 53

Bibliography . . . 55

Chapter 1

Problem description

With the continuous growth of the Internet there is also an increase in the need for replication of popular (and thus frequently-accessed) services. The existence of multiple copies of a service can increase its availability, as we reduce chances that none of the copies can be reached. More importantly, however, replication also aims at reducing the delay between sending a request to the service and having the answer retrieved. Since the actual data transfer accounts for about 80% of the total request time, accessing a nearby copy is expected to have a significant impact on performance [26]. The key problem here is how to select a service replica that is closest to a given client. Another problem that we need to solve is how to efficiently direct the client to the selected replica.

In this thesis we assume that a service has one home address. The home machine can redirect each client to a selected service replica that is likely to service it most efficiently. An important observation is that it is possible for the home machine to never run the service itself. Instead, it can execute only a redirection mechanism and each time let the selected replica do the main part of the job. As it will be shown in this thesis, the acceptance of this assumption makes the redirection problem a great deal easier.

The redirection problem can be split into the following two components: a redirection mechanism, and a redirection policy. The redirection mechanism (or, simply, the redirector) has three main duties. Firstly, it has to keep track of existing replicas, which can join or leave the group frequently. Each replica is described by a list of services it provides, thus allowing the redirector to isolate a subset of replicas supporting a specific service. The actual operations of inserting and removing replicas from the group are initiated by an external managerial component, for which an appropriate interface is provided.

Secondly, the redirector takes care of gathering information that can be useful for accurate replica selection. Both the type and the amount of the information depend on the redirection policy that is going to be used. It may include, for example, network load, processing capabilities of each replica, its availability, any QoS constraints it can meet, etc.

Thirdly, the redirector must react to incoming client queries. Each query contains a service identifier which is used to determine the subset of replicas supporting the service. Knowing this subset, the address of the client, and all the previously gathered information, we can apply a redirection policy to choose one or more "best" replica(s). Once they are known, the addresses of the respective replicas are sent back to the client, who can subsequently connect to any of the replicas to use the actual service.

While fulfilling the above three duties, the redirector has to adhere to a number of non-functional requirements. The most important one is that the redirection is transparent for the clients. As long as they are not aware of being switched between different replicas they can transparently access the service as any other, non-replicated one. With non-transparent redirection, references remain bound to a specific replica. Although some time later a different replica can become preferable for that client, it will keep on using the original one, which can result in performance loss. Worse yet, if this original replica has been removed from the set of replicas, the client can no longer use his reference. Instead, we want clients to be able to save references on replica A, and use them later, despite physically working with replica B this time.

Another requirement, which is somewhat related to transparency, is that the redirector makes use of network protocols that are understood by existing clients. In this way we ensure that our users do not need to change their software.

One more requirement is that the redirector is scalable – it can service a huge number of clients. Since the service has been replicated due to its popularity, the number of its clients is likely to reach extremely high values.

The last requirement is that the complete redirection mechanism is easy to deploy on a wide range of network architectures. To ensure portability, it has to be independent of both the underlying hardware and the network structure. Moreover, it cannot be too complicated, because it is going to be installed, configured, and maintained by people with possibly no detailed knowledge of computer networks.

Apart from the redirection mechanism we also need a redirection policy. It has a few important features as well. The most important one is that the method of selection is accurate. In general, it is difficult to predict which replica will service a given client most efficiently, as the network conditions and the load of replicas change continuously. Thus, in many cases, the policies will only provide approximations. The criteria used to evaluate quality of service depend on the service characteristics, and are usually hard to determine.

Another feature of the redirection policy is that it must be capable of selecting more than one replica for a given client, and rank them according to the predicted quality of service. By doing that the policy cooperates with the client-side redirection mechanisms, as they can still apply their own criteria of replica selection to the set of replicas chosen by the policy. Moreover, selecting several replicas increases the service availability (if the first replica is unreachable, it is still possible for the client to access another one).

Each result can also be associated with a lease period for which it is known to remain valid.

In this way we allow the results to be cached by the client, thus off-loading the redirector and reducing the time needed to obtain the service address.

Both the redirection mechanism and the redirection policy have to work fast. The complete redirection process must not generate significant overhead on the request-reply delay, as its main goal is to minimize it. Using non-local information or performing complex computations while servicing a client request is therefore prohibited.

We plan to use the redirector as part of another project of ours, called Globule [18]. Globule is a platform for self-replicating Web documents. It automates all aspects of document replication, such as creating and destroying replicas, and maintaining their consistency. Globule is designed as a module for the Apache HTTP server. To ensure that the redirection mechanism easily cooperates with other Globule components, we decided to implement it as part of Apache, too.

This thesis is structured as follows. In Chapter 2 we discuss advantages and drawbacks of three widely used redirection mechanisms. We compare them and show that DNS redirection comes closest to the objectives described above. In Chapter 3 we present our experience with implementing a DNS redirector inside Apache. Chapter 4 treats of redirection policies and different kinds of measures that can be useful for replica selection. Finally, Chapter 5 describes the efficiency tests we made, and Chapter 6 concludes.

Chapter 2

Redirection Mechanisms

Before the analysis of actual redirection mechanisms, let us first provide a context for further considerations by showing what actually happens when a client decides to contact a service. For purposes of this explanation we assume that the "service" is a Web page identified by its URL. In the following, we consider "the client" to refer to a Web browser and not the person controlling it.

Initially, the only item the client knows is the URL. Therefore, the first thing that it has to do is to resolve the host name encoded inside the URL into an IP address (see Figure 2.1). The name resolution is done on behalf of the client by its DNS server (a). The DNS server contacts intermediate DNS servers in order to locate the DNS server of the service (b, c). It eventually contacts this last server to determine the service address (d, e), which is subsequently returned to the client (f). We will return to a detailed description of DNS later.

When the service IP address is known, the client opens a TCP connection to it (see Figure 2.2). Initializing packets are sent by the client machine, then they are routed through the Internet and finally reach the gateway to the service network (a) and the service machine itself (b). As long as everything is in order, a new connection is established (c and d). From this moment on, all the communication between the client and the service machine follows the path shown in Figure 2.2.

Once the client has opened a network connection to the service machine, it can issue an HTTP request containing the URL of the service. The Web server on the service machine, in turn, receives the request, inspects it and returns the corresponding Web page. At this point the TCP connection is closed and the service usage ends. If necessary, the same steps are followed to retrieve embedded images, or the next page.

There are at least three places where the redirector can be installed: the service machine, the gateway to the service network and the service-side DNS server. In the following sections, we demonstrate the consequences of choosing each of them by taking a closer look at three redirection mechanisms [4]. We discuss how they fit into our initial requirements: transparency, scalability, maintainability, and efficiency. Finally, we compare them and show that DNS redirection fits the requirements best.

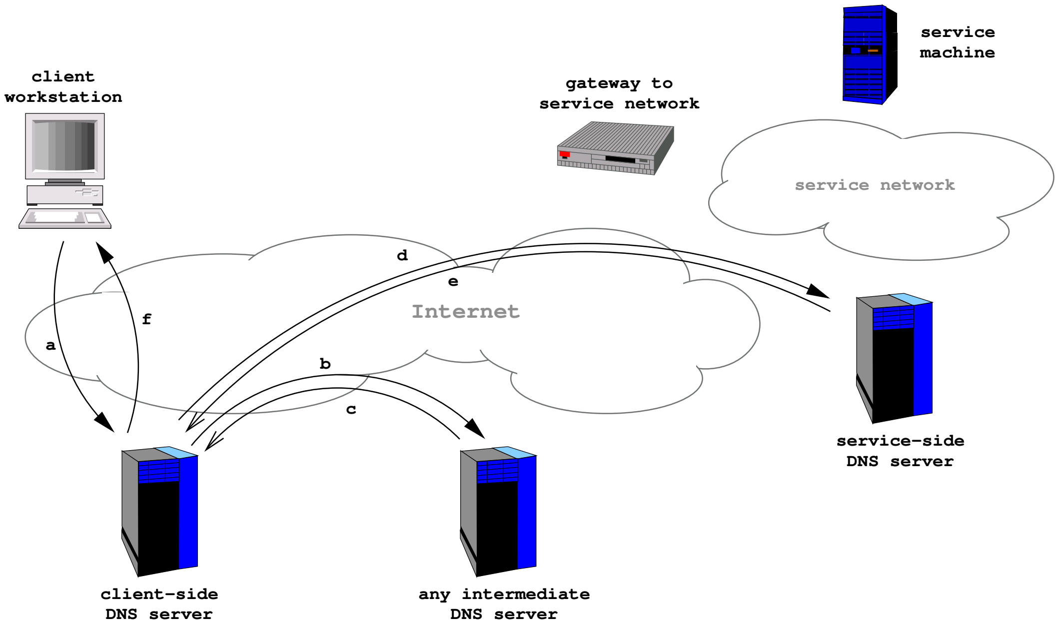


Figure 2.1: Name resolution

2.1. HTTP-based Redirection

The HyperText Transfer Protocol (HTTP) is an application-level protocol used mainly by Web servers and Web browsers [9]. It works as follows. Each document that can be transferred using HTTP is identified by means of a Uniform Resource Locator (URL). The URL contains the name of the Web server responsible for providing the document and usually also a server-internal path name. When a Web browser wishes to retrieve the document, it sends a request containing the URL to the server pointed to by the URL. The server inspects the request and returns the corresponding document. The document itself can contain references to other documents, which can be followed in the same way, if necessary. What is important for us is that, instead of sending the actual document, the Web server can respond with a single URL pointing to another location of the document (for example, a replica of the server). The server can also rewrite the references inside the document in such a way that they point to the replica. These two techniques perform redirection, since the Web browser starts communicating with the replica, and does not contact the main server any longer (see Figure 2.3).

The main advantage of HTTP-based redirection is that it is extremely easy to deploy. Basically, all that we need is the possibility of serving dynamically generated Web pages. Apart from creating the actual content, the generator can also determine an optimal replica and rewrite all internal references in such a way that they now point to this replica. Moreover, each reference can be treated separately, enabling each document to be replicated at a different set of replicas. This feature can be crucial for devising an efficient distribution strategy for hosted documents. What is also important is that we do not need administrative privileges

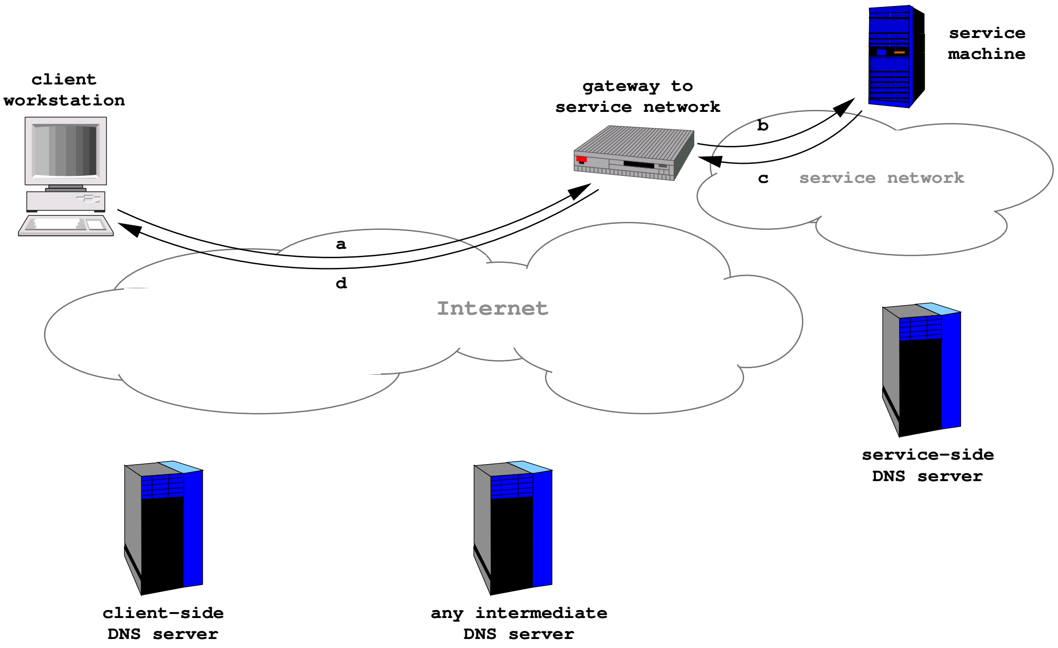


Figure 2.2: Establishing network connection

to run the redirector, neither do we have to run any additional network services. All these features make the HTTP-based redirection portable and attractive for the masses.

The HTTP-based redirection has also proved to be efficient. Although it is always required to retrieve an initial document from the main server, all further work proceeds only between the client and the selected replica, which is likely to offer optimal performance to the client.

As for scalability, the situation is somewhat worse. The necessity of making the first contact with always the same, single service machine can make it a bottleneck when the number of clients increases. However, as long as the initial document can be generated and delivered fast, we are still in a good shape. The easiest way of ensuring fast handling of this first transaction is simply to keep the initial document short.

The main drawback of HTTP-based redirection is that it lacks transparency. By receiving a URL which explicitly points to a certain replica, the browser becomes aware of being switched between different machines. This leads to the "bound reference problem" described in Chapter 1 when explaining the need for transparency. Since transparency constitutes our most important non-functional requirement, the lack of it makes the HTTP redirection unattractive.

Moreover, when using this kind of redirection mechanism, we cannot provide the client with more than one replica. Since each rewritten reference has to point to exactly one replica, the client is given no choice, and no alternative if the reference turns out to be unreachable.

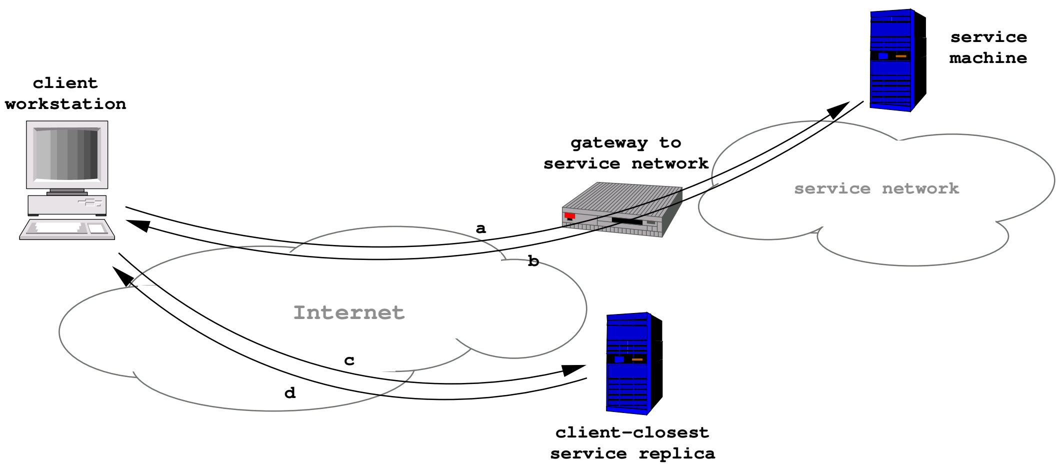


Figure 2.3: The principle of HTTP-based redirection

2.2. TCP Handoff

The Transfer Control Protocol (TCP) is a stream-oriented, transport-level protocol widely used in the Internet [21]. It is sometimes called TCP/IP, due to its tight relation to Internet Protocol (IP), a network-level protocol forming the core protocol in the Internet. A TCP stream is a bidirectional network connection established between two communicating parties. Each party is identified by means of an end point – a combination of the network address of the machine on which the party resides and the port number it uses. All data in the TCP stream are sent in portions, called segments. Each segment consists of a header, identifying the end points of both parties, and of a so-called payload, which is simply a piece of raw data provided by the higher-level software. TCP provides reliable communication by means of acknowledgments and retransmissions.

What is important for us is that the complete TCP segment is assembled on the machine that sends it. In particular, the machine can use an arbitrary origin end point when producing the header (instead of its own), and pretend that the segment originates from somewhere else. Not only can the end point machines play with the headers, but the routers in the path can do it as well.

Falsifying the end points in the segment headers is called "IP spoofing," and can be used to break into remote systems. In our case, however, it can be used with good intentions. Basically, we want to modify the behavior of the service network gateway. Instead of forwarding all the service-related traffic to the service machine, the gateway can switch the traffic between many service replicas. By looking at the target end point in the segment header, the gateway can recognize for which service the segment is intended. It can then select the best replica for the segment, and rewrite the destination address of the segment to forward it to the selected replica. The replica can send its responses directly to the client without bothering the gateway. It uses the "IP spoofing" technique to pretend that the response segments originate from the service machine (see Figure 2.4). What is important is that the selection

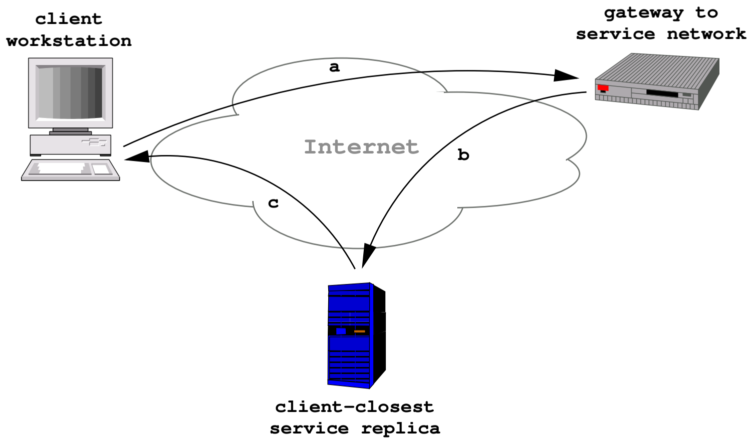


Figure 2.4: The principle of the redirection based on TCP handoff

made for the first segment of each stream has to be kept consistent through the lifetime of the stream, as no single stream can be shared by several replicas.

Another important detail is that the acknowledgments sent by the client still have to be dealt with by the gateway, as they are addressed to the service machine. In particular, they have to be forwarded to the respective replica as well. The complete mechanism is called a "TCP handoff," as it is based on passing the duty of servicing the TCP connection to the selected replica [23].

The TCP handoff is totally transparent from the client's point of view. Since it operates on transport-level streams, clients can never be aware of being redirected. They send their requests to the service machine and do not know that the intermediate gateway switches them between replicas.

Unfortunately, operating on such a low level also has a few negative consequences. The main one is that the complete mechanism is relatively complex. It requires using a specific network structure, modifying the operating system of each machine hosting a service replica, and implementing switching mechanisms on the gateway. All these requirements make it difficult to deploy and maintain this kind of redirection mechanism.

Another negative aspect of using the low-level traffic switching is that it does not scale efficiently over a wide-area network. This technique was originally designed for clusters and local-area networks where it proved to be highly efficient. But if we want to use it with replicas distributed worldwide, the matter starts to look much worse. We can no longer use local network facilities to pass a segment to a replica. Instead, we have to encapsulate the segment, and forward it through the external network, which introduces additional traffic and increases the response latency. This need for switching every incoming segment can dramatically degrade the overall performance of the system. Imagine a situation in which the service machine is placed in Europe and one of its replicas is in Australia. Usually, a client from Sydney is redirected to the Australian site. But every request is first sent to Europe, from where it is forwarded to Australia. Only the response is sent directly from the

Australian site to the client. It is clear that something is wrong here. We can conclude that the TCP handoff mechanism does not fit well in wide-area systems.

One more drawback of the TCP handoff treated as a redirection mechanism is that it distinguishes services based only on the combination of the target machine address and port number. In this way, if we want to replicate a service, we need to make a full copy on each replica, thus losing the flexibility of partial replication.

Yet another disadvantage of the TCP handoff, as was the case in the HTTP-based redirection, is that the client cannot be offered more than one replica to choose from. The redirection mechanism remains in full charge of what happens to the client requests.

2.3. DNS-based Redirection

The Domain Name System (DNS) is a distributed naming system widely used in the Internet [14]. Basically, it translates machine names into their network IP addresses and vice versa. It is based on its own protocol, supported by millions of DNS servers all over the world. DNS servers continuously exchange information on correspondence between domain names and network addresses in order to provide their users with one service: name resolution.

A classical DNS usage pattern is as follows. The client wishes to communicate with a certain machine, but it knows only the name of the machine. It needs to translate this name into a network address in order to communicate. Therefore, it asks its local DNS server to resolve the former into the latter. The DNS server, in turn, looks for the answer in its local cache and, if an appropriate record is found, returns the cached address to the client. Otherwise, it contacts other DNS servers, turning itself into a client that looks for the machine address.

There are two methods that the DNS server can use to resolve the query. By using the first of them, called iterative, the DNS server asks its peers for either the result or the address of another DNS server likely to know it. In the former case, the answer is cached locally and then passed to the client. Otherwise, the DNS server issues one more query to the DNS server pointed to by the peer, and the whole procedure repeats itself.

The second method is called recursive. When using this one, the DNS server asks its peer for nothing but the machine address. Therefore, it can be said that all queries issued by clients which are not DNS servers themselves are recursive. The peer can react in three ways. Firstly, it can forward the query (also as a recursive one) to another DNS server. The chain of recursive queries which is built in this way ends at the DNS server that knows the answer. Alternatively, the peer can decide to find the answer using iterative queries, and then return it. Finally, the peer can simply refuse handling the recursive query, as supporting recursive queries is resource-costly.

What can be noticed here is that unless the answer is cached at one of the intermediate DNS server the query eventually reaches the DNS server "authoritative" for the service domain. The authoritative server, in turn, can respond with any address it prefers. In particular, it can respond with the address of the service replica that it finds best for the

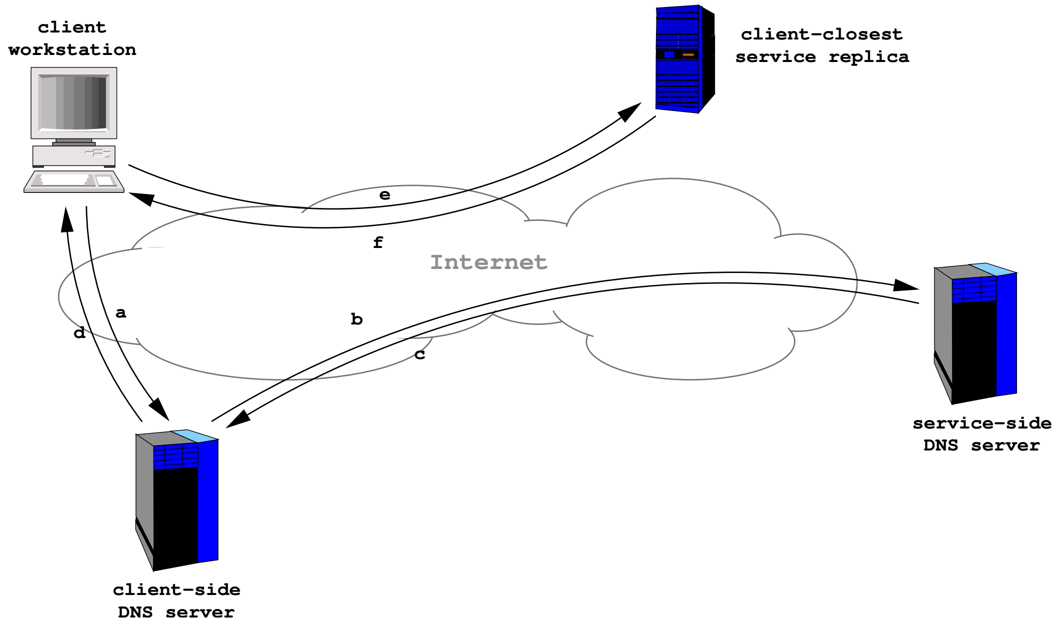


Figure 2.5: The principle of DNS-based redirection

query. The answer will finally be returned to the client which will contact the replica instead of the service machine (see Figure 2.5). As long as the client uses DNS names to reference the service, and not network addresses, this scheme will give the effect of client redirection.

DNS-based redirection has several advantages. The most visible one is that it achieves transparency without losing scalability. It is transparent because the clients are obliged to use the addresses provided by the authoritative DNS server, and cannot establish whether these addresses belong to the home machine of the service or to any of its replicas. DNS as a distributed name resolution service proved to be very efficient, even though the amount of people using it has increased tremendously with the growth of the Internet.

Another vital advantage of using DNS to redirect clients is that it is a natural way of informing the clients about the service addresses. It is used by many existing network services, and is very likely to be used by those to come as well. Moreover, DNS is supported by a huge infrastructure of millions of DNS servers, capable of caching the answers our redirector generates. Once we make this infrastructure work for us, both efficiency and availability of our redirector considerably increase.

One more important advantage of DNS is that it allows multiple replica addresses to be returned, enabling the client to choose one from them. This feature was not supported by any of the previously discussed redirection mechanisms.

The last advantage of DNS-based redirection is its good maintainability. Deployment of the complete redirection mechanism boils down to launching a single modified DNS server, and subsequently delegating a service domain to this server. From this moment on this server is responsible for answering requests for the service address. No other modification of the DNS infrastructure is necessary.

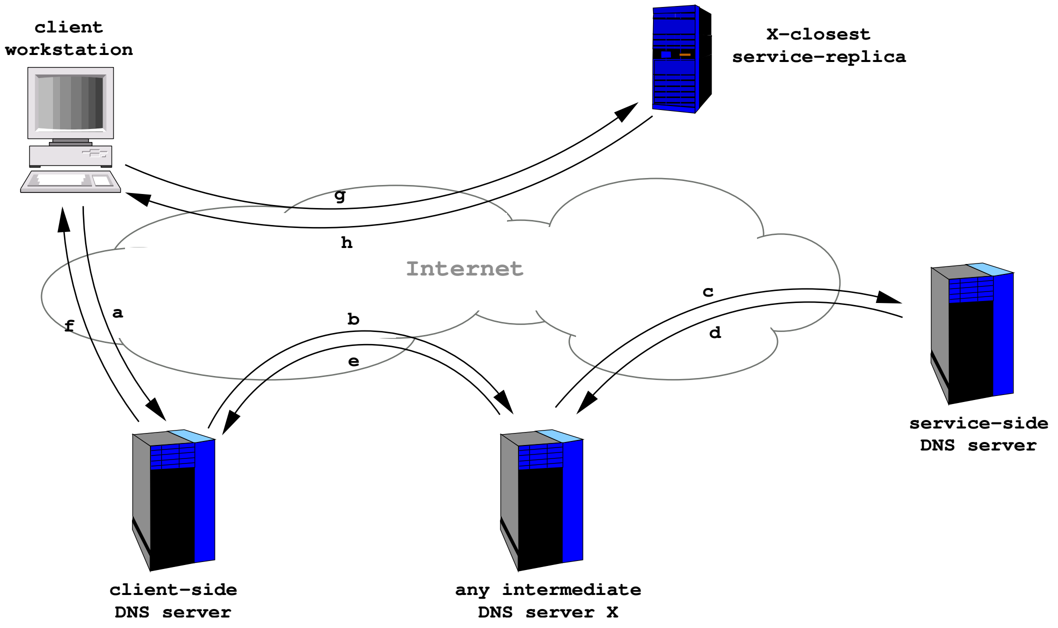


Figure 2.6: DNS-based redirection in the presence of recursive queries

On the other hand, using DNS-based redirection leads to a few difficulties. The first of them is caused by the fact that DNS queries carry no information about the client that triggered the name resolution. All that the service-side DNS server knows is the network address of the DNS server that asks about the service location. Therefore, we have to assume that clients always use a DNS server that is close to them, and approximate a client's location to that of its DNS server [20]. Whether we consider it to be a drawback or not depends on the accuracy we want to achieve. Studies show that 64% of clients are located in the same network as their DNS servers [12]. Thus, as long as we do not need strict per-client redirection, the location of the client DNS server approximates the client well enough.

Another problem, tightly related to the first one, is what happens when a recursive query occurs. To process this kind of query, DNS servers create a chain of queries that ends at the service domain DNS server. What poses the actual difficulty is that the latter knows only the address of the DNS server that is one step before in the chain, and not the origin of the chain. The service domain DNS server therefore has no information about the location of the client (see Figure 2.6). Even if one of the intermediate DNS servers decides to switch to iterative mode, the information about the client location remains uncertain.

What helps here is the policy suggested in the DNS protocol specification [14]. According to this policy, the chain of recursive queries should never cross the border of an administrative domain, be it a small school or a large Internet Service Provider. DNS servers located in different administrative domains are strongly recommended to exchange information using iterative queries only. This applies both to the DNS servers that send queries outside their administrative domains and to the ones that receive them. The latter should refuse handling recursive queries coming from a domain managed by another party (and it is very likely that they would do so anyway, as processing recursive queries requires more computational resources). Note that the administrative domain can cover any number of network domains – the only constraint is that they are managed by a single institution.

Assuming that the policy is always respected and that every institution cares about having fast network connections between all machines falling under its administration, even in the case of recursive queries the redirector can still approximate the location of the client with the address of the DNS server issuing requests.

Yet another complication can be caused by caching answers in DNS. To avoid contacting the service domain DNS server too often, and to resolve repetitive queries faster, each answer can be assigned a TTL value (Time To Live). The TTL specifies the maximum time for which the answer can be stored in the network and considered valid. Thus, as long as a DNS server has such an answer in its cache, it can respond to a client immediately without bothering the service domain DNS server that previously produced the answer. Since regular domain names seldom change, DNS servers usually use large TTL values (the suggested value is 3 days, and the maximum – one week [5, 15]). The problem is that all responses of the redirector are time-dependent and can expire shortly. If we cannot control their lifetime, we can end up in the situation in which the clients are redirected to replicas which are not optimal, or even no longer exist. A fundamental issue for our approach is how to choose TTL values. Large values favor caching, therefore efficiency. Small values, however, favor accuracy.

Fortunately, the problem of TTL selection is simpler than initially expected. According to recent research, caching efficiency is mostly stable for TTL values greater than 10 minutes [11]. Since we can expect replica lifetimes to be at least of the order of hours, choosing TTL values of 10 minutes appears to provide a reasonable trade-off.

Another (relatively little) problem is that URLs can contain a TCP port number on which the server is expecting requests. Since the client software will open TCP connections to the port number found in the URL, all replicas of the site have to be accessible through the same port. In particular, if the main server services several ports, all its replicas should service them all as well. This limitation may introduce some problems with keeping the configuration of all the replicas consistent. In comparison to making the content consistent, however, it is a rather small complication.

The last potential problem is that DNS cannot distinguish between different services located on the same machine. What can be helpful in case of DNS, however, is that different services hosted by the same machine can be referred to by different DNS names (normally resolved to the same IP address). In this way, if we need to replicate services separately, we can distribute the data from every service independently. Obviously, it is still less flexible than in the case of the HTTP-based redirection, where we can use a separate placement strategy for each document.

No.	Category name	HTTP-based redirection	TCP handoff	DNS-based redirection
1.	Redirection transparency	*	*****	****
2.	Level of integration	*****	*****	*****
3.	Scalability	****	**	*****
4.	Deployment	*****	*	****
5.	Redirection granularity	*****	***	***
6.	Client loc. identification	*****	*****	***
7.	Multiple response	*	*	*****

Table 2.1: A comparison of redirection mechanisms

2.4. Discussion

Let us now compare the redirection mechanisms described above. Table 2.1 presents a summary. A higher number of stars indicates a higher qualification. The maximum of five stars is given to the redirector that is the best in the given category. Other redirectors get stars relative to the winner. In particular, one star means either that the given redirector does not support a certain feature at all, or that it does so, but in a completely unsatisfactory manner.

The first category is the level of transparency that each of the methods offers. TCP handoff is the best here – virtually, clients have no means of discovering the redirection. The DNS-based redirection is a bit worse, as it relies on domain names and not network addresses. The HTTP-based redirection is definitely the worst, since it provides no transparency at all.

The second criterion is the level of integration into existing protocols that each of the method achieves. Since all methods are based on widely-known solutions, they all score well.

The third category evaluates how scalable each method is. The DNS-based mechanism is rated best in comparison to the others because of its caching features. The HTTP-based redirector is slightly worse, due to the necessity of retrieving the initial document from the main server. Scalability in case of the TCP handoff is limited to local networks. Hence, it has only two stars.

The fourth category considers maintainability, in particular the easiness of deployment. As it was already mentioned, the HTTP-based redirector is the best here, whereas the DNS-based one may take some effort to deploy. TCP handoff requires modified operating systems and network gateways, so it is given only one star.

The fifth criterion indicates the level of granularity at which each of the mechanisms distinguishes services. Fine granularity (high rank) means that we can redirect to each document separately. Coarse granularity (low rank), on the other hand, means redirection to an entire site. Since the HTTP-based redirection mechanism enables us to treat every single document separately, it is given the highest mark. As neither virtual servers nor port numbers provide such a level of flexibility, both the DNS-based redirector and the TCP handoff are noticeably worse.

The sixth criterion says how accurately we can establish the client location. Both HTTP redirection and TCP handoff perform best here – in both cases the redirector knows the exact address of the client. DNS redirection is worse, since it has to approximate the client location with the address of the client-side DNS server.

The seventh category comprises the possibility of returning multiple replica addresses in the redirection process. Since only the DNS-based redirector offers this functionality, it gets the highest mark, while the other two get the lowest one.

A crucial observation is that it is preferable to have a redirection mechanism that performs reasonably well in all criteria, rather than one that is exceedingly good in some, but bad in the others. DNS-based redirection obtains three to five stars in each evaluation category, clearly making it a better trade-off than the others. Therefore, we decided to implement a DNS-based redirection mechanism.

Chapter 3

Implementation Details

In this chapter we discuss the implementation of NetAirt, a DNS-based redirector in the form of a module for the Apache Web server. We first describe the architecture of the Apache HTTP server, version 2.0.32-beta [1]. Then, we present the structure of our redirector. Finally, we show how the redirector fits into Apache, and how we have solved the problems we encountered.

3.1. The Apache HTTP Server

3.1.1. Request Processing Model

For our purposes, the most interesting thing that we have to know about Apache is how it deals with client requests. With this knowledge we can adjust the server to our needs.

As was said in Chapter 2, HTTP is a TCP-based, application-level protocol. TCP, in turn, is a connection-oriented, transport-level protocol. This means that the communicating parties have to establish a network connection before any application-level data can be sent. The communicating parties thus first setup a TCP connection, send and receive HTTP data, and subsequently close the connection.

The request-processing model in Apache also reflects this scheme. What is important, however, is that Apache further splits the three main phases (connect, transmit, disconnect) into smaller ones. The need for having more specialized phases comes from using HTTP, in which we can distinguish many stages of request processing. Such HTTP-related stages include, for instance, client authentication, request verification and classification, and so on. In Apache terminology, each stage is called a “hook.” In general, a hook represents a certain class of operations that have to be performed on the request.

The number and type of hooks that process a given request depend on the request itself. Although there exists a default hook sequence, followed by typical HTTP requests, it does not have to be used each time a request is received. Since it is Apache that passes requests between different hooks, it can also modify the default sequence, depending on the status

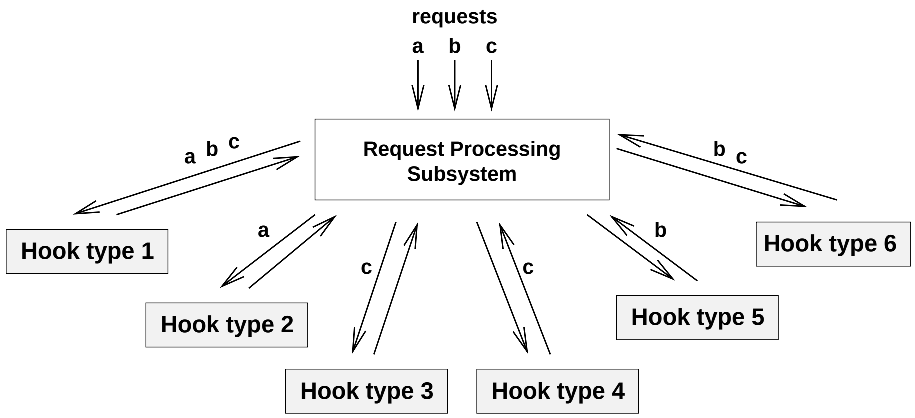


Figure 3.1: Each request can be processed by a different sequence of hooks

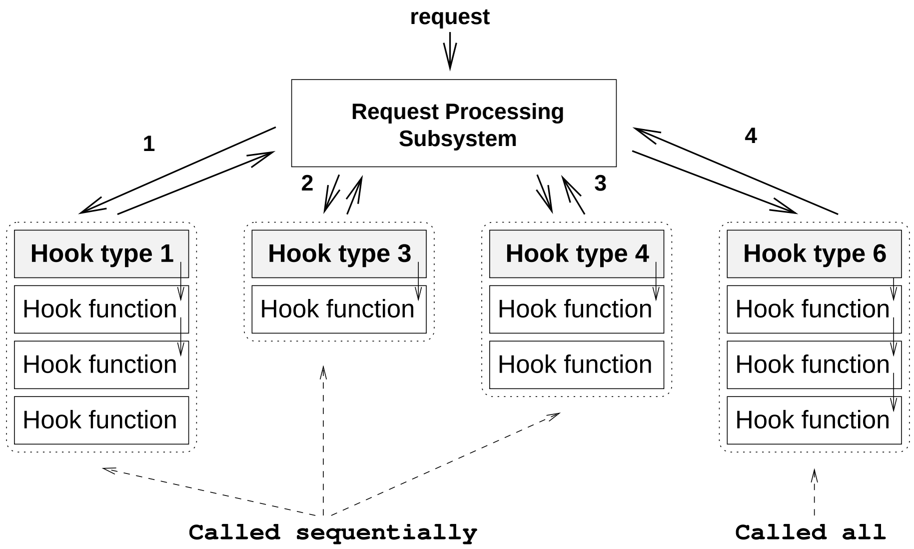


Figure 3.2: Each hook decides how its hook functions are called

codes returned by the hooks executed so far. In this way, the server can omit some of the hooks, or use some special, normally untouched ones (see Figure 3.1).

Each hook can be associated with one or more so-called "hook functions." Apache calls these functions each time a request is passed to the hook. For each hook it is also defined, *how* they should be called. Two ways are possible: either all the hook functions are called, or they are called sequentially until one of them reports to have handled the request (see Figure 3.2). In case none of the functions can deal with the request, an error occurs. This mechanism allows us to modify the behavior of the whole server: all we have to do is define custom hook functions and associate them with the appropriate hooks. The latter operation is called "registration."

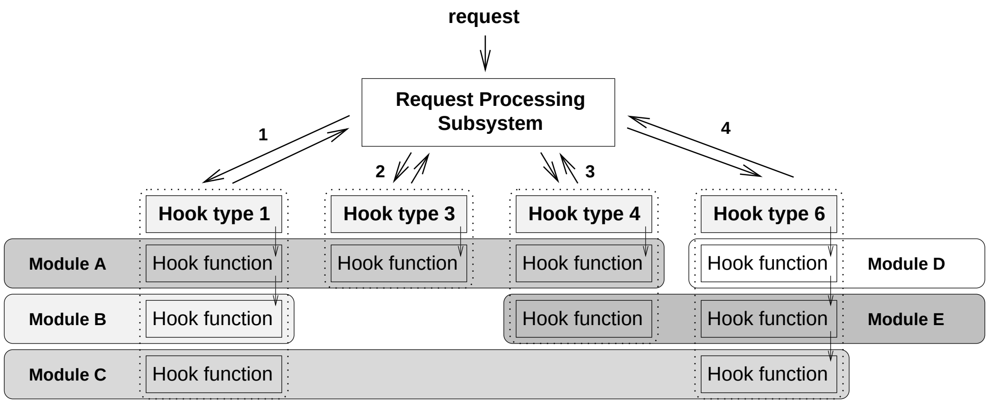


Figure 3.3: Hook functions usually come from different modules

3.1.2. Modules

Hook functions (possibly intended for different hooks) are grouped into modules. A module in Apache is a set of hook functions, together with some administrative data structures that allow registration of the hook functions with their respective hooks. Usually, a module provides hook functions for a small subset of available hooks, as it is not interested in changing the request processing entirely (see Figure 3.3). For example, one can implement a module supporting a new access control policy, or a module translating the server output depending on the location of the client.

3.1.3. Configuration

Apart from hooks used during request processing, there are also hooks used for server configuration. Usually, each configuration hook is associated with a sole hook function and a specific configuration directive. Each such directive can be inserted into an Apache configuration file. Every time Apache parses the file and encounters the directive, the corresponding hook function is called.

The original set of Apache configuration directives can be easily extended. If we need to make our module configurable, we simply have to insert the extra directives (and the corresponding hook functions) into a special table. This table can be found among the module's administrative data structures.

There are also configuration hooks that are not related to any configuration directive. They are not limited to have only one associated hook function, either. These include a hook executed just before parsing the configuration file, and a hook executed once the parsing is finished. Intuitively, the former can be used to create data structures modified in the other configuration hooks, while the latter allows to initialize the data structures according to the current configuration.

3.1.4. Apache Portable Runtime

Apache provides programmers with a so-called Apache Portable Runtime (APR). It aims at creating an abstraction of operating system, which makes the modules independent from the underlying architecture. There are many OS-specific versions of APR. All of them, however, implement the same interface, which can be used by module developers.

To mention at least a part of what APR offers, there are routines for memory allocation, network communication, locking, and logging. APR uses its own abstraction of memory (both private and shared), network connections, and files. It comes with a huge set of functions useful for data management, like operations on strings and generic data structures. All these facilities make module development easier, and the final code portable across Apache servers.

A special data structure that is available in APR is a resource pool. It is a collection of miscellaneous resources, including, for example, memory chunks and file descriptors. Pools help manage the allocated resources and allow to release them once they are no longer needed. A typical example of the application of pools is to associate a separate pool with each client connection so that resources needed only for the connection processing are released right after the connection is closed. We can perceive this mechanism as a specialized garbage collector.

3.1.5. Multi-Processing Modules

Apache solves the problem of portable multiprocessing in an interesting way. It defines a few hooks that are not related to request processing, nor to the configuration. Instead, they define the behavior of the server as a set of threads. They decide, for example, what type of concurrency is used (threads or processes), when new threads are spawned, and how incoming client requests are dispatched to them. Modules providing hook functions for this special type of hooks are called Multi-Processing Modules (MPMs).

Different MPMs are usually used on different system architectures. There are, for example, separate MPMs for Windows, OS/2 and NetWare. An exception to this rule is the group of MPMs that run on Unix systems. At the moment, there are three different Unix MPMs: *prefork*, *worker*, and *perchild*. The first one is process-based, while the two remaining ones use mixed processes and threads.

Various MPMs can implement different request-to-thread mapping strategies. They can also use different criteria to decide when to create a new thread. In this way, not only can they use the operating system in the most efficient way, but they can also adjust every instance of Apache to the conditions occurring at the specific site (e.g., load, range of documents, their size, etc.).

3.1.6. Internal Data Structures

Apache modules have access to selected internal data structures of the server. A good example here is the list of "listeners." A listener is a special data structure containing a network socket and a pointer to an "accept" function that is going to be used with this socket. By inserting and removing listeners to and from the list we can make Apache wait for incoming connections on multiple ports.

The list of listeners is used every time a client request is received. Recall that HTTP requests are transmitted using TCP connections. Typically, an incoming TCP connection is first "accepted" (which corresponds to the phase of establishing the connection) and then "processed" (which corresponds to the actual data exchange). Also, Apache makes sure that the complete HTTP response is sent ("flushed") to the client before the TCP connection is closed. It is done internally by the request-processing subsystem, however, and for this reason there is no customizable phase for that (and no special hook function, either).

To avoid conflicts between multiple threads trying to simultaneously accept the same connection, a special lock is used. In this way, all the "accept" operations are serialized. Moreover, each accept-then-process sequence is associated with its private resource pool. It is created before accepting, and destroyed after closing the connection.

We can thus perceive Apache as a server running the following loop:

1. **create** an empty connection resource pool;
2. **lock** the global "accept_lock;"
3. **poll** on the set of sockets extracted from the list of listeners;
4. take any socket reporting the arrival of new data;
5. **accept** a new TCP connection on the socket, using the accepting function associated with the socket (if present), or a general one (otherwise);
6. **unlock** the global "accept_lock;"
7. **process** the accepted connection by passing the connection to the request-processing subsystem; it then passes the connection to appropriate request-processing hooks;
8. **destroy** the connection resource pool;
9. **return** to step 1.

The reason for presenting such a detailed description of how Apache deals with TCP connections is that it provides a good basis to understand the way in which we add UDP support to Apache. We will return to this problem later in this chapter, when discussing the details of implementing the redirector inside Apache.

3.2. Redirector Design

Our redirector is much less complex than the Apache HTTP server. It consists of the following five parts (see Figure 3.4):

1. Replica database;

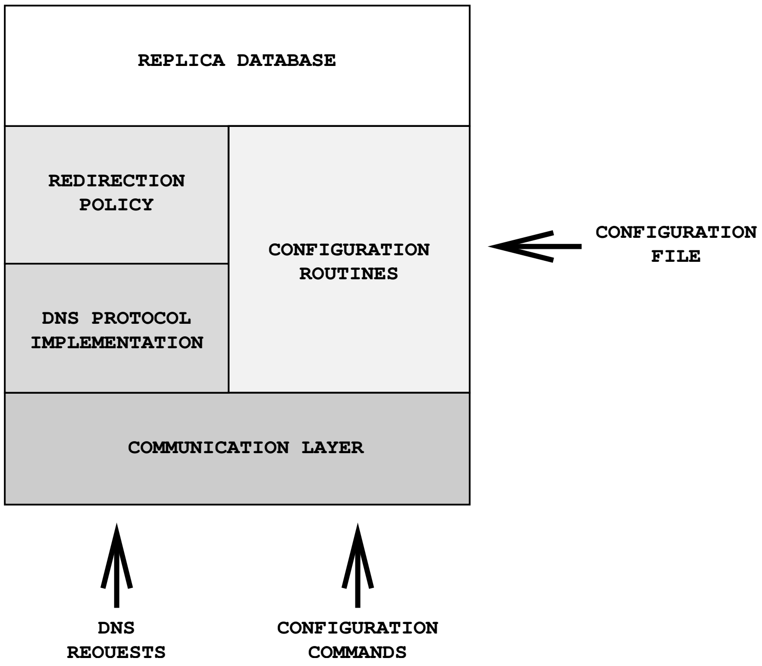


Figure 3.4: The design of the redirector

2. Redirection policy implementation;
3. DNS protocol implementation;
4. Configuration routines;
5. Communication routines.

Since the main purpose of the redirector is to provide its clients with replica addresses, a place to store them is needed: a replica database. It contains the list of service names, represented in our case by domain names. Each domain name is associated with a collection of replicas, described by their IP addresses and TTL values. Each replica can support many services. For each domain name it is possible to specify a different redirection policy. Therefore, a policy identifier is a part of the service description stored in the database.

The database is used by the heart of our redirector – the redirection policy component, which generates a list of (IP address, TTL value) pairs for the given domain. The way the list is generated is discussed in Chapter 4, where we present redirection policies.

The redirection policy component is called from and passes the results back to the DNS protocol implementation, which provides the basic functionality of a DNS server. It accepts a DNS packet and decodes it. All malformed or non-query packets are dropped. In case the packet is not a simple name-to-address query, an empty DNS response (containing an appropriate error code) is generated. Otherwise, the domain name found inside the query and the sender address are passed to the redirection policy implementation. The latter responds with a list of (IP address, TTL value) pairs, which is subsequently encoded into a DNS response packet and passed back to the caller.

The configuration routines enable the system administrator to change the default settings of redirector. These include, for instance, the number of IP addresses returned in a single DNS response and a default TTL value. They also allow to set the size limits of the replica database and its initial content. Finally, they let the administrator adjust policy-dependent configuration. The way in which all these routines are invoked depends on the software in which the redirector is going to be embedded.

The main duty of the communication layer is to receive DNS packets. Each DNS packet can be transmitted using either UDP or TCP. Therefore, the communication layer must support both of them. After a packet is received, it is passed on (in a format independent from the transport protocol) to the DNS protocol implementation. If the latter reacts with a response DNS packet, the communication layer sends it back to the sender of the original packet. Obviously, both packets use the same transport-level protocol.

Another duty of the communication layer is to support a networked maintenance interface. By means of a simple text-based protocol it enables the system administrator to modify the content of the replica database. In this way we enable the cooperation between the Apache-based DNS redirector and any external replica control program.

3.3. The Redirector as an Apache Module

Let us now take a look at how the redirector can be put inside a module for the Apache HTTP server. We will treat each of its five components separately, with special attention paid to the problem of using both UDP and TCP transport protocols to receive DNS packets.

3.3.1. Replica Database

Since Apache is a multithreaded server, any of its threads can access and modify the replica database. The problem is that the database content has to be consistent among all threads: changes made by one should be immediately visible for the others. Therefore, our database implementation should not only be accessible for multiple threads, but it should also properly synchronize all modifications of the content so to avoid race conditions. In our solution all the database structures are placed on shared memory chunks, provided by APR. In this way we ensure that it remains the same for all the threads, no matter which multithreading model is used – be it thread-based, or process-based. Access synchronization is done by means of mutual exclusion locks, also available in APR.

3.3.2. Redirection Policy Implementation

Apart from standard initializing routines, the redirection policy component provides a special policy function, which is called for each DNS request. The function takes four parameters:

1. The DNS name the request refers to;
2. The client IP address;
3. The current configuration of the redirector;
4. An empty table where it should store the resulting replica IP addresses and TTL values.

In general, the duty of the policy function is to fill in the table with a set of IP addresses of the replicas and their associated TTL values. It first retrieves all the information about the domain name from the database, including the list of replicas. Then, it passes this list to the actual implementation of the desired policy, indicated in the domain name description. In this way all the policies have an uniform interface, and do not have to know anything about the database. The fact of switching between different policies remains invisible to the other components of the redirector.

Many instances of the policy function can be ran simultaneously (in separate threads) without interfering with each other. They are serialized only during operations on the replica database by the internal mechanisms described above.

Since implementing the policy function is a complex problem, we return to its detailed description in Chapter 4.

3.3.3. DNS Protocol Implementation

The implementation of the DNS protocol reflects the algorithm used in BIND, a popular DNS server [8]. However, since our redirector supports only the most primitive DNS queries, it lacks all the advanced functions present in BIND. As a result, we have a compact DNS server with just the functions we need to achieve redirection. Moreover, our DNS implementation is completely independent from the Apache environment. It could be equally well embedded inside any other piece of software.

3.3.4. Configuration Routines

Configuration routines include all redirector-related configuration hook functions and the Apache-specific data structures describing the module. The most important directives have been discussed above in Section 3.2 and for this reason are not repeated here. What is worth mentioning, however, is that one of the general configuration hooks, called "post_config," is used. It is the best place to create all internal data structures of the redirector, since at this point the complete redirector configuration is known, and Apache has not yet started processing the client requests. Therefore, in the "post_config" hook function we initialize the replica database and create a special lock protecting the UDP socket (see below).

3.3.5. DNS/TCP Support

Adding the support for TCP-based DNS datagram exchange is simple. We can use one of the request-processing hooks, called "process_connection." Since it completely overtakes all other request-processing hooks, it allows to implement virtually any TCP-based protocol. The only complication is that the hook function needs to operate on the native Apache abstraction of a TCP connection, which involves some additional APR calls to access the raw content of the stream. A similar situation occurs when translating a raw DNS response to the Apache format. Except for these details, however, there are no substantial problems.

3.3.6. DNS/UDP Support

The situation is much worse in the case of UDP. In essence, Apache does not allow a module to wait for incoming UDP datagrams. Although APR provides some basic routines for UDP-based communication, the entire request-processing subsystem implicitly assumes that all data arrive through TCP connections. As a consequence, it accesses the data via "read" and "write" calls. Unfortunately, these functions do not work with UDP, where another two should be used ("recvfrom" and "sendto," respectively). To solve this problem, we have to extend the original server.

First, we need to force Apache to react to incoming UDP datagrams. We can exploit the list of listeners by adding our handmade UDP socket to the list. We thus instruct Apache to consider it while polling for network events. The socket is created in one of the directive-related configuration hooks, but its actual initialization takes place in the "post_config" hook, discussed above. The reason of creating the socket in two separate phases is that Apache examines the list of listeners after parsing all the configuration directives, but before invoking the "post_config" hook. Since during the examination Apache initializes some other internal data structures, adding any sockets to the list after the examination does not work. Interestingly, sockets do not have to be configured to pass the examination – it is enough that appropriate listener structures can be found on the list. By initializing our UDP socket in the "post_config" hook we remain consistent with the behavior of Apache, which also creates listener structures in the directive-related configuration hooks, but initializes them after the configuration phase has finished.

Unfortunately, adding the socket alone is not enough. The problem is that Apache expects sockets to have set up a TCP connection. Therefore, if Apache passed our UDP socket to the request-processing subsystem, everything would break down, as the routines used to retrieve and send data via TCP connections differ from those used for communicating via UDP.

Recall from Section 3.1.6 that, after isolating a socket to service, the server first calls an accept function which can be customized in a per-socket manner, and then passes the accepted socket to the request-processing subsystem. It is tempting to implement the whole DNS server in a customized accepting function, and then report some error to avoid calling the request-processing subsystem. There are a few problems, however. Firstly, recall that concurrent calls to the accept function are serialized. If we decided to process incoming UDP datagrams as part of an accept function, we would lose all the benefits of running a concurrent server. Secondly, in the original server there is no way of gently breaking the accept-then-process sequence – either it succeeds completely, or a critical error is signaled. Without getting into further details, we would have to change the semantics of the accept function. The problem is that it is called by the MPM, and for this reason we would have to change all the MPMs that can ever be used.

What we propose is modifying the Apache core to associate the socket with an additional "processing function," used to process the accepted socket *instead* of the request-processing subsystem. A pointer to this function can be added to the listener structure.

Then, we modify the very beginning of the request-processing subsystem as follows. It first checks whether a processing function has been provided for the just-accepted socket, and if so, calls it instead of proceeding with its usual operation. The benefit of having it done here is that there is only one request-processing subsystem: we need only one modification, unlike the case of MPMs.

The problem that arises here is where exactly the request-processing subsystem should look for the customized processing function. Since it only knows about the socket descriptor (and not the entire listener structure), it has to be informed about the processing function in a different way. Our solution exploits the connection resource pool. APR allows to treat each pool as a dictionary-like data structure. Using a special APR call, we can associate a pointer to the listener structure in the connection resource pool with a certain key. This is done in the customized UDP accept function. Later on, when the pool is passed to the request-processing subsystem, the latter can retrieve the pointer from the pool using the same key. In this way it can examine the content of the listener structure. In particular, it can check whether any custom processing function has been defined for the socket that is about to be processed.

Another problem we face is caused by the various ways MPMs deal with accepted sockets. Some of them can try to perform some socket-specific operations on the socket returned by the accept function. These operations include "setsockopt," or even "close," if the server is overloaded and cannot service the socket. Therefore, our accept function must return a valid TCP socket, even though in this particular accept-then-process sequence we are only going to use UDP. For this reason, we create a brand new TCP socket in the accept function, and return it to the calling MPM. The socket is destroyed at the beginning of the processing function, as its only purpose is to fool the MPM.

Now that we have two separate accept and process functions, called sequentially each time the UDP socket reports the arrival of new data, we can use them as follows. The accept function reads *all* datagrams that can be found in the socket buffer. In this way we prevent the socket buffer from being overflown, as it could happen if the accept function retrieved a single datagram each time. Then we register the list of datagrams in the connection pool

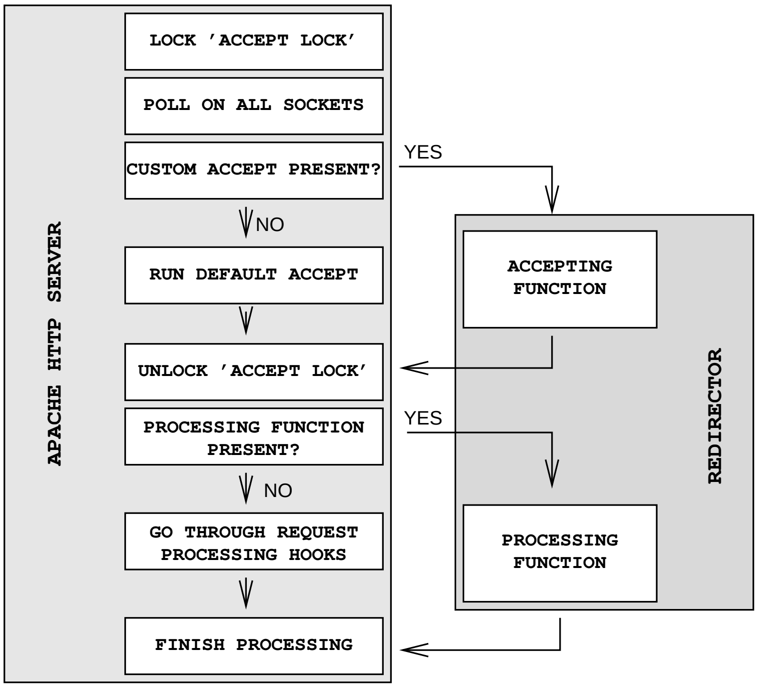


Figure 3.5: Overtaking UDP datagrams in Apache

under a special key, just as in the case of the listener structure.

The processing function reads the list of datagrams registered inside the connection pool. The datagrams are serviced one by one, and all the responses are sent using the same UDP socket. To avoid possible collisions between multiple "send" operations coming from different threads, they are serialized using a special lock. Since many calls to the processing function can be done in parallel, we still have a concurrent server.

Below we can see a typical UDP datagram processing, presented similarly to the previously discussed "Apache loop," and illustrated in Figure 3.5:

1. **create** an empty connection resource pool;

2. **lock** the global "accept_lock;"

3. **poll** on the set of sockets extracted from the list of listeners;

4. take any socket reporting the arrival of new data;

5. perform **accept** on the socket, by calling the special accept function associated with the socket;

   (a) **retrieve** any datagrams that reside in the socket buffer;

   (b) **register** the list of datagrams inside the connection pool;

   (c) **register** the socket-related listener structure inside the connection pool;

   (d) **create** a fake TCP socket and return it;

6. **unlock** the global "accept_lock;"
7. **process** the socket, by passing it to the request-processing subsystem;
   (a) **extract** the listener structure from the connection pool;
   (b) **unregister** the listener structure from the connection pool;
   (c) **call** the processing function found inside the listener structure;
      i. **destroy** the fake TCP socket;
      ii. **extract** the list of datagrams from the connection pool;
      iii. **unregister** the list of datagrams from the connection pool;
      iv. **service** all the datagrams;
8. **destroy** the connection resource pool.

The reason for unregistering the data (both the listener structure and the list of datagrams) from the connection pool is that some MPMs reuse pools, instead of destroying them and creating new ones. If we left the data registered inside the pool, Apache could try to access it again the next time the pool would be used. Obviously, it would be incorrect behavior, as the data associated with the pool should not be accessible after the processing function has terminated.

Chapter 4

Redirection Policies

In this chapter we present three different methods by which our redirector can select replicas for a given client. They include a static name-to-address policy, a "round-robin" address selection policy, and a proximity policy based on AS-path length metric. This chapter is organized as follows. We start with presenting a general policy model and the theoretical foundations of the two first policies. Then, we discuss the most advanced, the AS-path-based one. Finally, we give implementation details for all policies.

4.1. Introduction

4.1.1. Policy Model

As was said in Chapter 1, the redirection policy defines the method of replica selection. It takes as input the address of a client and the list of replica addresses (which are assumed to belong to the same service). Having this information the policy chooses a specific number of addresses of replicas that are likely to service the client most efficiently. Although this task may seem simple (if not primitive), the optimal set of addresses can be very hard to identify. As we will see later in this chapter, the selection algorithm can be very sophisticated.

4.1.2. Static Name-to-Address Policy

The simplest method for a policy to select replica addresses is to always return the first n addresses from the list. This method does not differ from what a typical DNS server does. Basically, it returns all the site addresses it knows. The fact that it may ignore an exceeding number of addresses (as it only returns a certain number of them, even if the list of addresses is very long) does not change anything – it still gives the same response each time it is queried (see Figure 4.1).

The reason for including this straightforward address selection method is very simple. The main purpose of the redirector is to support domains containing replicated services. However, these domains may also contain machines that host non-replicated services for which the static name-to-address mapping is good enough. Making the redirector capable of servicing such

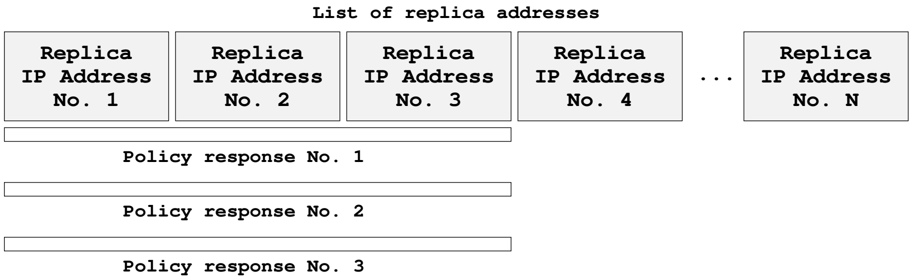


Figure 4.1: The static policy always returns the same addresses

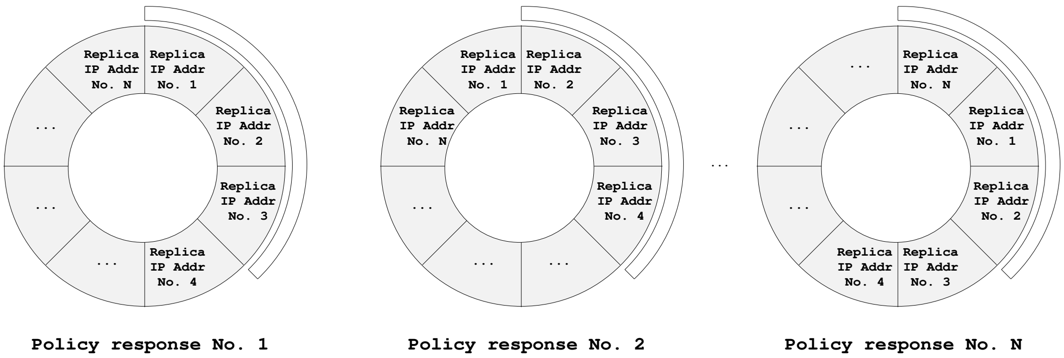


Figure 4.2: The round-robin policy uses a circular list of addresses

machines as well increases its flexibility. We will also see later in this chapter that this policy also serves as a fallback if more advanced policies fail to return a list of replicas.

4.1.3. Round-robin Policy

Round-robin is a slightly more advanced policy. Given a list of replica addresses, it also sequentially takes as many as needed, but diversifies its responses by starting at subsequent addresses each time. After reaching the end of the list, round-robin moves to its beginning, and so on, thus implementing a circular list. Additionally, after each query, it rotates the list by one address, so that the address now returned as second will be the first next time, and so forth (see Figure 4.2).

The simple idea of using a circular list of addresses makes round-robin much more powerful than the static name-to-address mapping. In theory, if 1000 clients came to use a service that has 10 replicas, each of the replicas would service 100 of them (assuming that each client contacts the first replica in the list of addresses it obtains). Therefore, it seems that round-robin provides simple load balancing. Practice shows, however, that it is not that easy. The problem lies in DNS response caching. Imagine what happens to the response that the redirector generates. As long as we use non-zero TTL values, a response will be cached on at least one DNS server somewhere in the world. Until the response expires, the DNS server will give it to anyone that needs it. As a consequence, all the clients of the DNS server will be (re)directed to a single replica. In this way the load balancing we implement is dependent on the number of clients that ask each DNS server for the replica addresses. This problem is known as a *hidden load factor problem* and has no satisfactory solution so far [7, 12].

For our purpose, restricting the redirector to use round-robin only is unacceptable. Since round-robin does not consider the client location when generating a response, it cannot be used to redirect clients to "their" replicas. It is still useful, however, to support this policy. Although the round-robin load balancing is imperfect, it may turn out to be good enough for some applications.

4.2. AS-path Length Policy

One of our objectives concerning policies is that the responses they generate depend on the client's location. Intuitively, each client should be redirected to a nearby replica. The idea standing behind this is that the shorter the distance between communicating parties is, the less time is needed to transmit the data. The client-perceived efficiency is thus likely to be higher.

The problem now is how to measure the distance between machines in the Internet. Geographical distance is not a good enough metric because it completely ignores the topology of the Internet, which plays a key role in determining the actual connection speed.

A more promising solution is to use routing data to discover the path followed by the data sent to the client by the service. In this section we introduce a so-called "AS-path length" Internet distance metric. We then use it to develop a proximity-based policy that selects the replicas closest to the client. To understand better how the metric works, let us first explain some basics of routing in the Internet.

4.2.1. Routing Basics

Each computer in the Internet is identified by its IP address. For our purposes, let us assume that addresses are unique, that is each IP address identifies one machine. An IP address is a 32-bit positive integer. For convenience it is usually decomposed into four 8-bit parts. If we then sequentially write all the parts as decimal numbers and put dots between them, we get the address written in the popular *Internet dot notation*, such as `192.168.16.2`.

A network is a range of IP addresses. It is often referred to by a *prefix*. A prefix is simply an IP address prefix of a certain size [3]. It is denoted similarly to IP addresses, with its size (in bits) added after the "/" mark, for example `192.168.8.0/21`.

Data in the Internet are sent in portions called packets. Each packet contains, among others, its sender's and receiver's IP addresses. After leaving the sender's machine, packets are shipped between routers until they reach their destination.

Routers connect the Internet together. Each router has a direct connection to two or more peer routers or local area networks. When a router receives a packet, it examines its destination IP address to determine which peer router the packet should be sent to. It does

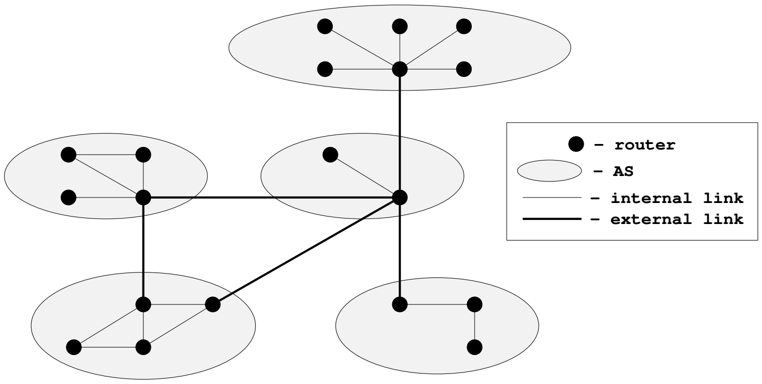


Figure 4.3: A network of routers split into 5 Autonomous Systems

so using a locally maintained routing table, which contains a list of prefixes. Each prefix is associated with a list of peer routers that accept the traffic heading to the prefix. One of these peer routers is then selected to receive the packet.

4.2.2. Autonomous Systems

The graph of routers representing the Internet can be split into subgraphs, called Autonomous Systems (ASes) [10]. Each such subgraph takes care of its internal routing using any internal routing protocol it prefers (OSPF, for example [16]). More importantly, however, the subgraphs exchange routing information with each other using an external routing protocol. In this way they ensure global reachability of the networks they enclose (see Figure 4.3).

Each Autonomous System is identified by a globally recognized 16-bit Autonomous System Number (ASN). It is also assigned a certain set of prefixes, and it is responsible for handling the traffic heading to them.

The relatively small number of ASes (currently around 13000) makes it possible to construct a graph with each node representing an AS, which can be treated as a map of the Internet. More importantly, because of the size of the graph, efficient global algorithms applied to it can give results within a reasonable time. Efficiency refers to the theoretical computational complexity of the algorithm.

We define a distance metric between two machines as the minimum number of (inter-AS) links between the ASes these machines belong to. This metric is used by the most common external routing protocol, BGP, which we discuss below. For example, in Figure 4.4 the distance between machines A (in AS64001) and B (in AS64004) is 2, as the shortest route between these ASes only goes through AS64003. This metric is commonly called the "AS-path length metric" [13]. It is the base of the AS-path length policy: given candidate replicas (to service a client) the policy chooses those which have a minimum distance from the client. Since the AS-path length metric is successfully exploited in external routing, we believe that it is suitable to determine the distance in the Internet.

We are aware that basing the distance estimations only on the number of AS boundaries

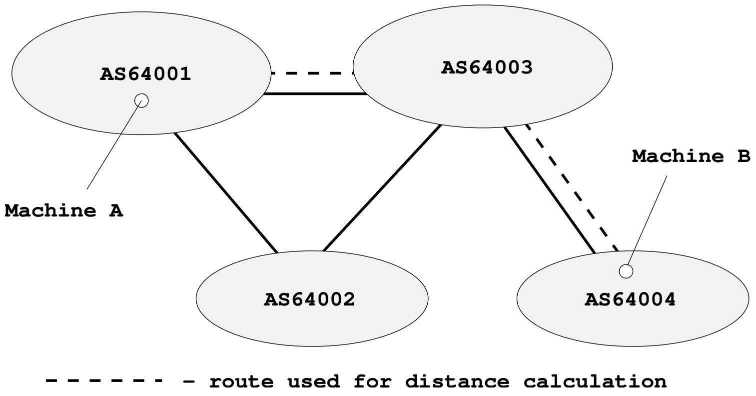


Figure 4.4: The AS-based distance means the minimal number of AS-links between two nodes

the route crosses may not be sufficient. The problem is that Autonomous Systems significantly vary in size, which makes it hard to ensure that the AS-internal communication remains fast. There are ASes covering hundreds of prefixes, and ASes having only a few of them. Some ASes are distributed over several continents, and some consist of routers all placed in only a single city. Finally, some institutions manage many different ASes, whereas the other control only one AS, which can influence the experience of the AS-managing staff. All these differences may introduce new factors, which should be considered in latency estimations as well [22]. Still, exploiting the AS-based map of the Internet is a way of dealing with the problem of distance measurement. It is also likely to be more accurate than the geographical distance.

4.2.3. Border Gateway Protocol

The Border Gateway Protocol version 4 (BGP) is an external routing protocol [19]. It was designed to support the inter-AS exchange of routing information.

In BGP, each router informs its peers about the prefixes it knows. We say that it "advertises" routes to these prefixes. For each prefix, it provides a so-called AS-path, which is a sequence of ASNs indicating through which Autonomous Systems the route to the prefix goes. For example, in Figure 4.5 one of the routes that AS64003 advertises to AS64001 contains the AS-path `64003 64004 64005`, which means that this route goes through ASes 64003 and 64004, and then ends in AS 64005.

The method of selecting the preferred route assumes that the shortest path indicates the minimal network distance, and thus the minimal cost of delivering a packet. In case a router has received several AS-paths for the same prefix from different peer routers, it chooses the one that offers the shortest AS-path. The chosen route is marked as *best*. In Figure 4.5 all routes in the BGP routing table are marked as *best*, except for one, for which a shorter alternative can be found. That is why AS64001 will decide to use AS64002 as the peer AS through which the traffic to the prefix `192.168.4.0/22` should be sent.

A BGP routing table contains a list of routes to prefixes, each of which is associated with

AS64001 BGP ROUTING TABLE

BEST	PREFIX	AS-PATH
YES	192.168.8.0/21	64002
YES	192.168.16.0/22	64003
YES	192.168.0.0/22	64003 64004
YES	192.168.4.0/22	64002 64005
NO	192.168.4.0/22	64003 64004 64005

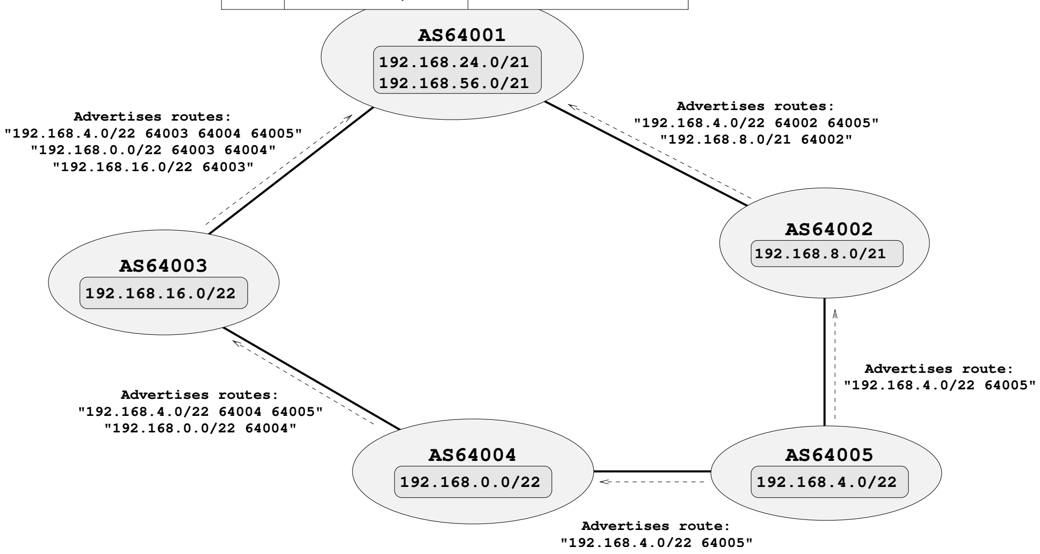


Figure 4.5: A simple BGP-based network

the IP address of the peer router that has offered the route, and the AS-path for this route. Each route has also a defined "origin" attribute, informing where the route information comes from: whether it was learned using an internal routing protocol, an external one, or another way. The last situation typically occurs when at some point the route switches from BGP to some other routing protocol. The peer IP addresses and the route origins are omitted in the routing table in Figure 4.5, as they are not essential to understand how BGP works. We will need them later, however, when discussing the BGP details.

BGP routers are allowed to compress the data in their routing tables to reduce the amount of memory needed, and the size of routing messages they exchange. A popular technique of BGP data compression is called *route aggregation*. In this technique, a set of route specifications is replaced with a single, aggregated route specification. Route specifications can only be aggregated if they are associated with contiguous prefixes, which in turn form a single, bigger prefix. In other words, the set of target IP addresses covered by the prefixes from the original route specifications, and by the big prefix from the aggregated route specification must not change. For example, the four following prefixes:

- `192.168.0.0/24`
- `192.168.1.0/24`
- `192.168.2.0/24`

- `192.168.3.0/24`

can be replaced by the one below:

- `192.168.0.0/22`

Compressing prefixes implies compressing the associated AS-paths. The AS-path in the aggregated route specification consists of two parts. The first one, called *AS-sequence*, is the sequence of ASNs common to all the original AS-paths. Importantly, the ASNs enumerated in the AS-sequence are listed *in the same order* as in all the original AS-paths. The second part of the compressed AS-path is called *AS-set*. It is an *unordered* set of ASNs that occur in at least one original AS-path, but cannot be included in the AS-sequence, either because they do not occur in some of the original AS-paths, or because they occur in all of them, but in various places. Obviously, if all the original AS-paths are identical, the AS-set in the compressed AS-path is empty, and the entire compressed AS-path is simply a copy of one of the original AS-paths. In textual AS-path representations, AS-sets are delimited by the { and } brackets. For example, one may compress the two following AS-paths:

- `64018 64011 64006 64043`
- `64018 64011 64007`

into

- `64018 64011 {64006 64043 64007}`.

Route aggregation usually leads to information loss. Since ASNs included in AS-sets are unordered, it is impossible to derive the original AS-paths from a compressed one, as long as its AS-set is non-empty. Importantly, it is still possible to determine which peer router should be given a packet heading to a prefix associated with an aggregated route. Since the AS-sequence in the compressed AS-path contains the initial part of the real route, its first ASN denotes the next Autonomous System, to which the packet should be passed. Following this scheme the packet will eventually reach the AS where the aggregation was performed, and where unaggregated route specifications are still known. In our example, the BGP routers in AS 64011 will process the packet using the two original route specifications, and forward it accordingly, either to AS 64006 or to AS 64007.

4.2.4. Constructing a Map of the Internet

Each BGP routing table represents a partial view of the Internet's topology. It reflects the routes that are likely to be taken by traffic starting from the router that holds the table. Having such a view, the router knows enough to deliver packets, but has little clue about the structure of the complete network. This subset of all available paths is extremely likely to cover only a small part of all existing inter-AS connections [6].

On the other hand, our aim is to build a graph of Autonomous Systems that is as complete as possible. Therefore, we need to gather routing tables from several routers all over the world, and extract a view of the Internet from each of them. Once we merge all these views together, we obtain a map of the Internet that is more complete than any of the original views. Also, the more places we take views from, the less chance that we miss existing inter-AS connections.

An important observation here is that it would be sufficient to collect the views coming from ASes that contain our replicas. The only inter-AS links we really have to know are the ones that can be used on the path to and from our service. Therefore, we do not need to bother with the links that cannot be found in any of these views. Unfortunately, the weakness of this approach lies in its main requirement: the ability to extract views from multiple routing tables, downloaded from BGP routers in several Autonomous Systems. BGP routers can seldom be accessed by unprivileged users, and arranging an access to hundreds of them is difficult. Moreover, failing to retrieve routing tables from the AS of only one replica would affect the entire system: if this replica cannot be located on the generated graph, clients cannot access the service using this replica.

Although we cannot gather a reasonable set of BGP tables ourselves, we can still use data that are publicly available. One of the most prominent projects that publishes BGP routing tables is *RouteViews* [2]. On its Web site we can find a huge set of routing tables, gathered from a special RouteViews BGP router. It is configured to communicate with many BGP routers all over the world and build its own, huge routing table. Moreover, the RouteViews BGP router does not perform any data compression, thus avoiding possible information loss. In this way, its table is a composition of all the routing tables of its peers, and contains many alternative routes to each prefix. This gives an effect similar to having many separate routing tables, each coming from different parts of the Internet. Although these parts are probably not exactly the ones that contain our replicas, they are still representative enough to provide the global picture that we need.

Using the RouteViews BGP routing table has two fundamental advantages. Firstly, it relieves us of the necessity of downloading several routing tables ourselves, and offers everything we need in a single file. The compressed version of this file is 17 MB large, which makes it feasible to download it from the RouteViews site. Secondly, a new snapshot of the complete BGP routing table is added to the collection every 2 hours, and made instantly publicly available. As long as the redirector periodically downloads and processes the most recent version of the RouteViews data, it can claim to always have an up-to-date map of the Internet. Moreover, people who need to retrieve the data more frequently can use special, differential BGP routing table snapshots. They are made every 15 minutes, and published on the RouteViews site as well. Since the routes in the Internet are relatively stable, updating the graph every 15 minutes should be frequent enough for most applications [17].

4.2.5. Policy-based Routing

Even when we finally manage to build a map of the Internet, there are still some difficulties we need to take into account. The main one is called *policy-based routing* [24], by which a

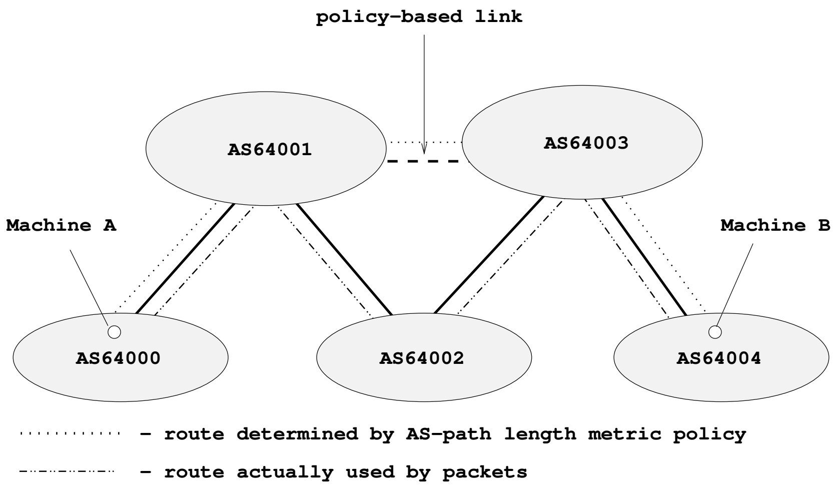


Figure 4.6: The policy-based routing can falsify the distance calculations

router is configured to use different criteria than just a distance metric to decide which peer to send a packet to. For example, some inter-AS connections are made available only to packets coming from a pre-selected set of ASes. This may lead to a situation in which the distance measured on the AS-based map is inconsistent with the actual route length.

For example, assume that we have built our map of the network presented in Figure 4.6 using the BGP routing tables coming from both AS 64001 and AS 64003. In this way the map we have built is complete. Also, assume that the policy-based link between these two systems can be used only by packets originating from one of them. According to our metric, the distance between machines A and B is 3, as the shortest route between them goes through ASes 64000, 64001, 64003, and 64004. It is inconsistent, however, with the real route of length 4 that packets from AS 64000 or AS 64004 will follow, as these packets are not allowed to make use of the link between AS 64001 and AS 64003. In practice, it is impossible to distinguish between "public" and "policy-based" inter-AS links given only a routing table.

4.2.6. IP-to-ASN Translation

Now that we have a map of the Internet, we need a method to find an ASN number for a machine, given its IP address. To do this, we can exploit the same set of BGP routing tables.

Recall that a BGP routing table contains a list of entries containing the prefix the entry concerns and the associated AS-path. Intuitively, the prefix belongs to the last AS in the path. We call this AS the destination AS of the prefix.

This simple way of mapping prefixes to Autonomous Systems works fine as long as we deal only with complete AS-paths. Unfortunately, our experience with the RouteViews BGP data shows that not all AS-paths have these features. For example, there are aggregated route specifications that contain non-empty AS-sets in their AS-paths. Since AS-sets are unordered, we cannot determine the destination ASes for the associated prefixes. Worse yet, as these prefixes are compressed, they comprise smaller prefixes, each having possibly a different destination AS, which we cannot determine either. Despite being defined in the BGP standard, aggregated routes are not many: they constitute below 1% of the RouteViews BGP data.

Our solution to this problem is to consider the compressed prefix as a whole, and treat the last unaggregated AS in the AS-path (the last one in the AS-sequence) as the destination AS of the prefix. Since this AS is the farthest AS that is guaranteed to be passed by the route, no matter how the route is split, it is also the best possible approximation of the real destination AS.

Another problem is posed by incomplete routes, that is routes for which the "origin" mark indicates an unknown information source. They constitute about 12% of the RouteViews data. As was said in Section 4.2.3, an incomplete route is a consequence of switching to another routing protocol at some point in the route. Since we have no other way of establishing the end of the route, we decided to treat the last available AS in a AS-path as the destination AS of the corresponding prefix.

Finally, although the RouteViews BGP data set is extremely large, some valid IP addresses may not be referenced there. We quantified them based on the list of clients that have visited the VU Web server. The number of clients for which we were unable to determine any destination AS turned out to be negligible: less than 0.5%. In other words, we can establish home ASes for 99.5% of clients of a typical Web site, even though these ASes are not always the true locations of the clients (due to the two irregularities described above).

4.2.7. Final Algorithm of the AS-path Length Policy

It is now time to describe more precisely the selection algorithm our policy uses. Given the IP addresses of the client and all replicas, it first determines which ASes they belong to. Then, it runs a Breadth Search First (BSF) graph search algorithm on the AS graph to identify the n replicas that are closest to the client. That is why in Figure 4.7, the client in AS 64006 will be given the addresses of replicas B, C, and D (in this order). Replica A is more distant than any of these three.

Possible optimizations to the above algorithm include precomputing (or caching) the ASNs of replicas, precomputing complete responses, and reordering peer ASes lists in our map of the Internet so that it becomes faster to search through. All these modifications are left for future work, as well as the investigation of the correspondence between the AS-specific features (e.g., number of peer ASes and prefixes connected) and their position in the optimal ordering in the peer ASes lists.

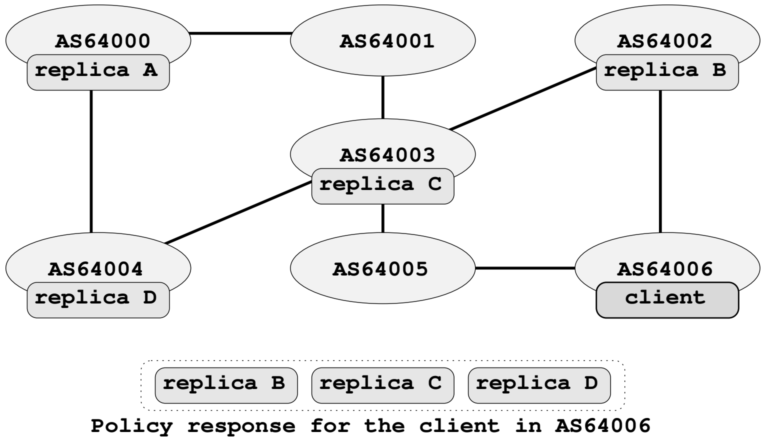


Figure 4.7: The AS-path length policy finds n client-closest replicas

4.3. Policy Implementation

4.3.1. General Concepts

Recall from Section 3.3.2 that policy implementations are called from the policy-switching component. Firstly, it forwards all the policy initialization and destruction calls, so that each of the policy implementations could initialize and release any resources it needs. Secondly, for each client query, it retrieves the list of candidate replicas from the database and passes them to the implementation of the policy that has been configured for the site in question. The implementation-specific functions accept the following parameters:

1. The IP address of the client;
2. The identifier of the site that replicas belong to;
3. The IP addresses of replicas;
4. The current configuration of the redirector;
5. An empty table for the policy to return the result.

The site name that is passed to the policy-switching component is translated to a site identifier. A policy implementation can use it to identify any policy-internal, site-related data. Also, since the list of replicas is passed as parameter, policy implementations do not need to contact the replica database. In this way they work independently from the rest of the redirector.

4.3.2. Simple Policies

The static name-to-address mapping policy is implemented directly in the policy-switching component. Apart from being called for sites that are configured to use it, it is also invoked when the site in question has only one replica, or if the original policy cannot choose any replica. It can thus be perceived as a fallback mechanism.

The round-robin policy first allocates a private table of counters, where each counter (one per site) points to the replica from which the next response should start. Each counter in the table is indicated by the identifier of the corresponding site. The table is placed in a shared memory chunk, and protected by a special lock. In this way it can be shared by all response-generating processes.

4.3.3. The AS-path Length Policy

The AS-path length policy exploits two data structures. The first of them is a binary tree of prefixes. It is used to determine the ASNs of the client and the replicas. In its most compressed version, the tree is about 2 MB large. The tree is built similarly to those used by routers to determine the peer router to send a packet to [25]. Each node contains a prefix, which is common to all prefixes in its descendants. Also, each node contains an ASN for addresses matching the prefix, or 0 if the prefix is too short to establish the ASN. If, for a given node, the ASNs of its two child nodes are equal, then the children are pruned and the node acquires their ASN. If a given node has only one child node, and the ASN of the parent node is 0, then they are merged. In this way, the tree height is made as low as possible. IP-to-ASN translations are extremely fast: below 1 microsecond on average, when running on a PIII/1.1GHz machine.

The second data structure is the AS graph used to run the BSF algorithm on. It is implemented in the form of association lists. The implementation consists of two tables. The first table contains the list of direct inter-AS links. The second table is indexed by ASNs. For each ASN it indicates the offset in the first table at which the AS-specific interval starts. All ASes that have no links point at offset 0.

Both data structures are created based on a BGP data file, downloaded, for example, from the RouteViews web site. Since the file reflects the situation in the Internet, it should be periodically updated by an external program, so that the redirector always operates on a recent version. The AS-path length policy re-reads the file periodically (once a day, by default), and builds new versions of the lookup tree and the AS graph.

Interpreting the BGP data file contents can take several minutes, as the file is over 400MB large (decompressed). Thus, the file cannot be parsed during the server configuration phase, as doing so would increase the server startup time. A similar problem occurs when reloading the file, as the AS-related data structures that are being updated cannot be used to answer client queries. In our solution we use two separate BGP data sets, each consisting of one lookup tree and one map of the Internet, and switch between them as follows. During initialization, the AS-path length policy allocates two shared memory chunks for the two separate BGP data sets, and creates a special switching structure. As long as none of the BGP data sets is available, the redirector uses the static name-to-address mapping instead of

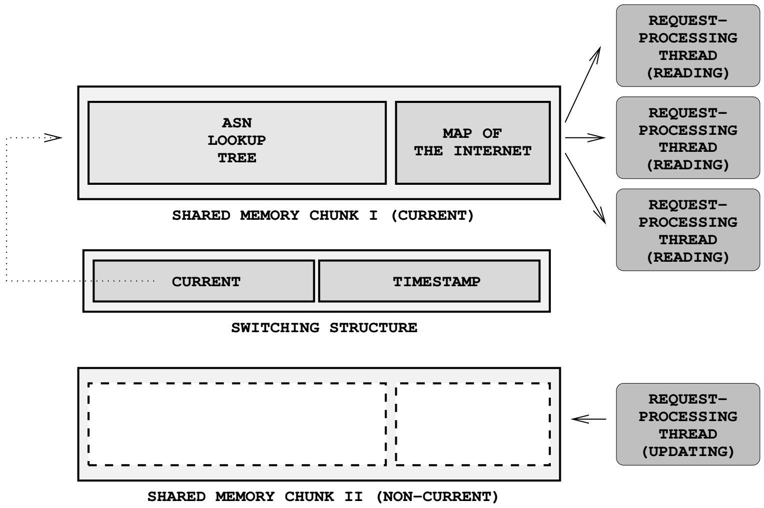


Figure 4.8: The AS-path length policy switches between two BGP data sets

calling the AS-path length policy. The actual loading of the BGP data file takes place after processing the first DNS request. Basically, the thread that was handling this request does not immediately return to accept a new one, but reads the file and prepares a new BGP data set, which is then placed in one of the shared memory chunks. As soon as it has finished, the data are made available to other threads by indicating the chunk as *current* in the switching structure (see Figure 4.8).

An analogous scheme is followed when reloading the BGP data file. One of the request-processing threads notices the necessity of updating the BGP data by checking the timestamp that can be found in the switching structure. Then, it re-reads the BGP data file and builds a new BGP data set, which is then placed in the *non-current* shared memory chunk. In this way, while the file is being parsed, the remaining request-processing threads can still use the *current* chunk. Once the new BGP data set is ready, the timestamp and the *current* chunk indicator are updated accordingly. For this reason, some version of the BGP data structures is constantly available, and the AS-path length policy can be used all the time.

The AS-related data structures are used in the main policy function that performs the replica selection. It first checks (in the switching structure) which BGP data set it marked as *current*, and then uses it to run the actual algorithm, described above. The function does not modify the shared BGP data. Therefore, no locks are needed here, and many instances of the function calls can be run simultaneously.

Chapter 5

Performance Evaluation

In this chapter we present the results of the experiments we conveyed. All of them were performed on a standard PC machine, equipped with one PIII/1.1GHz processor. We first present a performance evaluation of the DNS transport layer when using the two simple policies. Then, we show micro-benchmarks of the AS-path length policy.

5.1. Performance of the DNS Transport Layer

In our first test we wanted to investigate how long it takes to service a single client. Therefore, we measured an average Round Trip Time (RTT), which is the time between sending a DNS request and receiving a response. To make the results as accurate as possible, we did not use any physical network connection. Instead, we placed both the client and the Apache server running our redirector on the same machine. Also, to avoid any concurrency-related inaccuracies, we configured Apache to use only one request-processing thread.

RTTs were measured in the following way. A client sequentially issued 2000 DNS requests that regarded the same, fictitious domain. We measured the time of calling the "res_send" function, implemented inside the system resolver library, and calculated an average of all the results. This test was run using TCP-based and UDP-based queries.

Additionally, to measure the overhead introduced by different policies, we performed the above two tests for both the static name-to-address policy, and the round-robin policy.

Finally, to discover the correspondence between the number of IP addresses returned in a single DNS response and the actual RTT, we ran all the tests on four separate redirector configurations, each responding with a different number of addresses. The results of our tests are summarized in Figure 5.1.

As can be observed, there is a significant difference in RTTs between queries using TCP and UDP: retrieving a DNS response with TCP takes almost 80% more time than doing that with UDP. This difference is probably caused by the cost of establishing a TCP connection.

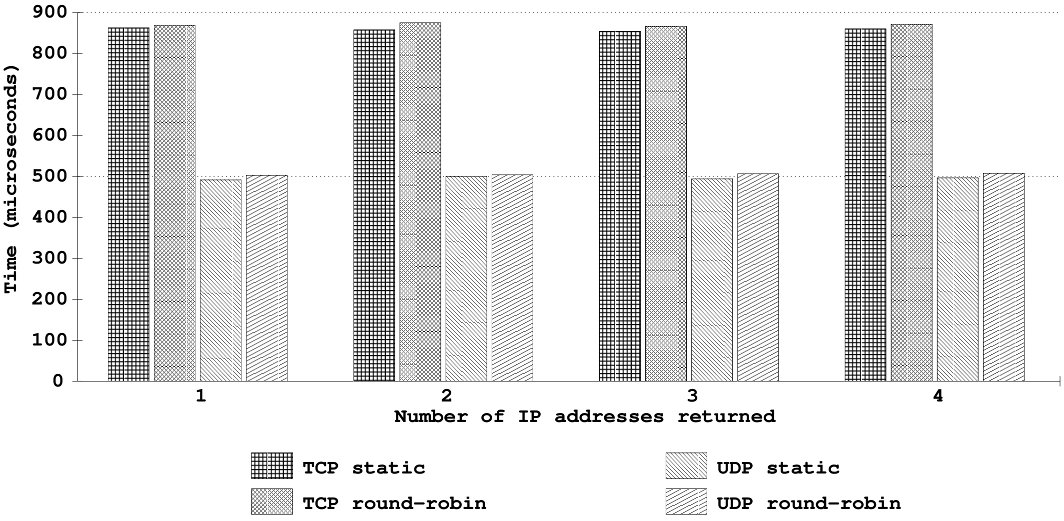


Figure 5.1: Round Trip Times for simple DNS queries

The overhead introduced by the round-robin policy turned out to be negligible in comparison to total execution time. Similarly, the increase in number of IP addresses returned in a single DNS response results in insignificant growth of the overall RTT.

5.2. Micro-Benchmarks of the AS-path Length Policy

After investigating the simple policies, we started experimenting with the AS-path length policy. We measured whether (and how) the AS graph search time varies, since this search constitutes the main source of overhead generated by the AS-path length policy. Intuitively, the more distant a client is from its closest replica, the longer it will take for the BSF algorithm to explore the AS graph before finding the replica.

The simulation we performed was as following. First, for each existing AS, we configured the redirector to service one non-replicated site located in this AS. Then, for each of those sites, we measured the time that the BSF needs to find it in the AS graph, assuming that the search always starts from the same AS, representing the location of a fictitious client. To measure only the delays due to the search (and not delays due to Apache), we removed the replica-selecting function of the AS-path length policy from Apache, and embedded it into a single-threaded C program. The measured delays included the time spent on the IP-to-ASN translation, which turned out to be negligible (mean translation time was below 1 microsecond). Since the test was run outside Apache and focused on the AS-path length policy function, the overhead generated by the DNS transport layer was not measured.

The above simulation was executed for 16 different fictitious clients, located at various academic sites throughout the world. In general, each client was "located" on the Web server of a certain university. As for the US clients, we placed them in Berkeley, in MIT, and at Washington University. Non-US clients were placed at universities in their respective capital cities. We present a selection of our results in Figure 5.2.

As expected, all the figures look similar. When the replica is very close to the client, the search algorithm needs to explore only a few nodes of the AS graph before finding the replica. When the distance grows, the number of nodes to explore dramatically increases, leading to bigger and more scattered delays. Finally, only a few nodes are located at a very long distance from the client. Searches for these nodes are long but all the same: they correspond to a full exploration of the AS graph.

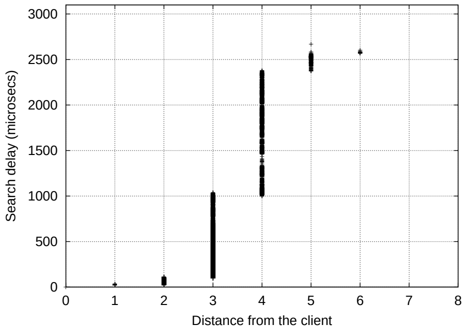
We support this interpretation by looking at distributions of distances between the client and each node in the graph. Figure 5.3 shows that most nodes are indeed located at distances 3 to 5 from the clients.

Using these results, we can derive the delay for locating n closest replicas for a replicated site. Basically, the time that is needed to generate a list of n replicas for the client is equal to the search time of the n-th closest replica. For example, if the n-th closest replica for a client at Vrije Universiteit Amsterdam is located at distance 3, then it will be found in between 0.05 and 1.2 milliseconds (see Figure 5.2a). Similarly, finding the replicas distant from a Polish client at Warsaw University by at most 6 inter-AS links will take between 0.8 and 2.5 milliseconds (see Figure 5.2b).

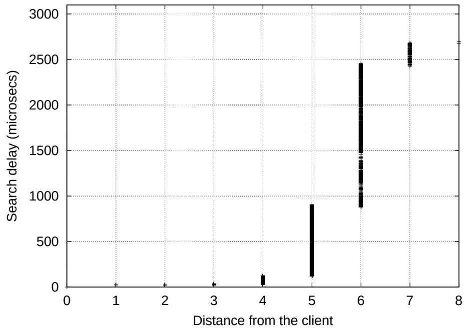
As we can see, the closer a client is to the replicas, the less time the AS-path length policy needs to generate a DNS response for him. More importantly, however, the search never lasts longer than 3.1 milliseconds, and the mean search time equals 0.64 milliseconds. These values are very low compared to the latency caused by any non-local network transmission.

Having the above search times, we may also conclude on the theoretical server throughput. In a pessimistic case, when all queries require exploring the entire map, a server running the redirector can handle about 320 queries per second. In a typical case, however, the throughput will be over 1500 queries per second, since the mean search time equals 0.64 millisecond. Note that the above values only refer to the AS-path length policy implementation, and do not consider the cost introduced by DNS message processing. This theoretical throughput is likely to be ample enough for most sites. Moreover, CDNs that need to service more queries can always combine several machines running identically configured redirectors to increase the overall throughput to any desired level.

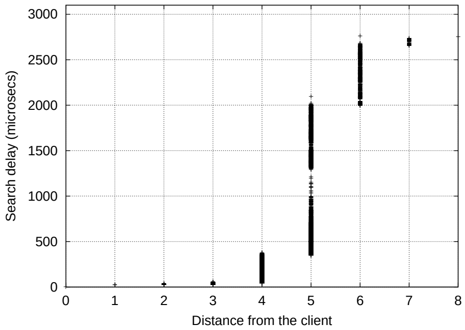
a. The AS-graph search times for a client inside the *.nl domain



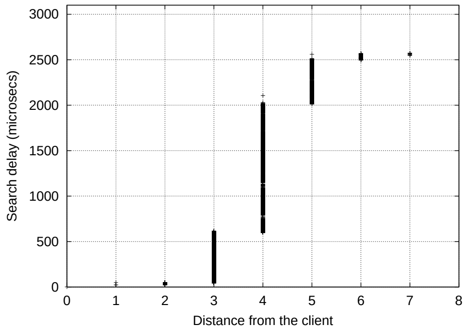
b. The AS-graph search times for a client inside the *.pl domain



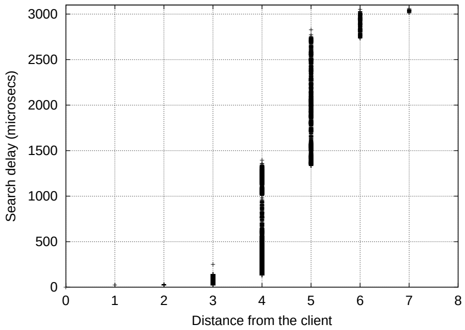
c. The AS-graph search times for a client inside the *.ru domain



d. The AS-graph search times for a client inside the *.mit.edu domain



e. The AS-graph search times for a client inside the *.za domain



f. The AS-graph search times for a client inside the *.br domain

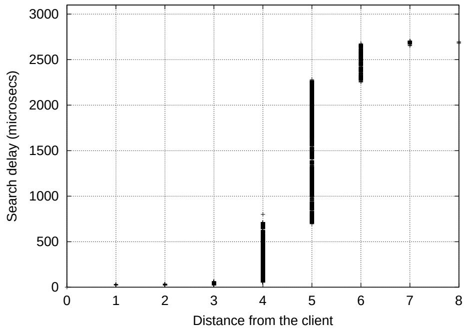


Figure 5.2: The AS graph search times for 6 representative clients

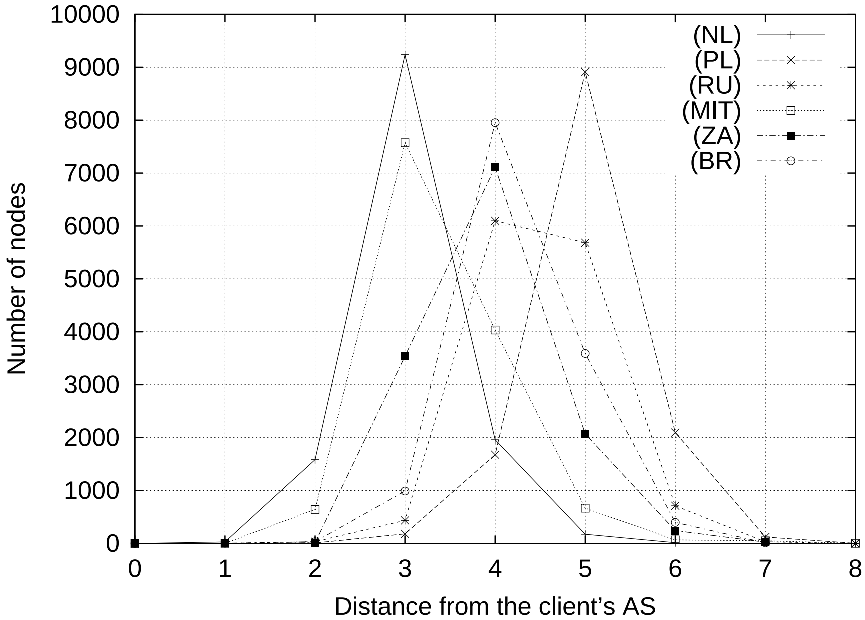


Figure 5.3: The inter-AS distance distribution for 6 representative clients

Chapter 6

Conclusion

This thesis presents the client redirection problem and the requirements that should be met by a good solution. These requirements include: transparency, scalability, maintainability, and efficiency. Among three popular redirection mechanisms (HTTP redirection, TCP handoff, and DNS redirection), we conclude that a DNS-based redirection mechanism meets the four above requirements best.

Our implementation, called NetAirt, proves that with little effort Apache can be made to service DNS packets, no matter which transport-layer protocol is used: UDP or TCP. Moreover, our module can easily be integrated with an external replication-managing component, designed either as a separate Apache module (like Globule, another project of ours), or as a completely independent program.

We have paid attention to the development of a reasonable set of redirection policies: static name-to-address mapping, round-robin replica selection, and the AS-path length policy. We have incorporated them into NetAirt, thus demonstrating its flexibility, and allowing users to switch between them. We have described in detail how the most advanced one, the AS-path length policy, can be implemented. Firstly, BGP routing tables can be exploited to associate an ASN with any IP address, as well as to build a map of the ASes. This map allows to define the AS-path length metric, which is the base of the AS-path length redirection policy.

We have also conveyed two experiments. The first of them investigates the following three issues: the impact of the transport protocol on the Round Trip Time (RTT) of a DNS packet, the overhead introduced by running the round-robin address selection instead of the static name-to-address mapping, and the correspondence between the number of IP addresses returned in a single DNS response and the RTT. The results prove that sending DNS queries over TCP is almost 80% slower than in case of UDP. On the other hand, the overhead introduced by the round-robin policy, and the cost of responding with additional IP addresses turn out to be negligible.

The second experiment evaluates the overhead due to the AS-path length policy. The analysis shows that the majority of Autonomous Systems is located at distance from 3 to 5 from a typical client. We have also measured the time overhead generated by the AS-path length policy when it searches for a closest replica. The results show that exploring the entire map of the Internet that we use for this purpose takes no longer than 3.1 milliseconds, and on average only 0.64 milliseconds. We find this time small compared to the overall RTT for a typical DNS request sent over a wide-area network. Thus, considering the beneficial impact of running the AS-path length policy, we believe that redirecting clients based on network distance calculations can be nearly as efficient as using other, less sophisticated policies.

NetAirt will soon be released for public use. We hope that it can contribute to the development of worldwide-distributed services, and thus help sustaining the continuous growth of the Internet.

Bibliography

[1] *The Apache HTTP Server Project*, http://httpd.apache.org/.

[2] *The RouteViews Project*, http://www.routeviews.org/.

[3] F. Baker, *Requirements for IP Version 4 Routers*, RFC1812, IETF, June 1995, http://www.ietf.org/rfc/rfc1812.txt.

[4] A. Barbir, B. Cain, F. Douglis, M. Green, M. Hoffman, R. Nair, D. Potter, and O. Spatscheck, *Known CN Request-Routing Mechanisms*, Internet Draft, IETF, May 2002, http://www.ietf.org/internet-drafts/draft-ietf-cdi-known-request-routing-01.txt.

[5] D. Barr, *Common DNS Operational and Configuration Errors*, RFC1912, IETF, February 1996, http://www.ietf.org/rfc/rfc1912.txt.

[6] H. Chang, R. Govindan, S. Jamin, S. J. Shenker, and W. Willinger, *On Inferring AS-level Connectivity from BGP Routing Tables*, Submitted to ACM Internet Measurement Workshop, 2001.

[7] M. Colajanni, P. S. Yu, and V. Cardellini, *Dynamic Load Balancing in Geographically Distributed Heterogenous Web Servers*, Proc. of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98), May 1998.

[8] Internet Software Consortium, *Berkeley Internet Name Domain (BIND)*, http://www.isc.org/products/BIND/.

[9] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach, and T. Berners-Lee, *Hypertext Transfer Protocol — HTTP/1.1*, RFC2616, IETF, June 1999, http://www.ietf.org/rfc/rfc2616.txt.

[10] J. Hawkinson and T. Bates, *Guidelines for creation, selection, and registration of an Autonomous System (AS)*, RFC1930, IETF, March 1996, http://www.ietf.org/rfc/rfc1930.txt.

[11] J. Jung, E. Sit, H. Balakrishnan, and R. Morris, *DNS Performance and the Effectiveness of Caching*, Proc. of ACM SIGCOMM Internet Measurement Workshop, 2001.

[12] Z. Morley Mao, C. Cranor, F. Douglis, M. Rabinovich, O. Spatscheck, and J. Wang, *A Precise and Efficient Evaluation of the Proximity between Web Clients and their Local DNS Servers*, Proc. of the USENIX Annual Technical Conference, 2002.

[13] P. R. McManus, *A Passive System for Server Selection within Mirrored Resource Environments using AS Path Length Heuristics*, Applied Theory, Inc., June 1999, http://proximate.appliedtheory.com.

[14] P. Mockapetris, *Domain Names – Concepts and Facilities*, RFC1034, IETF, November 1987, `http://www.ietf.org/rfc/rfc1034.txt`.

[15] P. Mockapetris, *Domain Names – Implementation and Specification*, RFC1035, IETF, November 1987, `http://www.ietf.org/rfc/rfc1035.txt`.

[16] J. Moy, *OSPF Version 2*, RFC1583, IETF, March 1994, `http://www.ietf.org/rfc/rfc1583.txt`.

[17] V. Paxson, *End-to-end Routing Behavior in the Internet*, IEEE/ACM Transactions on Networking Vol. 5, 1997, pp. 601–615.

[18] G. Pierre and M. van Steen, *Globule: a Platform for Self-Replicating Web documents*, Proc. of the 6th International Conference on Protocols for Multimedia Systems, October 2001, pp. 1–11.

[19] Y. Rekhter and T. Li, *A Border Gateway Protocol 4 (BGP-4)*, RFC1771, IETF, March 1995, `http://www.ietf.org/rfc/rfc1771.txt`.

[20] A. Shaikh, R. Tewari, and M. Agrawal, *On the Effectiveness of DNS-based Server Selection*, Proc. of IEEE INFOCOM, April 2001.

[21] W. R. Stevens, *TCP/IP Illustrated, Volume 1*, Addison-Wesley, 1994.

[22] L. Subrmanian, S. Agarwal, J. Rexford, and R. H. Katz, *Characterizing the Internet Hierarchy from Multiple Vantage Points*, To appear in INFOCOM 2002, June 2002.

[23] A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*, Prentice Hall, 2002.

[24] H. Tangmunarunkit, R. Govindan, S. Shenker, and D. Estrin, *The Impact of Routing Policy on Internet Paths*, Proc. of IEEE INFOCOM, April 2001.

[25] H. Hong-Yi Tzeng, *Longest Prefix Search using Compressed Trees*, Proc. of IEEE Globecom, November 1998.

[26] M. Zari, H. Saiedian, and M. Naeem, *Understanding and Reducing Web Delays*, Computer (2001), 30–37.