RepWeb: Replicated Web with Referential Integrity

Luís Veiga
INESC-ID Lisboa/IST
Rua Alves Redol, 9
Lisboa, Portugal
luis.veiga@inesc-id.pt

Paulo Ferreira
INESC-ID Lisboa/IST
Rua Alves Redol, 9
Lisboa, Portugal
paulo.ferreira@inesc-id.pt

ABSTRACT

Replication of web content, through mirroring of web sites or browsing off-line content, is one of the most used techniques to increase content availability, reduce network bandwidth usage and minimize browsing delays in the world-wide-web.

The world-wide-web does not support referential integrity, i.e., broken links do exist. This has been considered, for some years now, one of the most serious problems of the web. This is true in various fields, e.g.: i) if a user pays for some service in the form of web pages, he requires such pages to be reachable all the time, and ii) archived web resources, either scientific, legal or historic, that are still referenced, need to be preserved and remain available.

Current approaches to the broken-link problem are not able to preserve referential integrity on the web and, simultaneously, support replication and minimize storage waste due to memory leaks. Some of them also impose specific authoring and management systems. Thus, the limitations of current systems reside in three issues: transparency, completeness and safety.

We propose a system, RepWeb, comprised of an application to access and manage replicated web content and an implementation of an acyclic distributed garbage collection algorithm for wide-area replicated memory, that satisfies all these requirements. It supports replication, enforces referential integrity on the web and minimizes storage waste.

1. INTRODUCTION

Replication is widely used on the web today. It is a cost-effective way to allow more simultaneous accesses to the same web content and preserve content availability in spite of network and server failures. Furthermore, it gives users the ability to browse the same content from different locations (possibly very distant geographically from one another) choosing the nearest or fastest one. This technique is commonly called mirroring and there are some tools that perform this automatically[17]. In addition, local replication reduces browsing delays and allows off-line browsing of previously replicated web content. Thus, replication increases resource availability, minimizes transferring times and enables off-line browsing.

1.1 Motivation

Broken links, i.e., the lack of referential integrity of the web, is a classic dangling-reference problem. With regard to the web this has several implications: annoyance, breach of service, loss of reputation, effective loss of knowledge. When a user is browsing some set of web pages, he requires such pages to be reachable all the time and will be annoyed every time he tries to access a resource pointed to from some page just to find out that it has simply disappeared.

As serious as this last problem, there is another one related to the effective loss of knowledge. As mentioned in earlier works, broken links on the web can lead to the loss of scientific knowledge[4]. We dare to say that, in the time to come, this problem can affect legal and historical knowledge, as these areas become more represented on the web.

On one hand, most systems currently supporting replication[13, 15], do not address referential integrity, and consider broken-links as system or application failures that should be exposed to users. On the other hand, current solutions to the problem of referential integrity[1, 6, 7, 11] do not take replication into account. Thus, only some of the existing solutions attempt to enforce referential integrity on the web while being complete, i.e., not only preserving web resources targeted by links but also reclaiming content which is no longer referenced from any root-set (these root-sets may include bookmarks, subscription lists, etc). These solutions[6, 11], however, are not safe in the presence of replication because they were not designed to preserve referential integrity with replication in mind.

1.2 Solution

The purpose of this work is to develop a system that: i) enforces referential integrity on the web; ii) prevents storage waste; and iii) manages to do so correctly in the presence of replication. These three properties must be correctly and efficiently combined. The first property addresses correctness, the second performance requirements, and the third, both. We propose a solution, based on a distributed acyclic garbage collection (DGC) algorithm for wide-area replicated memory[16], that satisfies all these requirements. It sup-
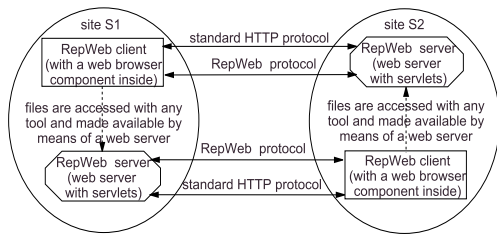
Figure 1: *General architecture of the application.*

ports replication, enforces referential integrity on the web and minimizes storage waste. Furthermore, this algorithm scales well in a wide area replicated memory system as is the case of the web. For ease of deployment, this solution makes use of standard web browsing components for the client application and standard web servers with extensions (e.g.: Java). Users are still able to access any non-RepWeb files available on the web.

Presently, in RepWeb, we are not able to reclaim distributed cycles of unreachable web content. This is due to the algorithm used. We do not address the issue of fault-tolerance, i.e. it is out of the scope of the paper how the algorithm used behaves in the presence of communication failures and processes crashes. Nevertheless, the algorithm used is safe w.r.t. message loss and duplication.

Thus, the contribution of this paper is a system that ensures referential integrity on the web, minimizes storage waste, achieves this correctly in the presence of replication, and scales to wide area networks. The remaining of this paper is organized as follows. In Section 2 we present the proposed architecture that was developed. The DGC algorithm used is briefly described in Section 3. In Section 4 we present a prototypical example of application use. Section 5 highlights some of the most important implementation aspects. Section 6 presents some performance results. The paper ends with some related work and conclusions in Sections 7 and 8, respectively.

2. ARCHITECTURE

In order not to impose the use of a new, specific, hypermedia system, the architecture proposed is based on regular components used in the world wide web or widely accepted extensions to them. RepWeb is designed using a client-server architecture, illustrated in Figure 1, with: servers - web servers with server extensions, namely Java servlets and; clients - applications using web browsing components, with replication code that interacts with server extensions.

The entities manipulated by the system are web resources in general. These come in two flavors: i) HTML content documents that can hold text and references to other web resources, and ii) all other content types (images, sound, video, etc.). Resources of both types can be replicated and are preserved while they are still reachable. The latter, however, cannot contain references to other resources and are viewed, by the system, as leaf-nodes in a web resources graph. Thus, memory is organized as a distributed, partially replicated graph of web resources connected by references (in the case of the web, these are URL links).

We considered, mainly, four kinds of web usage: i) web browsing without replication, i.e., standard web usage; ii) web browsing with replication when desired explicitly by the user, in a page-per-page basis; iii) web browsing of web-pages replicated on-demand, i.e., in which parts of the web graph are incrementally and automatically replicated, as they are rendered, for future off-line browsing or to minimize download delays; and iv) site mirroring, i.e., complete replication of significant parts, or the whole, of web sites. These replicas can, after edition (translation, adaptation, summary, etc.) or any other content authoring activity performed with any tool, be made to diverge from the master replicas but still be preserved, as well as those referenced from them. Note that this last scenario can not be performed resorting only to off-the-shelf mirroring tools[17].

With regard to the issue of preserving replica consistency, it is not necessary for the system to function, and it may even not be desirable as in the context of some of the authoring activities mentioned above. However, our replication system allows refreshing or updating of replica content, either from or to the master replica.

From the user point of view, the client side of the application is a normal web browser with an extra menu button called "make-replica". This function allows the user to replicate a file into his machine, i.e., to create a stable local replica of the file he is looking at. This file being browsed may be, implicitly, already replicated in the user's computer in some browser's specific directory. However, this copy only acts as cache and not as stable replica. Nevertheless, the fact that this file is already available in the system, though in temporary store, lowers the cost of replicating a page (simple file move or copy).

With this application, a typical user in site S1 browses the web (in this case, web servers supporting the server-side of the application) and makes replicas of some of the pages from site S2. End-users may replicate web resources (HTML documents or any other files) for future browsing. However, if desired, these web resources can then be accessed and/or edited using any other application, possibly making the replicas to diverge. Once ready, this replica can also be made available to the outside world from the user's local server.

Upon request from some user at a different site, the web resource can be replicated again. It can also be used to update the content of its master replica. These replicas may hold references to other (not locally replicated) web resources in site S2. Thus, it is desirable that such resources in site S2 remain available as long as there are references pointing to them. The system ensures that such resources in site S2 remain there as long as they are referenced from some other site. In addition, web resources in site S2, which are no longer referenced from any other site are automatically deleted by the garbage collector. This means that neither broken links nor memory leaks (storage waste) can occur.

3. STORAGE INTEGRITY/MANAGEMENT

To enforce referential integrity and reclaim wasted storage in RepWeb, we made use of a distributed garbage collector for wide area replicated memory[16] and tailored it to the
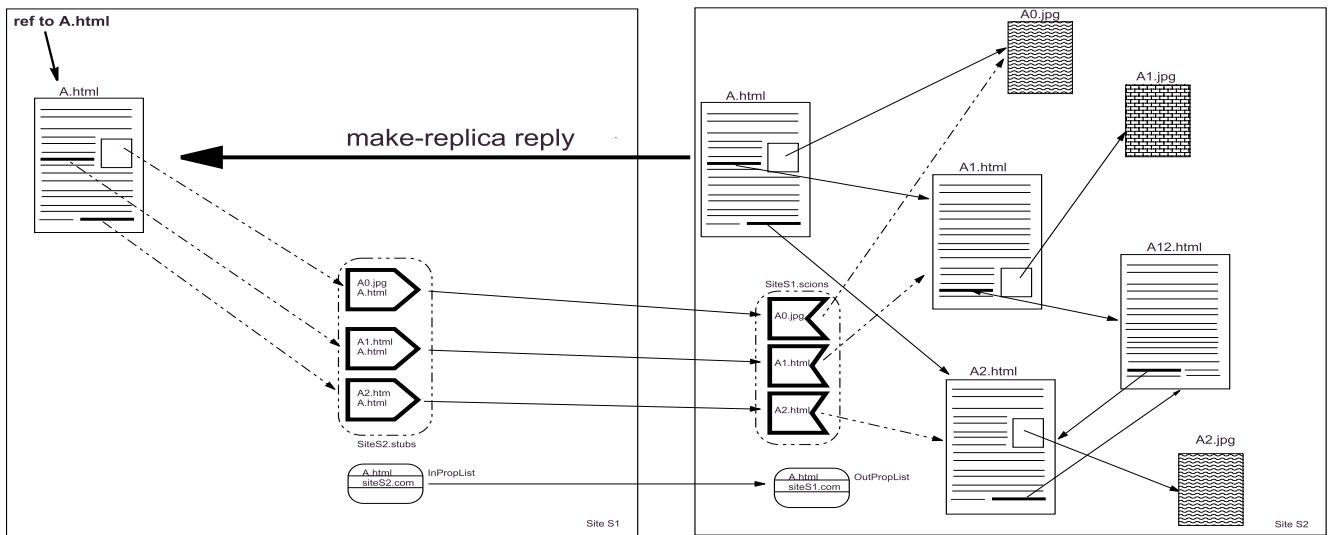
Figure 2: *After first replication*

web. The algorithm is an hybrid of tracing and reference listing. Thus, it cannot collect distributed cycles of garbage without the use of an auxiliary cyclic distributed collector. In each site there are two GC components: a local tracing collector[10], and a distributed collector. Local tracing is performed independently in each site. The distributed collectors, based on reference-counting[2, 12], work together by exchanging asynchronous messages.

The root-set of documents for both the local and distributed garbage collectors in each site is comprised of local roots and remote roots: i) local roots are web documents, created in or, replicated to the site and referenced from a special HTML file (bookmark file) managed by the RepWeb system; ii) remote roots are all web documents that are remotely referenced, i.e., protected by scions (see Section 3.1). These web resources must be preserved even if no longer locally reachable, i.e., reachable from the local root-set. The root-set of the whole system corresponds to the union of the root-sets in all sites.

3.1   GC Structures and Rules

The RepWeb system and RepWeb garbage collector manipulate the following structures to store information about replicated resources and references contained in web pages (see Figure 2):

- An **InPropList** that describes web resources that have been replicated to this site from another site.
- An **OutPropList** that describes web resources that have been replicated from this site to another site.
- A **stub** describes an outgoing inter-site reference, from a document in the site to another resource in a target site.
- A **scion** describes an incoming inter-site reference, from a document in a source site to a local resource in the site.

It is important to note that stubs and scions do not impose any indirection on the access to replicated web pages. They

are simply DGC specific auxiliary data structures. The algorithm obeys to the following safety rules:

- **Clean Before Send Make-Replica-Reply:** Before replying to a make-replica request for an object y from a site S1, y must be cleaned (i.e. it must be scanned in S2 for references) and the corresponding scions created in S2.
- **Clean Before Deliver Make-Replica-Reply:** Before delivering a make-replica reply message from site S2 for an object y to a site S1, y must be cleaned (i.e. it must be scanned in S1 for outgoing inter-process references) and the corresponding stubs created in S1, if they do not exist yet.
- **Union Rule:** A target object z is considered unreachable only if the union of all the replicas of the source objects do not refer to it.

In the prototypical example in Figure 2, the first rule implies the creation of the scions corresponding to files A0.jpg, A1.html and A2.html in site S2 before the make-replica-reply message is sent. The second rule implies the creation of the stubs corresponding to files A0.jpg, A1.html and A2.html in site S1 before the make-replica-reply message is delivered and replication of file A.html is complete. The last rule is presented only for completeness. Details are explained in[16].

We decided to make use of this algorithm for three fundamental reasons: i) it is orthogonal to any protocol that maintains, or not, the replicas coherent among the participating processes, i.e., the DGC does not require replicas to be coherent; ii) it does not require causal delivery to be supported by the underlying communications layer, a fundamental aspect to ensure the DGC algorithm scalability, given that supporting causal delivery in wide area networks is difficult and inefficient; and iii) it is safe in presence of replicated objects.

3.2   Integration with the web

The world wide web owes, a significant part of its success until now, to the fact that it allows clients and servers to be loosely coupled and different web sites to be administrated autonomously. Therefore, RepWeb, while providing interesting properties to a set of adhering sites, namely in the fields mentioned earlier, must not impose total world-wide acceptance in order to function. Integration with RepWeb can be seen from two perspectives, client and server. Regular web clients can freely interact with RepWeb servers to retrieve web content but cannot replicate web resources or interfere with the DGC in any way (e.g.: indexing). Regular web servers can be accessed via RepWeb client application but it will not be possible to replicate, via RepWeb, content residing there since regular web servers cannot fulfill the RepWeb protocol.

We intend to extend our system to collect distributed cycles of unreachable web content. Based on work in[14] we can estimate the importance of cycles in web content. A large proportion of objects are involved in cycles but they amount to a limited, yet not negligible, fraction of storage occupied(12-14%). Furthermore, individual cycles are small both in number of objects as in space occupied.

4. PROTOTYPICAL EXAMPLE

In this section we present a prototypical example of RepWeb usage to explain it in greater detail: initially, there is a web site, site S2, composed of a web resources graph with HTML documents and images. In site S1, the user only has a reference, a URL, to a page A.html in site S2. After browsing its content the user wishes to replicate this document.

In Figure 2, the HTML document A.html has already been replicated and the DGC structures are updated, on each site, to reflect this new situation. More precisely, there are new entries in the sites' InPropList and OutPropList reflecting that a replication has occurred. Furthermore, a set of stubs is created in site S1. Conversely, a set of scions is created in site S2. The scions in site S2 protect the files from being reclaimed by the local collector and the stubs in site S1, representing remote references to documents in site S2, protect the scions from being deleted by the distributed collector.

During the browsing of A.html, the user decides to click on the link to the document A1.html. As in the previous situation, after browsing it, the user decides to replicate document A1.html from site S2 to site S1 and the DGC structures are updated. InPropList and OutPropList are appended with the information about this latter replication and the set of stubs and scions are appended with the references contained in A1.html (see Figure 3). As a result A.html and A1.html are replicated in site S1 and every resource referenced from them, and yet not replicated, is protected by stubs and scions. These stubs and scions referring document A1.html, however, have become redundant because there are duplicated references among the stubs/scions and the In/OutPropList. The system optimizes this information eliminating redundancies as it can be observed in Figure 3, deleting the stub-scion pair crossed.

5. IMPLEMENTATION

The system is implemented in Java. This includes the client code (that uses a regular HTML rendering package) and

file size	# URLS	scan time	stub time	hashtable size	time to serialize
43563	326	38	3	19252	67

Table 1: *Performance results from cnn.com site.*

all the code of the local and distributed collectors. We intend to integrate the client as a component in standard web browsers. The local collector is implemented as a stand-alone application. The distributed collector is implemented by the servlets and by the client. Client-only sites receive DGC messages piggy-backed in replies from servers. To allow incremental replication of web pages, every navigation operation performed by the user is trapped and the system decides whether to feed the rendering component with a replica created before or to perform an HTTP request to get file content. By default, this can be replicated and made available for future use.

The code in the servlets implements the safety rule Clean Before Send Make-Replica-Reply (applied when make-replica is requested); the client code implements the safety rule Clean Before Deliver Make-Replica-Reply (applied when the reply to a make-replica request is received). The implementation of these rules consists of scanning the web pages being replicated and creating the corresponding scions (at the server) and stubs (at the client). The first time a file is replicated, at the server site, its content is scanned, the corresponding scions created, and the enclosed set of URLs is kept in an auxiliary file. Later, if this same page is replicated again, at the server site, it only has to be scanned again if it has been modified after the last scan.

6. PERFORMANCE

In this section we present the most relevant performance results concerning the implementation. We devised two different types of performance measurements: i) to evaluate application and algorithm performance with a large set of files; ii) To evaluate most common usage with two specific sets of files that bound typical browsing sessions. The critical performance results are those related to the implementation of DGC safety rules I and II. Thus, we downloaded a well-known web site (cnn.com) and ran on each file the code implementing the safety rules. All results were obtained in a local 100 Mbits network, with Pentium II PCs with 64 Mb of memory running Windows NT. We downloaded 155 HTML files from the cnn.com (recursive download with a maximum depth of five) web site and obtained for each one the time it takes to: scan it, create the corresponding stubs, and serialize the hash table containing them. In this section, for clarity, we simply refer to the time it takes to create stubs and their size because the same values apply to scions.

In Table 1 we present, for the 155 files set: the mean file size, the mean number of URLs enclosed in each file, the mean time to scan a file, the mean time it takes to create a stub in the corresponding hash table, the mean size of the hash table containing all the stubs corresponding to all the URLs enclosed in a file (that depends on the size of the corresponding URL), and the mean time it takes to serialize a hash table with all the stubs corresponding to a single file.

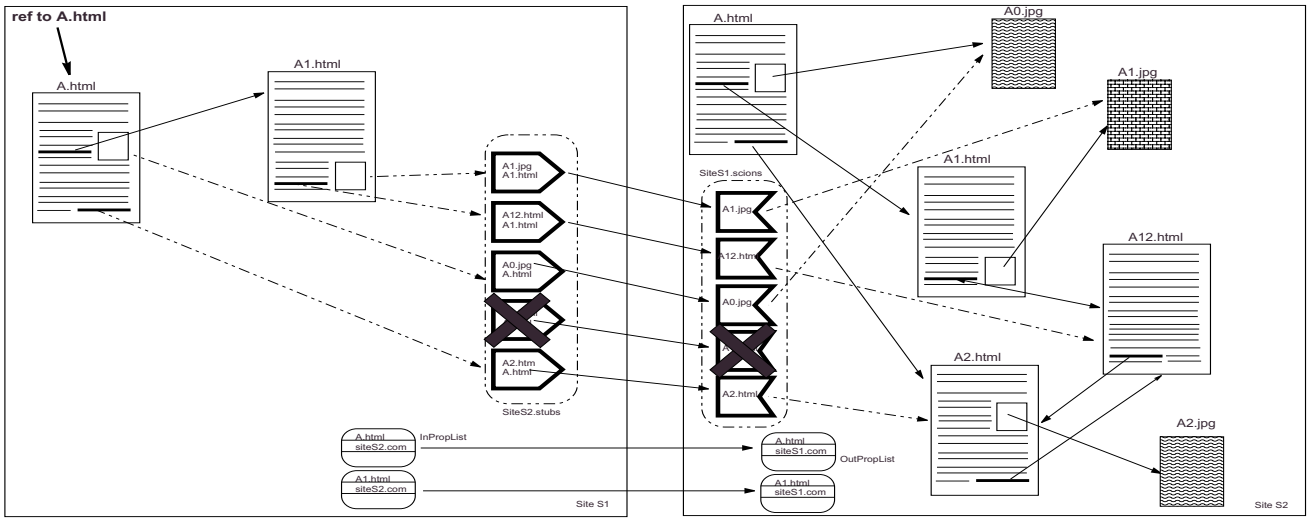However, in a normal browsing session, the user does not

Figure 3: *Optimized structures after second replication*

make replicas of all the files. We expect the user to browse a few top-level pages and then pick one or more branches of the hierarchy and follow them down. Some of these files will be replicated into the user's local computer. So, in order to obtain more realistic numbers, we picked 10 files from the top of the cnn.com hierarchy. These files are mostly entry points to the others with more specific contents. We call this set of files the top-set. We also picked other 10 files representing a branch of the cnn.com hierarchy, world/europe. We call this set of files the branch-set. In Figures 4 and 5 we present, for each file of these sets, the times spent in each of the relevant operations, the space occupied by the files themselves, the URL references enclosed in them, and the Java implemented data structures.

These time measurements are fixed initial costs due to replication, reference scanning and storing, that are dimmed after just a few visits (hits) to the pages. These performance results are worst-case because they assume all the URLs enclosed in a file refer to a file in another site, which is not the usual case. However, they give us a good notion of the performance limits of the current implementation. In particular, the most relevant performance costs are due to the scanning of a file and the serialization of the hash table. We believe that these values are acceptable taking into account the functionality of the system, i.e. it ensures that no broken links and no memory leaks occur. In addition, when a user runs the browser and accesses any web page without making a local replica of any file, there is absolutely no performance overhead due to DGC.

We can also conclude that the size on disk of the hash table containing all the stubs for a file is about half the size of the HTML file. This rather large size is mostly due the size of the URLs which are responsible for about 90% of that size. The stubs relative to the top set files account approximately for 385 Kb and can be compressed to between 92 Kb (max) and 110 Kb (fast), using the Java compression package (java.util.jar) when reading and writing information from and to files.
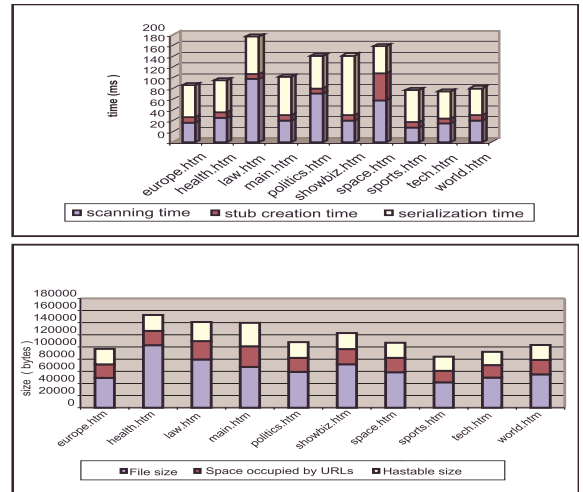

Figure 4: *Results for top group.*

7. RELATED WORK

The task of finding broken links can be automated using several applications[5, 9, 18]. However, these applications do not enforce referential integrity because, while useful detecting local and remote broken links, they cannot prevent them from occurring or reclaim wasted storage. Enforcing referential integrity on the web has been a subject of active study for several years now[4]. There are a few systems that try to correct the broken-link problem and, thus, enforce referential integrity, preserving web content availability. There are a number of techniques that make use of replication in order to provide greater availability, increase performance, minimize connections time, bandwidth and allow geographically oriented mirroring and off-line browsing. Most of these, however, do not deal with the issue of referential integrity.

LOCKSS[13, 15] is an open-source system that makes use of replication, namely spreading, in order to preserve web content. There are some fundamental differences to our work:
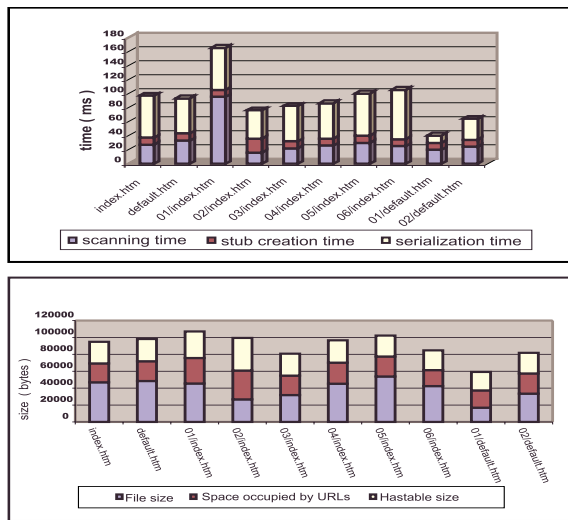
Figure 5: *Results for branch group.*

much work has been devoted in LOCKSS to ensure replica consistency, namely using hashing for each document. Storage reclamation is not addressed in LOCKSS since all documents in the system are considered important enough to be preserved forever. In LOCKSS, the system tries to preserve everything consistent. Our system tries to prevent memory leaks while preserving referential integrity and allowing replica discrepancy when needed.

Author-Oriented Link Management[3] is a system that tries to determine which pages point to a certain one. It describes an informal algebra for representing changes applied to pages, like migration, renaming, deletion, etc. It does not handle replication and it relies on the usage of custom-made, or customized authoring tools, i.e., referential integrity is not transparently provided to the user or developer. It does not try to reclaim storage space occupied by useless, i.e., unreachable documents.

Hyper-G[1, 8] is a "second-generation" hypermedia system that aims to correct web deficiencies and provide a rich set of new services and features. W.r.t. referential integrity, it is enforced using of a propagation algorithm that is described as scalable. In Hyper-G, there are no replicas of documents, just temporary cached copies. In our work, there is no need for the mentioned propagation algorithm since replicas are allowed to diverge.

The W3Objects[6] approach is also based on the application of a distributed garbage collector to the world wide web. The main difference is that it only handles migration and does not handle replication. Thus, existing solutions to referential integrity either do not aim at recycling unreachable documents or are not correct w.r.t. replication, or they are not integrated with the standard web.

8. CONCLUSIONS AND FUTURE WORK

In this paper we presented a new way of enforcing referential integrity in the world wide web. It deals correctly with the replication of web content for better performance and allows incremental, on-demand, replication of web content. It does not interfere with any protocol that maintains the replicas coherent among the participating sites. The system does not require replicas to be coherent. It limits storage waste, memory leaks, deleting any resources no longer reachable and does not require the use of any specific authoring tools. It integrates well with the web since it does not impose its model to the whole web. It does not impose causal delivery to be supported by the underlying communications layer, a fundamental aspect to ensure the system scalability.

Concerning future research directions, we intend to address further the fault-tolerance of our system, i.e., which design decisions must be taken so that it can remain safe, live and complete in case of process crashes and permanent communication failures, as well as address the collection of distributed cycles in replicated memory.

9. REFERENCES

[1] K. Andrews, F. Kappe, and H. Maurer. The Hyper-G network information systems. *J.UCS*, 1(4), April 1995.

[2] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, Dec. 1960.

[3] M. L. Creech. Author-oriented link management. In *5th Int'l WWW Conference*, Paris, France, May 1996.

[4] S. L. et al. Persistence of Web references in scientific research. *IEEE Computer*, vol 3(2), pp26-31, Feb. 2001.

[5] HostPulse. Broken-link checker, www.hostpulse.com.

[6] D. Ingham, S. Caughey, and M. Little. Fixing the "Broken-Link" problem: the W3Objects approach. *Computer Networks and ISDN Systems*, 28(7–11), 1996.

[7] D. B. Ingham, M. C. Little, S. J. Caughey, and S. K. Shrivastava. W3Objects: Bringing object-oriented technology to the Web. *World-Wide Web Journal*, 1, 1995.

[8] F. Kappe. A Scalable Architecture for Maintaining Referential Integrity in Distributed Information Systems. *J.UCS*, 1(2):84–104, Feb. 1995.

[9] LinkAlarm. Linkalarm, http://www.linkalarm.com/.

[10] J. L. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, Apr. 1960.

[11] L. Moreau and N. Gray. A community of agents maintaining link integrity in the world wide Web. In *Proc. of the 3rd Int'l Conference on the Practical Applications of Agents and Multi-Agent Systems (PAAM-98)*, 1998.

[12] J. M. Piquer. Indirect reference counting: A distributed garbage collection algorithm. In Aarts et al., editors, *PARLE'91 Parallel Architectures and Languages Europe*, volume 505 of *LNCS*. Springer-Verlag, June 1991.

[13] V. Reich and D. Rosenthal. Lockss: A permanent Web publishing and access system. *D-Lib Magazine*, 7, 2001.

[14] N. Richer and M. Shapiro. The memory behavior of the WWW, or the WWW considered as a persistent store. In *POS 2000*, pages 161–176, 2000.

[15] D. Rosenthal and V. Reich. Permanent Web publishing. In *Freenix Track, Usenix Annual Technical Conference, Usenix*, Berkeley, California, June 2000.

[16] A. Sanchez, L. Veiga, and P. Ferreira. Distributed garbage collection for wide area replicated memory. In *Proc. of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, San Antonio (USA), Jan. 2001.

[17] SyberSystems. Cvsviaftp: Automagic Web site mirroring via ftp. http://www.siber.org/cvs-via-ftp/.

[18] Xenu's. Linksleuth http://home.snafu.de/tilman/.