# Distributed redirection for the Globule platform

Aline Baggio

October 2004
Technical report IR-CS-010

**Abstract**

Replication in the World-Wide Web covers a wide range of techniques. Often, the redirection of a client browser towards a given replica of a Web page is performed after the client's request has reached the Web server storing the requested page. As an alternative, redirection can be performed as close to the client as possible in a fully distributed and transparent manner. Distributed redirection ensures that we find a replica wherever it is stored and that the closest possible replica is always found first. By exploiting locality, latency is kept low. The main disadvantage of distributed redirection is that it relies on a rather static collection of hierarchies of redirection servers that are used during the lookup of replica addresses. It is, however, possible to make both the creation and use of the hierarchies dynamic so that the distributed redirection protocol can be used in a peer-to-peer environment. A hierarchy is then not statically defined anymore but is determined on-the-fly. Moreover, any distributed redirection server is able to compute its position in a given hierarchy based on local knowledge.

## 1 Introduction

Replication in Content-Distribution Networks (CDN) and more generally in the World-Wide Web is often used to allow clients to use the replicas that best suit their needs in terms of network distance, consistency or security. The use of location-dependent URLs in today's World-Wide Web does not facilitate transparent access to the replicated Web pages. Instead, it is often necessary to explicitly *redirect* clients towards a given replica. Redirection is, in most cases, achieved in a *home-based* way using the HTTP protocol or DNS. A client is redirected only after its request has reached the *home* server, that is to say, the host named in the document's URL. The decision where to redirect a client to is therefore centralized.

Centralized redirection mechanisms induce latency, load on the Web page's server and network traffic. Distributed redirection [1, 2] offers a way to redirect a request as soon and as close to the client as possible. The redirection decision can be taken locally at the client machine or, in the worst case, before the HTTP request leaves the client's network. To do so, distributed redirection uses a collection of redirection-enhanced Apache Web servers installed close to the clients. These redirection servers are further organized in a hierarchy. Whenever a redirection server receives a request for a Web page, it first tries to find the address of a replica of the requested Web page in its vicinity and gradually expands the search area if necessary. The gradual search expansion is achieved by forwarding the lookup request along the hierarchy of redirection servers.

A disadvantage of distributed redirection is that it uses static hierarchies of redirection servers: servers cannot come and go and a hierarchy is build off-line using global knowledge. This paper describes how we can make distributed redirection more dynamic and integrate it within the CDN Globule [4].

The paper is organized as follows. Section 2 briefly describes the principles of distributed redirection. Section 3 gives an overview of the characteristics of the Globule environment. Section 4 presents how the distributed redirection protocol and most importantly its hierarchies of redirection servers can be adapted to the needs of the Globule platform. Section 5 further refines the adaptation and concentrates on bringing dynamicity into the hierarchies whether changes are

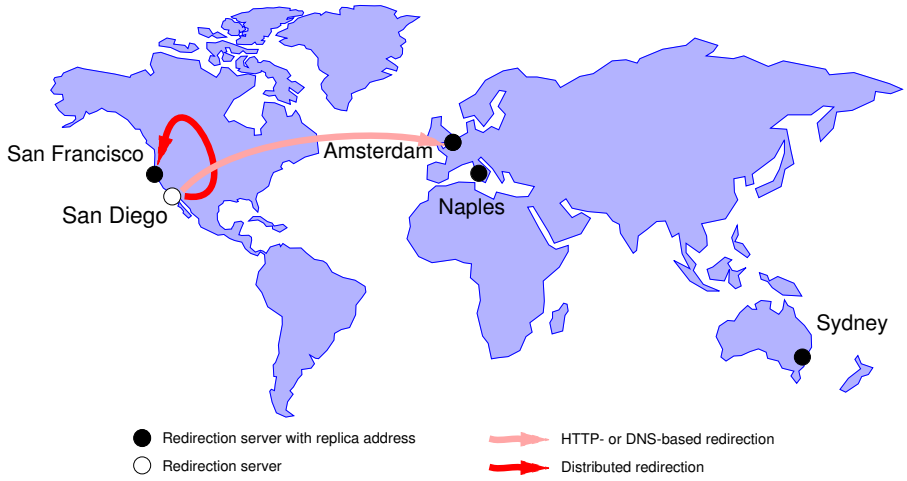


Figure 1: Using distributed redirection to access a close-by replica

needed for load balancing purposes or for fault tolerance. Finally, Section 6 concludes and gives some future work directions.

## 2 Principles of distributed redirection

Considering the disadvantages of HTTP- and DNS–based redirection, we would like to use a redirection method offering a fine granularity in redirection without loss of scalability or transparency. We consider scalability by locality important. First, a request to look for a replica of a Web page has to avoid traveling a long distance. Second, the selected replica should remain the nearest possible to the client browser.

To explain, consider a replicated Web page, referred to as http://www.globule.org/index.html that is available at four Web servers: Amsterdam (the "home" location), Naples, San Francisco and Sydney (see Figure 1). Assume a client browser located in San Diego issues an HTTP request for the Globule page. With the current redirection mechanisms, the request travels, in principle, to the home server in Amsterdam and only there is it redirected to a close-by replica. Locality can be improved for client HTTP requests by using a collection of redirection servers installed close to the clients. In our example, the browser's HTTP request is processed first by its local redirection server in San Diego.

For preserving locality when looking up replicas, a redirection server knows only about pages that are available in its own area. Since the Globule Web page is not locally available, the redirection server in San Diego has to issue a lookup request to find a replica. To keep the communication costs relatively low and preserve locality, a redirection server always tries first to find a requested Web page in its vicinity and gradually enlarge its search area. The gradual search expansion is achieved by organizing the whole collection of servers as a hierarchy and by forwarding the lookup requests along this hierarchy as depicted in Figure 2. The organization of the redirection servers is done on a per-page basis: each page or group of pages has its own separate hierarchical organization of servers that assist in redirecting HTTP requests. To further enforce locality, only leaf servers store addresses of replicas. The information on which leaf server holds which replica is distributed to the relevant intermediate servers so that any replica can be found when issuing a request at any point of the hierarchy. Figure 2 shows the hierarchy for the page http://www.globule.org/index.html.

In our example, the replica lookup request issued by the redirection server in San Diego is further treated as follows. It reaches the parent redirection server USA. The intermediate USA

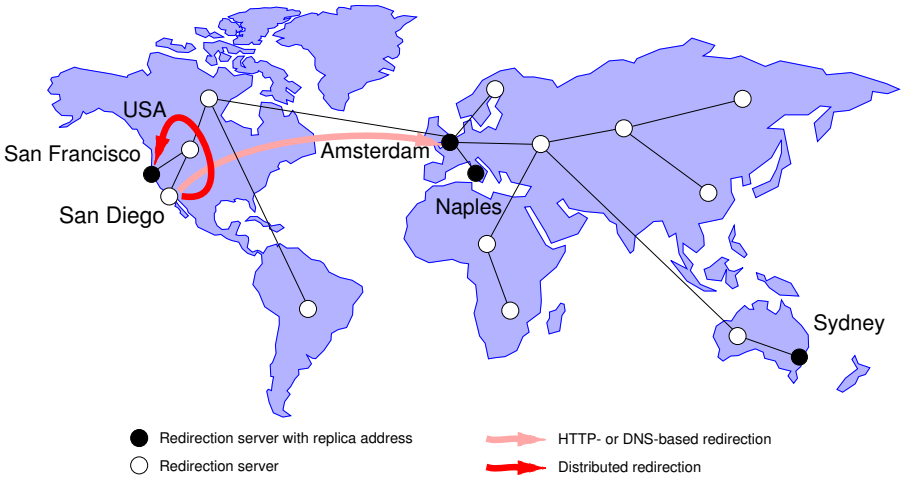


Figure 2: Building a hierarchy of redirection servers

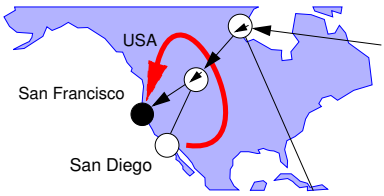


Figure 3: A forwarding pointer at the USA server

server does not have an address for the Web page. However, as shown in Figure 3, it holds a pointer to a child redirection server, here in San Francisco, which is known to have information about a replica of the page. We call this pointer a *forwarding pointer*. The USA server further forwards the lookup request to the redirection server in San Francisco. San Francisco replies with the address of the Web page completing the lookup request. It is the task of the client's redirection server in San Diego to actually retrieve the Web page from the San Francisco Web server and send the results to the client browser. Finally, the San Diego server can decide to cache the replica address. The client browser will benefit from caching, for example, when requesting the inline images of the document.

This scenario shows that by contacting its local redirection server, a client browser implicitly initiates a lookup for a replica at the lowest level of the redirection service. In the best case, the address of a replica can be found at this server (local replica or cached address). If not, the forwarding of the lookup request takes place. Each step up in the hierarchy of redirection servers broadens the search. Having lookups always starting locally at the client site and gradually expanding the search area guarantees that the potential local and close-by replicas are found first. This also guarantees us to find a replica wherever it is stored. The forwarding of the requests along the hierarchy goes no further than necessary and allows us to avoid unnecessary communication with parties that are far apart. In addition, by keeping the number of levels in the hierarchy relatively small, we can also keep the latency minimal when forwarding the requests. A detailed description of the distributed redirection mechanism can be found in [1, 2]. In the following sections, we call *location data* either a replica address or a forwarding pointer.

# 3 Integration within the Globule platform

A potential platform for deploying distributed redirection is the Globule [4] Content-Delivery Network. A CDN such as the well-known Akamai [3] aims at providing scalability through replication: highly popular Web pages or documents such as inline images are replicated close to their clients. The CDN takes care of redirecting the client requests to the most appropriate replica, for example the closest or the less loaded one.

In principle, Globule aims at replicating Web content in the same manner Akamai does. In practice, Globule relies on user-provided resources, such as CPU time or disk space, rather than on a proprietary and dedicated infrastructure. In other words, Globule's goal is to provide a cheap, yet efficient, CDN for small companies, academic institutions or individuals that do not have sufficient traffic to gain Akamai's interest. As a matter of fact, Globule does not have a central controlling institution but is composed of a multitude of small peers, collaborating together and exchanging resources. As such, Globule is called a *user-centered* CDN.

One important aspect of Globule is that replication of Web pages occurs automatically, that is to say without user interaction. Based on the analysis of client requests, a Globule-enhanced Web server can decide to replicate some of its Web pages. It does so by negotiating disk space with a remote Web server. After the page is replicated, client requests need of course to be redirected towards the most appropriate replica. In the current implementation, Globule uses HTTP- or DNS-based redirection. Globule is therefore a good candidate for deploying distributed redirection.

The Globule environment somewhat differs from the (static) environment in which the distributed redirection protocol was developed [1, 2]. The characteristics of this environment are as follows. The Globule platform is composed of a set of peers – million of extended Apache Web servers – distributed world-wide. Some of these servers are promoted to the status of *broker*: they work as intermediates between "regular" Globule servers willing to trade disk space for replicas of Web pages. However, the code running on a broker is in no way different from the code of a "regular" server, which makes the brokers easily interchangeable. For load balancing purposes, each broker is only responsible for a number of Globule servers in a given area or *cell*.

A given Globule server depends of only one broker, namely the closest one available in the server's surroundings. The distance between a Globule server and a broker is estimated in terms of latency. Latencies are not measured for every pair of Globule server and broker but calculated using a latency-based coordinate system [5].

A broker cell therefore carries a notion of locality: Globule servers placed in the same cell are assumed to be close by. However, it is not mandatory that the broker is itself physically located in the cell. For example, we can have a cell with Globule servers are located in Rotterdam whose broker is physically in Amsterdam. In such a case, logical coordinates have to be used to determine which Globule server depends from which broker.

Moreover, a cell also carries a notion of load: a broker has a given capacity. It can only deal with a fixed number of Globule servers. If the load in the cell becomes higher than what the broker can support, the cell is split and a new broker is installed to balance the load. The density of the cells therefore reflects the density of the Globule servers in a given area.

Finally, Globule supports geographic routing. Used in conjunction with the latency-based coordinates, this allows Globule servers to find remote storage space for their replicas. We will see in Section 4 that geographical routing is also of interest for distributed redirection.

The consequences of these environment characteristics are multiple. The number of peers makes it impossible to know or to communicate with all the servers. The peer-to-peer model implies that there is no natural hierarchy in the system anymore and that the degree of dynamicity is high (servers can come and go, disconnect, crash). Moreover, the cell boundaries are moving as the load is balanced. In other words, the hierarchies redirection servers used in the distributed redirection protocol cannot be naturally mapped onto the Globule environment. The hierarchies have to become dynamic to allow servers to join and leave and cell boundaries to change.

The problems we want to solve are as follows. The hierarchies of redirection servers have to be built in a cheap way, without having to know all the redirections servers available world-wide

and with exchanging as few information as possible about the hierarchy. Letting the root of the hierarchy (i.e. the home server) compute the entire hierarchy of redirection servers and having it broadcast to all the participating servers is therefore not an option. Instead, the root has to be off-loaded as much as possible. In line with the principles of distributed redirection, we want each redirection server to be able to compute locally what its position in the hierarchy is. This means that a server should at least be able to discover its parent server.

In the following, we concentrate on three main intertwined topics: how to build a hierarchy of redirection servers in a peer-to-peer environment, how to select a parent server, how to distribute the information about parent servers. In the rest of the article, we will call a Globule (Web) server an Apache Web server supporting both automated replication and distributed redirection.

# 4 Building a hierarchy of redirection servers

Since the peer-to-peer environment does not naturally organize as a hierarchy, we have to build one artificially for use in the distributed redirection protocol. Yet, the way the hierarchy is built and used has to be efficient and scalable. This section shows how these requirements are met.

## 4.1 Basic protocol

Let us go back to our example: a client browser in San Diego requests the Globule Web page referred by http://www.globule.org/index.html. In this case, the root of the hierarchy is the server www.globule.org. In order to find the address of a replica for the Globule Web page, we first have to discover what the leaf and intermediate redirection servers are.

The client browser is unaware of the redirection process and knows nothing about the hierarchy of redirection servers for the Globule Web page. The client's request, however, has to be redirected to a local leaf server. In the distributed redirection protocol, this redirection occurs transparently by having the client's authoritative DNS server forward clients to the leaf server [1, 2]. As for leaf server, we use a Globule broker, namely the one responsible for the client browser's area in San Diego.

In our example, the San Diego leaf server has no replica address and has to contact its parent server USA, that is to say forward the lookup upwards. In the case of a dynamic hierarchy, the parent server may not be known to the leaf and has to be discovered. The underlying idea is to avoid contacting the root server either to find a replica address or the parent server. Instead, we want the leaf server to compute which server its parent is. Furthermore, the computation should only use local knowledge and thus avoid initiating world-wide communication.

The basic protocol to let a leaf server compute its parent for a given root server is as follows. We divide the entire Globule world into a fixed number of cells (NxM grid), as shown in Figure 4 (a). We select one intermediate server per cell: the server the closest to the center of the cell. The center of the entire grid is the root server (i.e. home server). This leads to different hierarchies for different root servers and cares for load balancing. Given the root coordinates, any server can compute the grid, find its own cell and route a message towards the center of this cell. We use here Globule's geographical routing: the target of the message is the coordinates of the center of the cell. The message is gradually routed towards the destination by brokers, until no better move can be made. The parent server is the server where the routed message ended-up, in other words, the closest to the center of the cell (best match). The last stage of the parent discovery protocol consists in having the parent server send back a reply message to the leaf server. In our example, the USA parent server replies to the San Diego leaf server, which can store the IP address of its parent for later use. Note that we assume here that the leaf broker is physically located in the cell where the client is. If it is not the case, we have to use an elected Globule server physically located in the cell instead of the broker.

Building a hierarchy of redirection servers for a given Web page can therefore occur on-the-fly. The root and leaf levels naturally map to the home Web server and the local brokers, respectively. For all the intermediate levels between leaf and root, the intermediate servers are computed once

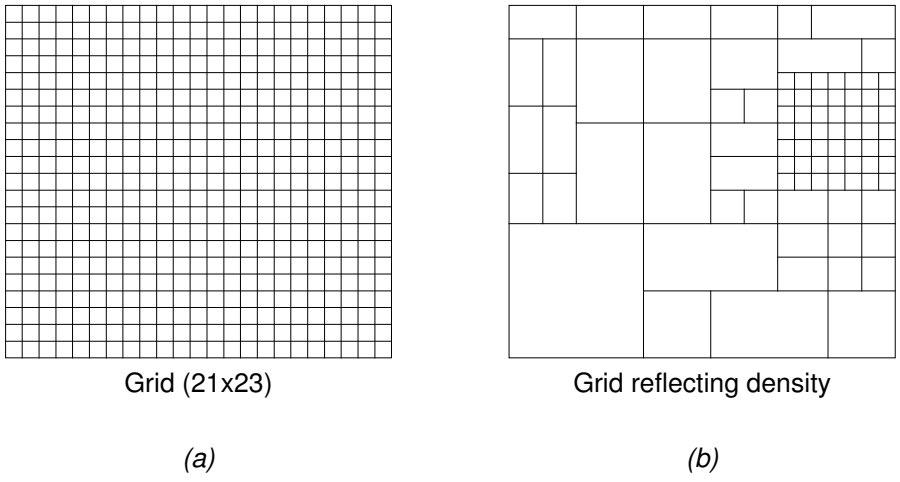


Figure 4: The (NxM) grid and a density grid

whenever necessary, using one grid per intermediate level. The advantage is that we (almost) never contact the root (only as a fall-back mechanism in the case of a network failure or server crash). Nevertheless, the computation requires that the leaf server knows both the root coordinates and its own coordinates and that we use a global coordinate system. In the basic computation protocol, the size of the grid is fixed and global, for each intermediate level. The size of a grid cell being fixed, it is also less easy to balance the load once the grid is build. A grid cell cannot be subdivided. We see in the Section 5 how we can counterbalance these limitations.

### 4.2 Request handling

The way client requests are handled in the Globule environment does not differ much from the original distributed redirection protocol [1, 2]. For all the operations – lookups, inserts and deletes – an extra stage may be added for parent determination before the request is actually forwarded, if need be. The parent determination occurs just once, after what the parent is stored locally. However, a parent server is valid only for a given Web page or Web server. A given Globule server may thus have to compute several parents, valid for different pages.

During a lookup, a Globule server (leaf or intermediate server) has to compute its parent once and store it, as described earlier. In the case of an insert request, the parent servers are immediately computed on the path going from the root to the leaf where the replica address is to be stored. In the case of a delete request, all the parent servers on the path to the root should be known as they were installed during the insert request. Only if the parent server has disappeared should we be forced to recompute a parent during a delete request. Note that the deletion of a non local address is forbidden.

## 5 Refinement to the protocol

We saw in Section 4 that the basic protocol for determining a hierarchy of redirection servers presents some restrictions, mostly regarding the grid size or its dynamicity. This section describes refinements that alleviate the limitations of the basic protocol.

### 5.1 Extensions

We stated in the basic protocol that the grid size (NxM) is fixed per level, well-known and shared by all the Globule servers. Using a newscast protocol, it is possible, however, to adapt the grid

size: new values for N and M are selected whenever needed and distributed to all the servers. This mechanism allows the grid to enlarge or shrink as the load changes. However, since the grid is shared by all the hierarchies of redirection servers, a grid-size readjustment has a dramatic effect. First distributing the new values is costly. Second, all the parent servers for all the hierarchies have to be recomputed. Location data may have to be moved between redirection servers as a result of parent change. Instead, it would be preferable to use one grid size per hierarchy (i.e. root/home server) and per level. In such a case, the grid can be adapted to the load (in terms of requests) of the home server and a grid-size change only impacts the concerned hierarchy. Subsection 5.2.1 describes how the grid size for a given home server can be distributed to the Globule servers.

Another set of limitations relates to load balancing and dynamicity. Ideally, we would like to have the load, in terms of client requests, to be balanced over all the grid cells. In practice, it is not possible to build a grid with cells of different sizes that would reflect the load, as depicted in Figure 4 (b). All the grid cells have to have the same size if we are to compute the grid only from local knowledge. However, the grid and the mapping of intermediate redirection servers onto it naturally reflects the density of Globule servers in a given area, as shown in Figure 5. For example, in Figure 5 (b), the grid cell $C_{a,3}$ has no broker as it is probably located in an underpopulated area (in terms of Globule servers). In such a case, the parent-discovery message routed towards the center of cell $C_{a,3}$ ends-up at the closest possible broker located in the grid cell $C_{a,2}$. This happens automatically: the broker from $C_{a,2}$ cannot route the message anymore. It is therefore the best match. This means that the grid cells $C_{a,3}$ and $C_{a,2}$ form together a virtual cell which is larger than a regular grid cell. In densely populated areas, there will be enough brokers per grid cell, such as in cells $C_{a,1}$ or $C_{e,1}$. The upper limit for balancing the load is therefore the size of a grid cell. In the case of an heavily loaded home Web server, it is therefore better of choose N and M large for the (NxM) grid.

Balancing the load does not only occur by adjusting the cell size altogether with the grid size, but also by adding or removing brokers from the system. This of course has an impact on the hierarchy. An underpopulated area may for example finally get a broker locally installed, by instance cell $C_{a,3}$ in Figure 5 (c). New parent and child servers therefore have to be discovered and installed. In order to support dynamicity in the hierarchy of redirection servers, brokers responsible for one or more grid cells have to exchange location data. For that purpose, we can use the fault tolerance mechanisms already available in the Globule system: to support crashes or disappearing brokers, a Globule broker replicates information about its clients (i.e. regular Globule servers) at one or several of its neighboring brokers. In our case, we can make each intermediate server replicate location data at the broker which is the next best match. For example in Figure 5 (c), the new broker in cell $C_{a,3}$ will replicate its location data at the broker in cell $C_{a,2}$. This means brokers can come and go without a need for deregistration. A new broker may become intermediate server: it simply learns about it when the current broker replicates its location data on it. The new broker may become "primary" (i.e. the best match) for some clients, and the old broker becomes "secondary" (i.e. the next best match), as it is the case in cell $C_{a,3}$ in our example. Note that the brokers at which the location data are replicated (next best match) are not necessarily in the same grid cell.

## 5.2 Protocol details

In addition to the above extensions, this section specifies further the protocol. It describes how a broker can discover the root (i.e. home server) coordinates if necessary and how a broker should deal with a parent server which is not valid anymore.

### 5.2.1 Discover Root coordinates

A broker, for example a leaf server, does not a priori know the root coordinates. Calculating them using latency estimations is computation-intensive and adds unnecessary latency in treating a client's request. In addition, due to potential errors in the latency estimates, the calculated root coordinates may be slightly different from the actual coordinates, that is to say the coordinates the

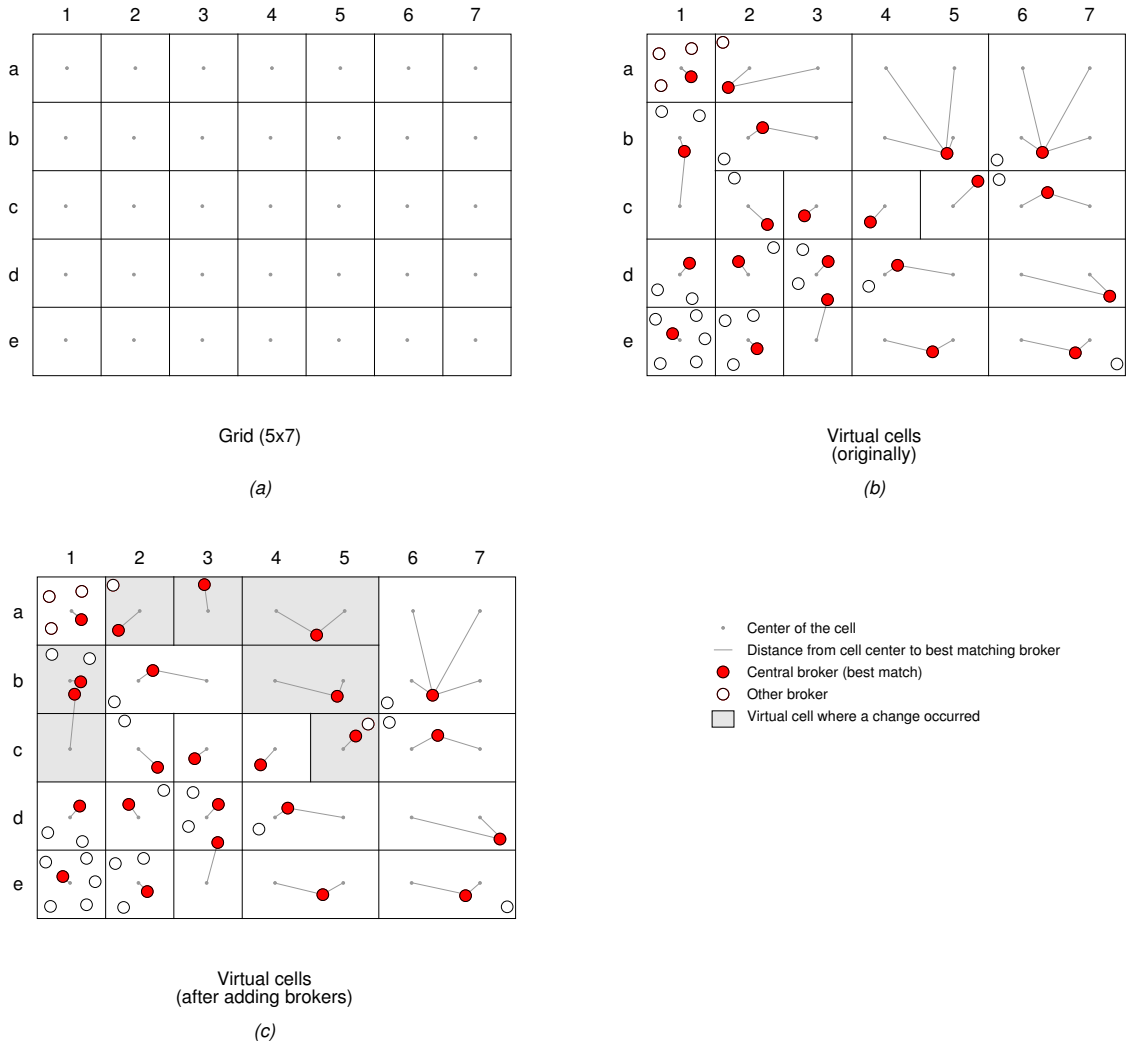


Figure 5: The (NxM) grid and the selected intermediate servers (brokers)

root calculated itself. This shift can lead to problems with the grid, such as misplaced intermediate servers. In other words, the root's grid and the calculated broker's grid may differ. For example, an intermediate server placed by the root in a cell $C_{a,3}$ can be placed in an adjacent cell $C_{a,4}$ by the broker.

Misplacing an intermediate server is not a problem in itself. It becomes a problem only if the broker starts contacting a misplaced intermediate server because it believes it is its parent. If the fault tolerance mechanisms are in place (as described above), the contacted intermediate server can (1) reply to the client broker using replicated location data, (2) ask it the client broker contact its actual parent next time (the actual parent is assumed to be a neighbor of the contacted intermediate server).

Given these problems, we want to avoid calculating the root coordinates and rather retrieve them. A solution is to let the root's authoritative DNS server store the root's coordinates along with its IP address in a TXT record. Note that together with the root coordinates, the authoritative DNS server can also store the grid size values. This makes it possible to use different grid sizes for each root server if need be.

The interaction for retrieving the root coordinates happens as follows:

1. A client browser requests the Globule Web page;
2. The broker (leaf server) gets the client request and resolves the domain name for Globule server. Either the DNS entry is found in the local DNS cache, or the leaf's DNS eventually contacts the root's authoritative DNS server. We consider here the second case (the root's authoritative DNS server has to be contacted);
3. The root's authoritative DNS server replies to client broker with the root IP address and root coordinates along with extra information such as grid sizes;
4. The leaf computes the grid for the root (the root being the center of the grid) and places itself on the grid (i.e. it finds its own cell $C_{x,y}$);
5. The leaf routes a message towards the center of its own cell $C_{x,y}$. The message reaches a server (best match) which acts a intermediate server for cell $C_{x,y}$;
6. The intermediate server sends its IP address to the leaf;
7. The leaf stores the intermediate server as parent;
8. From now on, the lookup can go on as usual: the leaf contacts its parent and the lookup is recursively forwarded upwards if necessary.

The protocol for retrieving the root coordinates can be further optimized. We stated at the beginning of this article that we want to avoid contacting the root server as must as possible. Of course, we also want to avoid contacting the root's authoritative DNS server. In the latter case, we benefit from DNS caching mechanisms but a DNS request may still have to travel up to the root's network. To minimize this problem, we let the leaf servers exchange information. The leaf servers cannot directly exchange parent information (i.e. who is the parent server for a given URL) but they can exchange root coordinates and even already computed grids. This information can be piggybacked in messages aimed at replicating location data between neighbors.

### 5.2.2 Dealing with an incorrect parent server

Since the hierarchy of redirection servers is dynamic, it can happen that a parent server is not valid anymore. For example, a server quit or has crashed. Consider the following example: a broker (leaf server) that has discovered and stored its parent some time ago. The broker sent a lookup to its parent. The broker can encountered the following errors cases:

1. The parent (IP address) is unreachable;
2. The parent is reachable but is not a Globule server anymore;
3. The parent is reachable but is not an intermediate server anymore;
4. The parent is reachable but is not an intermediate server for that broker anymore.

We concentrate here on error case 4. The remaining cases can be solved by using the fall-back mechanism described in [1, 2], that is: the requested Web page is fetched from the root. In case 1, the IP address can be marked as dubious for some time. In cases 2 and 3, the parent IP address has to be removed (i.e. not stored at the broker anymore). At the next request, the broker will have to look for its parent again.

In error case 4, a broker contacts its parent and it turns out to be that this intermediate server is not the broker's parent anymore. In other words, new intermediate servers were installed. This also means that the intermediate server should be able first to detect that the broker is not its child (anymore) (the intermediate server knows child coordinates and its neighboring brokers/intermediate servers). It should be costless and fast for an intermediate server to check whether a broker is really its child server.

The first time a broker contacts its parent, the parent has to install the broker as child. The parent gets the child's IP address and coordinates in the routed message. The parent calculates the grid as well – or has it already precalculated as it is intermediate server for this URL/root server – and checks if the broker is in its own cell or if there is no better parent in the adjacent cells. We saw that in the case of underpopulated areas, a broker can depend from an intermediate server in another cell (see Section 5.1). If the broker is a child server, it is actually installed as child. If not, the message is simply rejected.

If there are changes in the cells around the parent broker or inside its own cell, there could be a parent change. We count two possible changes: a new broker was installed, a broker is gone, either in the same cell or in an adjacent one. In both cases, the child servers have to be redistributed among the (new) intermediate servers. This can be done lazily when a child request comes in or as soon as the change is detected. The root also has to be informed of intermediate servers that appeared or are gone. Note that an intermediate server has to know initially only the child servers having a replica address.

In the case where a new broker appeared, the new broker can be closer to the center of the cell. This is for example the case in cell $C_{a,3}$ in Figure 5 (c). The child servers located in cell $C_{a,3}$ are transfered to the new broker in cell $C_{a,3}$, while the others keep their parent in cell $C_{a,2}$. This partial information transfer happens if the old and the new broker are not located in the same cell. On the contrary, all the child servers from a cell are transfered to the new broker if both the old and the new broker are in the same cell. We have a simple switch between intermediate servers. This occurs for example in cell $C_{c,5}$ in Figure 5 (c).

Note that adding a new intermediate server in a cell does not necessarily mean that the old server is not used anymore. For example, in Figure 5 (b), both cells $C_{b,1}$ and $C_{c,1}$ are using the same intermediate server in cell $C_{b,1}$, forming together a virtual cell. In Figure 5 (c), since a new broker was installed in cell $C_{b,1}$, a server switch occurred for those of the child servers located in cell $C_{b,1}$. The child servers located in cell $C_{c,1}$ remained dependent from the old intermediate server in cell $C_{b,1}$. This server is still the best matching broker, although it is located in a neighboring cell.

In the case where a broker is gone, if the broker disappeared without notice, we let the fault tolerance protocol handle the problem first. The location data is already replicated and is still available in the system. After some time, the faulty intermediate server has to be removed from the hierarchy, or at least from the leaves. If the broker gracefully shuts down, the root and the child servers can be warned and the location data replicated to the new parent server(s). The removal of a broker acting as intermediate servers leads to the same reshuffling of child servers as the addition of a broker, take for example the case where we would go back from the state in Figure 5 (c) to the state in Figure 5 (b). For both the addition and the removal of a broker, the child reshuffling can happen lazily, taking benefit of the fault tolerance protocol.

Whenever it encounters errors about a given intermediate server, a broker (leaf server) is supposed to (1) empty the parent field for this Web page; (2) start a fall-back mechanism: contact the root to get the page and in parallel start to compute the parent again. The fall-back mechanism is used here not to let the client wait while the broker tries to (re)discover its parent. We can also decide to let the client wait and rather avoid contacting the root.

# 6 Conclusion and future work

The redirection mechanisms used in today's World-Wide Web in CDNs such as HTTP-based redirection or DNS-based redirection exhibit characteristics that make them not fully satisfactory. The main concern is that with any of these methods, a *home-based* approach is used. The request of a client is in most cases redirected only after it has reached the Web page's home location. Not only does this put a load on the home Web server and does it generate traffic on the network, it also induces latency that can be perceived by the end user. The redirection can, however, be fully distributed while preserving locality: the redirection decision has to take place as locally to the client as possible and the selected replica of the requested Web page has to remain close to the client. In such a way, we avoid unnecessary communication for both finding a replica and contacting it. Latency is kept low.

The protocol for enabling distributed redirection makes use of a world-wide collection of redirection servers organized as a collection of hierarchies, one per Web page or group of Web pages from the same leaf domain. Leaf servers store addresses of replicas and perform lookup requests on behalf of clients. Using the hierarchy, a lookup request can search for addresses of replicas of Web pages while gradually enlarging the search area. The lookup finds replica addresses regardless of where they are stored, while retaining locality: the search area is not enlarged any further than necessary. Together with HTTP- and DNS-based redirection, distributed redirection is a good candidate for integration in the Globule user-centered CDN.

The distributed redirection mechanism was originally developed in a rather static environment that did not met the requirements of the peer-to-peer environment in use in Globule. Most importantly, the hierarchies of redirection servers did not map naturally onto the peer-to-peer model. We devised a protocol to keep the advantages of the distributed redirection hierarchical searches but yet make the creation of the hierarchies lightweight and more dynamic. Given a home server, any distributed redirection server is able to compute its position in the hierarchy using mostly local information. For each level in the hierarchy, a (NxM) grid is used along with latency-based coordinates. A server is able to position itself in a grid cell $C_{x,y}$ and find its parent server, i.e. the server the closest to the center of the cell $C_{x,y}$.

This simple yet efficient positioning in the hierarchy is achieved on an on-demand basis: whenever a client requests comes in, the parent server is computed if need be. The operation is repeated for any intermediate level of the hierarchy. In addition to its simplicity, the parent-discovery protocol has interesting properties regarding dynamicity, fault tolerance and scalability.

Although work was already done regarding the simulation and implementation of the distributed redirection mechanism in a static environment, few studies have been conducted in a peer-to-peer environment. Future work directions encompasses thus the integration of the distributed redirection protocol as Apache module in the Globule platform, trace-based simulations of the protocol in the Globule environment with server creation, graceful removal, crashes, and deployment in a real-life system onto the already existing Globule servers.

# References


[1] A. Baggio and M. van Steen. Distributed redirection for the World-Wide Web, 2004. Submitted to Elsevier Computer Networks.

[2] A. Baggio and M. van Steen. Distributed redirection for the World-Wide Web (extended version). Technical Report IR-CS-009, Vrije Universiteit, Oct. 2004. http://www.cs.vu.nl/globe/techreps.html.

[3] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, September-October 2002.

[4] G. Pierre and M. van Steen. Globule: a platform for self-replicating Web documents. In *6th Int. Conf. on Protocols for Multimedia Systems*, pages 1–11, Enschede, The Netherlands, Oct. 2001.

[5] M. Szymaniak, G. Pierre, and M. van Steen. Scalable cooperative latency estimation. In *Tenth International Conference on Parallel and Distributed Systems (ICPADS)*, Newport Beach, CA, USA, July 2004.