Design and Implementation of a User-Centered Content Distribution Network

Guillaume Pierre and Maarten van Steen
Vrije Universiteit
De Boelelaan 1081a
1081HV Amsterdam
The Netherlands
{gpierre,steen}@cs.vu.nl

Abstract

Replicating Web documents at a worldwide scale can help reduce user-perceived latency and wide-area network traffic. This paper presents the design and implementation of Globule, a platform that allows Web server administrators to organize a decentralized replication service by trading Web hosting resources with each other. Globule automates all aspects of such replication: document replication, selection of the most appropriate replication strategies on a per-document basis, consistency management and transparent redirection of clients to replicas. To facilitate the transition from a non-replicated server to a replicated one, we designed Globule as a module for the Apache Web server. Therefore, converting Web documents should require no more than compiling a new module into Apache and editing a configuration file.

1. Introduction

Large-scale distributed systems often address performance and quality-of-service issues by way of caching and replication. In the Web, content delivery networks (CDNs) such as Akamai and Digital Island have emerged as a viable solution to achieve scalability through replication [1]. In this approach, content is replicated to places where user demand is high. Content itself can vary from simple static pages to bandwidth-demanding video streams.

Content delivery networks are quite popular among administrators of large Web sites. However, servers holding open-source content and small businesses may prefer a cheaper, yet efficient, solution. We believe that such users are willing to contribute their unused resources to the community in exchange for improved performance for their own site. We propose a decentralized scheme where the owner of a Web site can accept to host replicas from other sites, and obtain in return the ability to deploy replicas of his own documents at remote places. This allows administrators to independently organize a service similar to that of commercial CDNs. This approach is attractive, since it allows one to acquire valuable remote resources in exchange for relatively cheap local resources.

Globule is a user-centered content delivery network that our group is developing [6]. It allows Web servers to host each other's replicated Web documents. To favor integration into existing Web systems, it is designed as a module for the popular Apache server. Making use of our system should require no more than compiling a new module into Apache and editing a configuration file.

This paper presents the design and implementation of Globule, together with performance measurements. Unlike most Web replication systems, Globule does not apply a fixed replication policy to all documents. As we have shown in previous research, there is no single policy that is best in all cases [5]. This statement is true even for simple Web documents that are constructed as a static collection of HTML files, images, icons, and so on. As a consequence, Globule contains a multitude of replication policies, and associates each document with the policy that suits it best. This is realized with an object-based approach in which each document is encapsulated in an object that is fully responsible for its own distribution. In other words, each Web document is considered as an object which does not only encapsulate its state and operations, but also the implementation of a replication policy by which that state is delivered to clients. This allows a document to monitor its own access patterns and to dynamically select the replication policy that suits it best. When a change is detected in access patterns, it can re-evaluate its choice and switch policies on the fly [7].

We evaluate the relative performance of Globule versus Apache, and show that Globule has a constant overhead of

about 200 μs per request, which accounts for at most 10% of the total request latency. The throughput of Globule is also acceptable, in the worst case between 5% and 17% below that of unmodified Apache.

Whereas [6] described the overall architecture of Globule, in this paper we concentrate on the details of its design and implementation, in particular as an Apache module. Our main contribution is that we demonstrate how a user-centered CDN can be developed with existing components, and that it supports per-document replication policies at virtually no cost.

This paper is organized as follows: Section 2 describes our document and server model; Section 3 details the implementation of Globule; Section 4 presents a performance evaluation; finally, Section 5 discusses related work and Section 6 concludes.

2. The Globule model

Our system is made of servers that cooperate in order to replicate Web documents. This section describes our document and server models.

2.1. Document model

In contrast to most Web servers, we do not consider a Web document and its replicas only as a collection of files. Instead, we take a more general approach and consider a document as a physically distributed object whose state is replicated across the Internet. All replication mechanisms are hidden from clients behind the object's interfaces. There is one standard interface containing methods such as `get()` and `put()` to allow for delivering and modifying a document's content.

The design of Globule is inspired by that of Globe, a platform for large-scale distributed objects [10]. Its main novelty is the encapsulation of issues related to distribution and replication *inside* the objects. In other words, an object fully controls how, when, and where it distributes and replicates its content.

We have shown in previous papers that significant performance improvements can be obtained over traditional replicated Web servers by associating each document with the replication strategy that suits it best [5, 7]. Such per-document replication policies are made possible by the encapsulation of replication issues inside each document.

The selection of the best replication policy for each document is realized internally by way of trace-based simulations. Replicas transmit logs of the requests they received to their master site. At startup or when a significant access pattern modification is detected, the master re-evaluates its choice of replication strategy. To do so, it extracts the most recent trace records and simulates the behavior of a number of replication policies with this trace. Each simulation outputs performance metrics such as client retrieval time, network traffic and consistency. The "best" policy is chosen from these performance figures using a cost function. More details about these adaptive replicated documents can be found in [7].

2.2. Cooperative servers

One important issue for replicating Web documents is to gain access to computing resources in several locations worldwide (CPU, disk space, memory, bandwidth, etc.). On the other hand, adding extra resources locally is cheap and easy. Therefore, the idea is to trade cheap local resources for valuable remote ones. Server administrators negotiate for resource peering. The result of such a negotiation is for a "secondary server" to agree to allocate a given amount of its local resources to host replicas from a "primary server." The primary server keeps control of the resources it has acquired: it controls which clients are redirected to which secondary server, which documents are replicated there and which replication policies are being used.

Of course, servers may play both primary and secondary server roles at the same time: a server may host replicas from another server, and replicate its own content to a third one. We use these terms only to distinguish roles within a given cooperation session.

Each document is made of one primary replica located at its primary server, and a number of secondary replicas located at various secondary servers. In our model, all document updates are performed at the primary replica, and then propagated to the secondaries according to the document's own replication policy.

2.3. Security issues

The design of Globule raises two different security issues: security of a secondary server against malicious replicas, and security of a primary server against malicious secondary servers.

For a secondary server, hosting a replica from a remote site is acceptable only if the replica cannot interfere with its other operations. In particular, foreign code should be isolated so that in cannot compromise the security of the server. In the case of Globule, no code is ever shipped between servers. All replicated objects belong to the same class, namely the static document class. Every Globule server contains an implementation of this class, so it is not necessary to ship it. This is also true for policy objects: every Globule server contains an implementation of every policy that is likely to be used, so it is enough for a primary

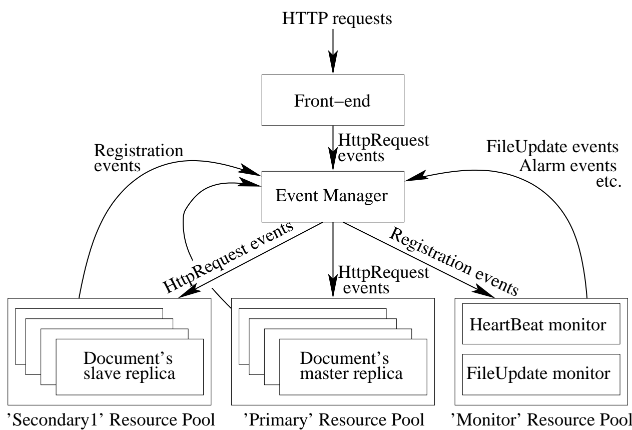


Figure 1. Server Architecture

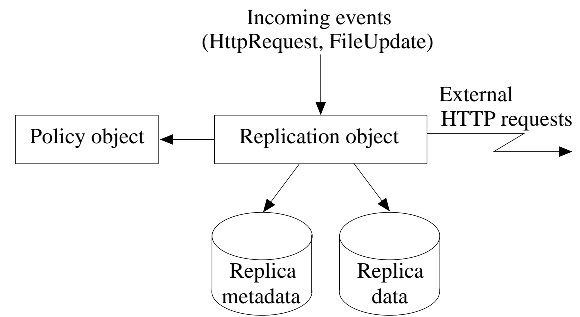


Figure 2. Document Replica Architecture

to associate each of its documents with the identifier of the replication policy that should be used.

The other issue is that a primary server must make sure that secondary servers are trustworthy, so that, for example, they do not deliver modified documents to the clients. This is a very difficult problem, because the primary cannot control its secondaries' behavior. We currently address this issue by requiring administrators to setup cooperations explicitly with partners whom they trust. We plan to relax this model by allowing servers to negotiate peering relationships autonomously, and by using a trust model to address the associated security risk.

3. System architecture

We first describe the overall architecture of Globule, then we show how this architecture has been implemented as a module for the Apache Web server.

3.1. Internal architecture

3.1.1. Server architecture

Figure 1 shows the general architecture of a Globule server. The system is based on events: events can represent an incoming HTTP request or a variety of internal signals such as a registration to a component to monitor file updates or a time-triggered alarm.

All document replicas and monitors can receive events. Monitors may receive registration events. For example, a document can register to the HeartBeat monitor in order to be sent alarms at a given periodicity. Document replicas can receive HttpRequest events which represent incoming HTTP requests, as well as alarms sent by monitors.

Event receivers that belong together are grouped into logical entities called resource pools. Every Globule server contains two standard resource pools. The first one contains monitors such as the HeartBeat monitor to send periodic alarms and the FileUpdate monitor to send alarms when a specific file has been updated. The second resource pool contains all objects representing primary replicas of local documents.

Globule servers can have an arbitrary number of additional resource pools, each one of them containing the set of secondary replicas belonging to a specific primary server. Figure 1 shows a server that contains a resource pool for secondary replicas from site 'Secondary1', in addition to the two standard resource pools.

The Event Manager is in charge of delivering events to their destination. Each event is addressed to a specific element of the system using a two-level scheme. Addresses are made of the name of the resource pool of the destination, and a resource-pool-dependent name that identifies the final destination.

A front-end component is in charge of transforming incoming HTTP requests into HttpRequest events. Other types of events can be sent by any system component, under the condition that it provides a destination address. Some destination addresses are fixed in the system, such as those of monitors. Document addresses are made of the name of their primary server and their URL. For example, the internal address for document `http://www.foo.com/bar` will be the pair `(pool="www.foo.com";name="http://www.foo.com/bar")`.

3.1.2. Document replication

The internal architecture of a replica is depicted in Figure 2. The core of a replica is formed by the replication object. This object receives incoming HttpRequest events directed at the replica and ensures that the appropriate actions are taken to allow the replica data to be delivered in the response. Such actions can range from fetching a fresh copy of its document at the primary server, compare the local copy to the primary, registering to or sending invalida-

tions, or even simply doing nothing.

The nature of tasks that must be performed before a delivery takes place is dictated by the replication policy object. There are different kinds of policy objects, each representing a specific policy. All policy objects implement a standard interface used by the (generic) replication object. Basically, this interface allows the replication object to ask the policy object for instructions each time a request is received. Depending on the policy it represents, the policy object then returns a list of actions that must be performed before the delivery can take place. The actions themselves are performed by the replication object.

Globule contains a number of simple policies, such as "TTL." This policy allows a secondary replica to deliver a copy to a requester without any consistency check during a fixed amount of time since a fresh copy had been fetched. If the period has expired, a consistency check is required. To implement this policy, the policy object of each secondary replica must simply maintain the date of the last retrieval for that document. When requested, it checks whether the delay has expired. It subsequently allows the immediate delivery of the local copy, or otherwise first requests the primary to check for consistency. The policy object located at the primary replica is even simpler: it allows all transfers without any former action.

A more complex policy is "Invalidation." In this scheme, the primary replica sends a message to all registered secondary replicas when the document is updated so that they drop their outdated copy. To do so, each primary replica must maintain a list of its secondaries. This list is constructed dynamically: when the primary's policy object receives a request from a not-yet-registered secondary, it extracts the secondary's callback address from the request and adds it to its local list. A similar mechanism is used when a secondary replica is destroyed: a special request is sent to the primary replica to remove the deleted replica from the invalidation list. In addition to maintaining a list of secondaries, the primary replica must also subscribe to events from the FileUpdate monitor. Whenever the file containing the document is updated, the monitor sends a FileUpdate event to the master replica. Upon reception of this event, the replication object requests its policy object for the list of registered secondary replicas to send them an invalidation. Keep in mind that this is only one possible implementation, and that a server may contain several invalidation policies which have different approaches to issues such as unreachable replicas and invalidation propagation among a large number of replicas.

Policy objects can use arbitrary internal state to perform their task. Examples include no state at all, the date of the last consistency check, and a list of secondary replica addresses. Each replica is given a 'meta-data' file to store this internal state. This is necessary when unloading replica objects from memory. Objects are then given a chance to save their state, which is used again when the object is brought back into memory. Such state changes take place when the server needs to reclaim memory, or when the server is being stopped. State changes are discussed in more detail in the next section.

3.1.3. Resource management

Maintaining replicated documents requires more resources than hosting regular non-replicated documents. To achieve replication, we use replication and policy objects to handle requests to replicas; replicas are given a meta-data file to store their internal state; and finally, secondary servers must store copies of document replicas. These elements obviously require storage in memory or on disk, which brings us to resource management.

Two issues have to be addressed. First, servers must be able to control the amount of disk storage that they use for secondary replicas. We expect that the negotiation involved between the administrators of a primary and a secondary server will decide on a disk quota that can be used to host replicas. However, in many cases this quota will be significantly smaller than the total set of documents that may be replicated there. To address this problem, each resource pool at a secondary server keeps track of the storage space its replicas are using, and maintains its usage below the quota. This is done using standard replacement policies: when the disk usage is rising above quota, then the least recently used replicas are simply deleted.[1]

The second issue is that of memory management. Both primary and secondary servers must maintain replication and policy objects in memory across multiple requests to avoid the overhead of (un)loading them at each request. It is therefore necessary to control the number of loaded objects so that they do no not exceed the memory capacity of the server. The management of loaded objects is again realized on a per-resource pool basis: each resource pool is given a maximum number of loadable documents. A replacement mechanism similar to that of disk resources is in charge of unloading replicas from memory when necessary. Note that these objects can save their state to disk before being unloaded. This allows to unload objects representing primary as well as secondary replicas.

To keep track of resources used by replicas, resource pools provide replicas with an allocation layer that intercepts all resource-consuming or -releasing requests. In par-

[1] Deleting arbitrary secondary replicas is acceptable because they can always be re-created from the primary server. On the other hand, one should not delete primary replicas, since they hold irreplaceable data. We do not consider this as a problem, since the storage space for holding primary replicas would be used on this server anyway, even if it was not replicated. This is the reason why primary resource pools are given no disk quota.

ticular, this layer allows replicas to request the creation of new files, as well as read, write, and release them. By doing so, resource pools can keep track of their current disk usage while giving all freedom to replicas to behave according to various policies. In addition, resource pools keep track of resource ownership. This allows them to prevent replicas to access each other's resources, and to release all resources associated to replicas that are being destroyed.

3.2. Globule as an Apache module

3.2.1. The Apache module model

Apache is an HTTP server structured as a set of modules. The original distribution contains a number of modules, but third-party modules can be provided as well. This enables one to easily add new features [3].

The treatment for each request is decomposed into several steps, such as access checking, actually sending a response back to the client, and logging the request. Modules can register handler functions to participate in one or more of these steps. When a request is received, the server runs the registered handlers for each step. Modules can then accept or refuse to process the operation; the server tries all the handlers registered for each step until one accepts to process it.

The architecture of Apache provides us all the tools necessary to implement a replication module: Globule can in particular intercept requests before being served by the standard document delivery modules to let replication objects check for consistency. Likewise, servers can communicate with each other by HTTP to transfer document copies or invalidations.

3.2.2. MPMs and memory management

One problem that we ran into when implementing Globule concerns the interaction between Apache's multiprocessing and Globule's memory management.

Apache implements several strategies to treat concurrent requests, called Multi-Processing Modules (MPMs). Depending on the operating system, the server is automatically compiled with the MPM that works best for it. Examples of MPMs are a module that treats each request in a separate process, one that implements multiple threads inside a single process, and a hybrid MPM that maintains multiple processes, each of which contains multiple threads.

The choice of one MPM over another should be transparent to all other modules. However, Globule is a special case in that respect because it needs to maintain objects in memory across multiple requests. These objects must be accessible to any request-serving thread or process, even if they were created by another thread or process. The problem is that by default memory is shared among threads of the same process, but not among different processes.

The solution is obviously to make use of shared memory. However, objects that need to be shared are of various sizes, and they can be created or destroyed at any time. This makes it difficult to use simple structures like an object table allocated in shared memory. This lead us to implement our own shared memory management scheme.

At startup, Globule creates a chunk of shared memory where shared objects will be stored. It is associated with its own memory allocator similar to that of the standard C libraries, except that it (de)allocates pieces of the shared memory chunk.

3.3. Client redirection

Like all CDNs, Globule must direct client requests to the replica that can best serve them. It does so by means of "DNS redirection": before sending an HTTP request, the client needs to resolve the DNS name of the service. The DNS request eventually reaches the authoritative server for that zone, which is configured to identify the location of the client and return the IP address of the replica closest to it [4].

Globule incorporates such a custom DNS server as part of its implementation. Although Apache has originally been designed to handle only the HTTP protocol, its versions starting from 2.0 allow one to write modules that implement other protocols. It is therefore possible, with minimum changes to Apache, to integrate a DNS server *inside* Apache. More details on this aspect can be found in [9].

4. Performance evaluation

We evaluate the overhead introduced by Globule in addition to regular document delivery by Apache. The experimental setup consists of two dual-processor 1GHz Pentium III machines connected by 100Mbit/s Ethernet. The first machine is used to run Apache either with or without the Globule module. The second machine is used to send requests to the server.

We measure Globule's overhead independently from the particular effects of specific replication policies by manually selecting a policy that always allows the delivery of the local replica without any prior action. We then compare the resulting performance with the same requests sent to an Apache server running without Globule. More complex replication policies may of course introduce additional costs, but these costs are already taken into account as one aspect of the cost/benefit analysis that selects replication policies [7].

We compare the relative performances of Globule and Apache from two points of view: request latency mea-

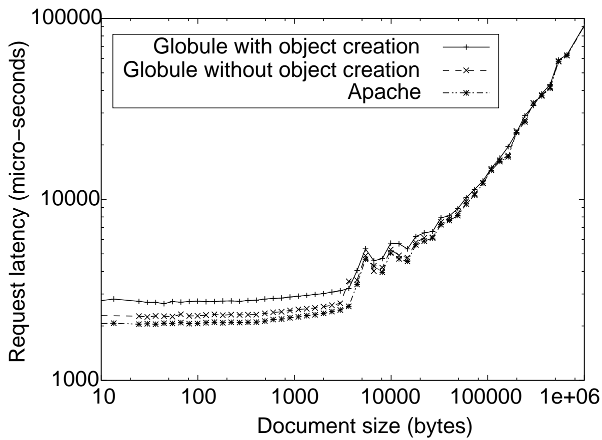


Figure 3. Request latencies

sures reflect the performance as seen by the clients, whereas server throughput reflects the performance as seen by the servers.

We stress that we are interested in measuring the overhead introduced by Globule, and as such concentrate on micro benchmarks for our Apache implementation. The overhead introduced by Globule as a whole is relevant only when considering the effects of individual replication policies, which has been discussed at length in [5]. For this reason, we did not perform wide-area experiments at this point.

4.1. Request latency

We measure the latency of requests as seen by the client, i.e., the duration between when a TCP connection is initiated by the client and when the response has been fully received. To reproduce a realistic request access pattern, our client is configured to reproduce requests taken from the log file of our department's Web server. Document sizes are reproduced as well. To load the server to its maximum capacity, we replay the trace file as fast as possible.

Figure 3 shows the request latencies observed by the client when document size varies. To make the graph readable, we did not plot the performance of each request that was performed, but instead, showed only the median latency values for requests to documents of each size.

Note that because we are making use of a high-speed network, we are measuring pure-server performance. This setup can be considered as a worst-case scenario compared to a realistic wide-area setting.

We split Globule latency measures into two curves. The first one concerns the first request that Globule receives for a document, and the second one concerns all subsequent requests. When receiving the first request for a given document, Globule must create a replication object, a policy object and a meta-data file to hold the replica's internal state. As can be seen in the graph, these operations have a fixed cost of about 400 μs. Note that this cost is incurred only once for each document.

When no object creation is required, the latency of requests to Globule replicas is approximately 200 μs higher than that of the same requests to unmodified Apache. This fixed cost is mostly due to Globule finding the replication object that must be forwarded each HttpRequest event.

The additional cost for delivering Globule replicas is primarily visible only for small documents. When the document size is greater than a few kilobytes this cost becomes negligible compared to the time needed to actually transfer the document over the network. We expect this latency difference between Globule and Apache to become even lower when requests are sent over a wide-area network.

4.2. Server throughput

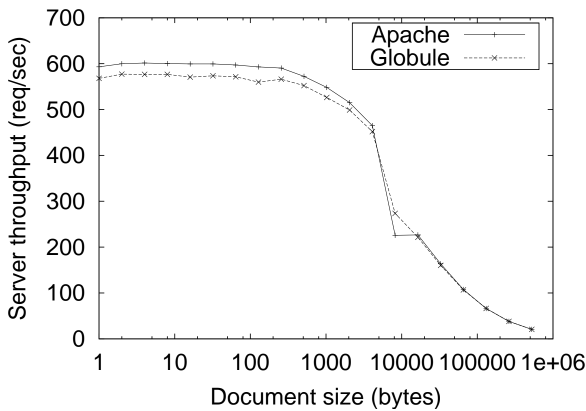
Figure 4 shows the throughput that a Globule server can achieve compared to that of Apache. Throughputs are measured using `ab`, the standard Apache benchmarking tool. We use this tool to send requests repeatedly to documents of different sizes. We measure the throughput of Apache and Globule when being requested with different levels of concurrency.

All graphs show a decrease of throughput when document size increases: obviously, it takes more time for a server to deliver a large document than a small one.

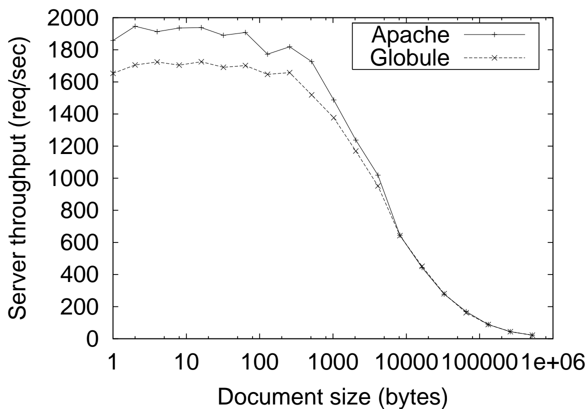
When there is no concurrency between requests, Globule's throughput is very close to that of Apache: for document sizes below 8kB, Globule's throughput is about 5% lower than that of Apache. Throughputs for larger documents show the same effect as in the request latency measurements: the difference becomes negligible because network transfer costs take over any additional cost caused by Globule.

When we increase the concurrency level to higher values, we see that the difference in throughputs becomes larger. The difference is about 5% for a concurrency level of 1, 9% for a concurrency level of 2 and 17% for a concurrency level of 5.

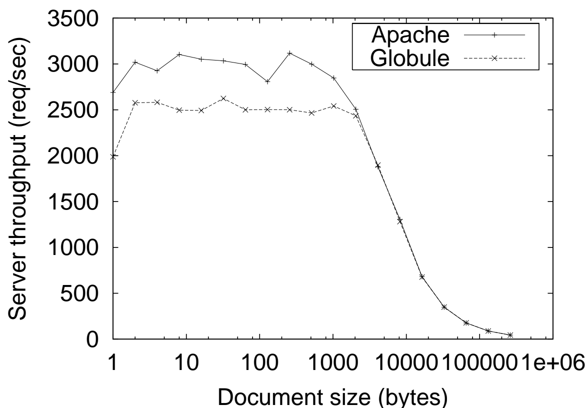
We explain this degradation by the way Globule allocates shared memory. To prevent race conditions, such allocations are serialized by a global mutex. Apache implements such cross-process and cross-thread mutexes by using one interprocess lock, plus one intraprocess mutex per process. (Un)locking the global mutex before (after) each shared memory allocation requires to (un)lock $n + 1$ mutexes. We can therefore expect that shared memory allocations become slower as the server load increases due to request concurrency.



(a) Concurrency level=1



(b) Concurrency level=2



(c) Concurrency level=5

Figure 4. Server Throughput

Like in the latency analysis, these throughput measures reflect a worst-case scenario because they were performed over a high-speed network. In a wide-area environment, the cost for acquiring locks will become negligible compared to the total request latency, and the throughput of Globule will likely be much closer to that of unmodified Apache.

5. Related work

Many systems have been developed to cache or replicate Web documents. The first server-controlled systems have been push-caches, where the server was responsible of pushing cached copies close to the users [2]. More recently, content distribution networks (CDNs) have been developed along the same idea [1, 8]. These systems rely on a large set of servers deployed around the world. Consistency is realized by incorporating a hash value of a document's content *inside* its URL. When a replicated document is modified, its URL is modified as well. This scheme necessitates to change hyperlink references to modified documents as well. In order to deliver only up-to-date documents to users, this scheme cannot use the same mechanism to replicate HTML documents; only embedded objects such as images and videos are replicated.

Globule presents three major differences with CDNs. First, since its consistency management is independent from the document naming scheme, it can replicate all types of objects. Second, contrary to CDNs that use the same consistency policy for all documents, Globule selects consistency policies on a per-document basis so that each document uses the policy that suits it best. Finally, the system does not require one single organization to deploy a large number of machines across the Internet: Globule users are encouraged to trade resources with each other, therefore incrementally building a worldwide network of servers at low cost.

6. Conclusion

We have presented the design and implementation of Globule, a user-centered content delivery network. Globule integrates all necessary services into a single tool: document replication, selection of the most appropriate replication strategies on a per-document basis, consistency management, and automatic client redirection.

Globule is implemented as a module for the Apache server. This will enable administrators of non-replicated Apache servers to switch to replicated documents by simply compiling an extra module in their server and editing a configuration file.

Globule will soon be made available from `http://www.globule.org/` under an open-source license.

References

[1] 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.

[2] J. Gwertzman and M. Seltzer. The case for geographical push-caching. In *Proc. 5th Workshop on Hot Topics in Operating Systems (HotOS)*, Orcas Island, WA, May 1996. IEEE.

[3] B. Laurie and P. Laurie. *Apache: The Definitive Guide*. O'Reilly & Associates, Sebastopol, CA., 2nd edition, 1999.

[4] P. R. McManus. A passive system for server selection within mirrored resource environments using AS path length heuristics. Technical report, AppliedThory Communications, Inc., June 1999. `http://www.gweep.net/~mcmanus/proximate.pdf`.

[5] G. Pierre, I. Kuz, M. van Steen, and A. S. Tanenbaum. Differentiated strategies for replicating Web documents. *Computer Communications*, 24(2):232–240, Jan. 2001.

[6] G. Pierre and M. van Steen. Globule: a platform for self-replicating Web documents. In *Proceedings of the 6th International Conference on Protocols for Multimedia Systems*, LNCS 2213, pages 1–11, Oct. 2001.

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

[8] M. Rabinovich and A. Aggarwal. RaDaR: A scalable architecture for a global Web hosting service. In *Proceedings of the 8th International World-Wide Web Conference*, May 1999.

[9] M. Szymaniak. A DNS-based client redirector for the Apache HTTP server. Master's thesis, Vrije Universiteit, Amsterdam, The Netherlands, July 2002. `http://www.globule.org/`.

[10] M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7(1):70–78, January-March 1999.