The Resource Locator Service: Fixing a Flaw in the Web

M.P.Evans and S.M.Furnell

Network Research Group
Department of Communication and Electronic Engineering, University of Plymouth,
Plymouth, UK.

Abstract

The architecture of the World Wide Web has scaled beyond its original expectations, but problems are now emerging that could undermine its effectiveness as an information system, and restrict its future growth. Nearly 30% of all web pages experience link rot, DNS domain names are rapidly running out, and older web pages are deleted without being archived, leading to the loss of potentially important information. These problems are caused by the URL and its reliance on the Internet's DNS for its namespace, which we argue represent a serious flaw in the web's architecture. In this paper, we present a new web-specific name resolution service that has been designed to address these problems. Called the Resource Locator Service, it offers an unconstrained namespace, and a mechanism for transparent resource migration that can dynamically locate static resources across time and space.

Keywords: Referential integrity; Resource migration; Link rot; Temporal references; Web namespace

1. Introduction

The World Wide Web (web) was designed by Tim Berners-Lee as a social creation rather than a technical one [14]. The ease with which its users can publish information as well as read it, combined with its exponential growth, has made it a social platform from which ideas and concepts emerge at an ever-increasing rate. However, the sheer volume of users and information has applied enormous pressure on its architectural foundations, which was not foreseen during its development. As its size continues to grow exponentially, increasing pressure is placed on its architecture, such that any flaw will become a major weakness in the system. With the web's role in society becoming increasingly important, and with the development of new access devices such as Personal Digital Assistants increasing the number of users, it seems appropriate to address any flaws before they disrupt the system.

In its present design, the most serious flaw in the web's architecture currently stems from the design of the Uniform Resource Locator (URL), which is used to reference a web resource. The URL has proven to be an unfortunate means of referencing a resource on the web, and its technical limitations are well documented [8, 14, 18, 27, 30, 35-36, 39]. It sits uneasily between the machine world of the web's architecture and the human world of the User Interface: the machines need the URL to be syntactically consistent and constrained to tell them *where* a resource is, whereas humans need it to be intelligible and memorable to tell them *what* the resource is. For the purposes of this paper, we have focused on three key problems that are inherent within the URL's design, which together could threaten the web's development if left unchecked:

1. *Link Rot* – the URL incorporates a server's hostname in order to provide a name for a resource. When the resource migrates to a different host, it must use a new URL that incorporates the new hostname. This causes all hyperlinks that use the old URL to break, or 'rot'. Currently, 28.5% of web pages suffer from link rot, with 5.7% of all links broken [38] and an average of 5.3% of links in search engines also broken [21]. The informational content and overall usefulness of the web will decrease as links become less reliable.

2. *Shrinking Namespace* – a URL not only defines a resource's location, it is also used as its name. Any company that wants to be remembered needs a memorable name, and the trend on the web has been to name a company after a memorable hostname to create a memorable URL. However, the URL is based on the aging Domain Name System (DNS) of the Internet, and the namespace this provides is running out. In March 2000 alone, new hostnames were being requested at a rate of nearly one a second, and some 14,322,950 distinct hostnames have been registered just for web server use [29]. The problem is exacerbated by copyright and trademark issues regarding the ownership of certain URLs, and the centralised nature of the DNS, making the URL in its role as a resource's identifier, the "web's achilles heel" [3].

3. *Lost History* – the web is designed for society, but crucially it neglects one key area: its history. Information on the web is *today's* information. Yesterday's information is deleted or overwritten, with no consistent means of searching through archived material other than manual navigation through a web site in the hope of finding archived material. The URL is a spatial identifier only, unconcerned with the temporal ordering of the web's resources, and so prevents the consistent retrieval of archived information [22].

In this paper, we examine existing solutions to these problems and highlight their weaknesses when confronted with a system the size of the web. We argue that the architecture of the web itself is flawed, and that solutions built on top of a flawed architecture cannot work. As such, we present a new approach with the design of the Resource Locator Service (RLS), which effectively addresses these problems by replacing the DNS with a name service designed specifically for the web.

The remainder of the paper is presented as follows: Section 2 discusses the background to the problem, and related work that has tried to provide a solution. Section 3 provides an overview of the RLS, while section 4 presents the design in much greater depth. Finally, sections 5 and 6 discuss the RLS in operation, while sections 7 and 8 present the results of performance measurements that have been taken from a prototype, which has been developed to demonstrate the effectiveness of the design, and discuss issues and further work that remains to be done.

2. Background and Related Work

2.1. Solutions to Link Rot

The use of the URL as a means of identifying a resource has caused all links to be inherently brittle, as once a resource moves to a new location, the link breaks. Designing a system for the web whose identifier does not change when the resource migrates (i.e. a location-independent identifier) will help to prevent link-rot. Such systems exist on the web and can be classified as using one of five approaches:

1. *The Chain Approach*
   A forward reference is left behind on the machine that the resource has migrated from, pointing to the new location. Although arguably optimal in terms of network traffic overhead [14], this approach can lead to forward references outnumbering

resources. Various shortcut operations can limit the length of the chain of forward references, but this approach is still inherently brittle, as locating your resource is dependent upon the state of someone else's server. Also a resource can only migrate onto a server that supports this approach. Examples include W3Objects [14].

2. *The Callback Approach*
   A database of all the links on the web is maintained. Each time a resource migrates, the database is updated and calls back all documents that contain a link to the resource, enabling each document to update its links. This approach guarantees referential integrity, as it is modelled on database technology. However, the web is not a database, and any server on the web may be down at any time. As the database must store all updates to servers that are down, it would eventually be overwhelmed by the number of pending updates [6]. This approach also requires the documents to be intelligent enough to remove links identified as broken, and so is not backwards compatible with the web's existing architecture. Examples include the HyperG system [18] and Atlas [30].

3. *The Search Approach*
   Whenever a resource needs to be located, a network-wide search is performed, with a flooding algorithm used to guarantee that all servers are queried. Although reliable, such an approach produces too much network traffic overhead for use on the web, and is the least optimal of all the approaches [14]. No examples currently exist for web-wide resource location through search.

4. *The Name Server Approach*
   A set of distributed name servers are used that manage a resource-identifier/location mapping. The name server is queried using the identifier of the resource in much the same way that a machine's IP address is determined through its hostname using the DNS [25, 26]. However, a name server system is essentially a distributed database, whereas the web is a federated system, and so locating the correct name server without breaking the web's existing architecture presents a significant challenge. Examples include the Handle System [39], and the Resolver Discovery Service (RDS) [36], both of which break the web's existing architecture.

5. *The Lecturing the User Approach*
   Not a technical approach, more a philosophical one. Berners-Lee and others have argued that a URL need not break if considered thought is given to its design [2]. However, despite the numerous technical arguments against this viewpoint, it is people who create URLs and people who are notoriously bad at consistent regular maintenance. Ultimately, as broken links on the W3C web site itself testify (e.g. the link http://discuss.w3.org/mhonarc/w3c-tech/threads.html on the document located at http://www.w3.org/MobileCode/Workshop9507/ is broken), lecturing the user will be ineffective at best.

2.1.1 Semantic Ambiguity

Although each of the five approaches provides its own solution to the problem of link rot, the semantics of the link and what it references are left in an ambiguous state. Referential integrity can ensure that links always reference the same resource, but what happens if the content contained within the resource changes? Should the semantics of the link require the content to persist for the lifetime of the resource, thus requiring a new resource and identifier to be created each time the content changes; or should the semantics be defined such that new content simply overwrites existing content? The former option will preserve all content, but will lead to an explosion of new resources, each with its own distinct identifier. Web sites that contain frequently changing content, such as daily news sites, will generate many new

resources, making linking to the site virtually impossible. Conversely, the latter option controls the number of resources but destroys information. Links will only be able to reference the web *site*, rather than specific information on the site, requiring the user to search manually for the story within the site's archives (if they exist). Although this problem of semantic ambiguity exists in the web's current architecture, the design of a new name service is a suitable opportunity for the ambiguity to be resolved.

2.2. Solutions to the Shrinking Namespace

Since the birth of the web, the number of domain names registered has exploded exponentially, leading to the number of memorable names shrinking rapidly. Companies that register domain names without actually using them in order to resell them for a profit exacerbate the problem by driving up the price of the remaining names. Furthermore, name disputes are becoming increasingly common, as the rights to the remaining names are fought over by companies and organizations with similar trading names. The Internet Corporation for Assigned Names and Numbers (ICANN) is the organization responsible for assigning Internet names, and acts as the central registrar for domain names on the web. ICANN has a defined policy for resolving domain name conflicts, called the 'Uniform Domain-Name Dispute Resolution Policy' [16], which attempts to resolve the issue of two parties fighting over the same name. However, before the policy can be invoked, one party must first file a complaint in a court of law. This is not an elegant solution, and with the number of available domain names dwindling, the problem can only get worse.

To resolve the problem, ICANN is currently examining ways to extend the top level domain name space. A Top Level Domain name (TLD) is the last part of a domain name (e.g. .com, .org, etc.). ICANN has recently extended the original list of seven (excluding country codes) to include new names such as *.biz*, *.coop*, and *.aero*, etc [13]. However, this must be seen as a short-term solution, as it simply constricts the same problem to vertical commercial and organizational domains. Equally, there is no guarantee that the new top level domains will be used, as companies are currently fighting to use the *.com* TLD over all other alternatives. Currently, 82.8% of all registered domain names use *.com* [20], largely because it is perceived as being associated with the web in people's minds far more strongly than any other TLD [37]. As such, the competition for *.com* names will still remain, regardless of how many new TLDs are introduced.

An alternative proprietary solution can be found in the RealNames system [31], which provides an alternative namespace to that of the DNS, and is used by various web portals including AltaVista, MSN, Google and LookSmart. RealNames uses *Internet Keywords* as 'human friendly identifiers' [23] that are registered by a company or organization usually associated with that name (e.g. 'Ford'). The RealNames system maps the human friendly identifier onto a company's web site, enabling a user to navigate to the site using the brand name of the company. In addition, it allows more than one party to register a web site under the same Internet Keyword, presenting several links to the user (either through an affiliated web site, such as AltaVista, or an affiliated browser, such as Microsoft's Internet Explorer 5) when a shared name is entered. In effect, it can be seen as occupying the middle ground between a naming scheme and a search engine, indexing Internet Keywords rather than every word in a web document, and so makes search results more reliable than a general web search. However, it is not a true architectural solution, as the same identifier does not uniquely identify a specific resource, leaving it to the user to manually select the most appropriate resource from a list. As such, an Internet Keyword cannot be used by the web as a machine-readable identifier, and so cannot replace the URL. Furthermore, it does not open up the namespace, as it is a proprietary solution that enables the RealNames company to determine what is and what is not a suitable Internet Keyword. The company is also in a position to limit the number of times that an Internet Keyword can be used by users to just 1000 times a year, at a cost of $100 per Internet Keyword per year [32].

The official solution is to use Uniform Resource Names (URNs) rather than URLs. URNs are designed to be permanent identifiers that identify a resource through a location-

independent name, thus simultaneously removing the dependence on the DNS and providing a solution to link rot. However, URNs have been on the agenda since 1992, and despite many short-term solutions [7, 27], no architectural method of providing URN to URL resolution has been developed. Worse, URNs are designed as machine-readable identifiers only [36], and so ignore the shrinking namespace problem completely as they are not designed for human use.

2.3. Solutions for Archiving the Web

Because the URL is a spatial locator and has no means of referencing a resource according to its time of creation, the web is always stuck in the present. A user can manually locate an archived version of a resource using the textual cues contained within a web page, but a web crawler cannot, as it does not understand the text. Existing solutions to this problem are again proprietary and unfocused in their approach. For example, the Internet Archive project [17] was begun in April 1996 by Brewster Kahle to literally archive the entire Internet. However, access to the archive is free only to researchers, students and not-for-profit organizations, and only if a research proposal indicating the need for access is first submitted and approved. Like the RealNames system, this is hardly in line with the open environment of Berners-Lee's original vision of the web.

Other systems have been designed in an ad hoc fashion in order to archive a particular library's digital contents, but no the other resources on the web [12, 28]. Although fully operational in themselves, these systems are isolated from one another and from the public at large because they are not part of the web's architecture.

2.4. The Need for a New Approach

The existing solutions described here are all isolated, independent approaches that do not address the cause of the web's architectural flaw and do not sufficiently integrate with the web's existing architecture. Consequently, they cannot provide effective long-term solutions to the flaw.

We argue that it is the use of the URL and its reliance on the DNS that is the root of the problems identified here, because:

- the DNS is designed to map a hostname onto an IP address, whereas the web needs a system to map a resource name onto a location;
- the DNS deliberately constrains its namespace as it only has to deal with the names of servers, whereas the web needs an unconstrained namespace to cater for all different types of resources and the needs of their owners;
- neither the DNS nor the URL have any way of storing and referencing a resource's time of creation.

In order to fix the flaw the web needs a new service tailored to its own needs, which can provide referential integrity, an unconstrained namespace, and can locate a resource according to its position in time as well as space. In the next section, we present the design principles for such a service, which we have termed the *Resource Locator Service.*

3. The Resource Locator Service

3.1 Overview

The Resource Locator Service (RLS) has been designed as a name resolution service that is specific to the web. The service is fully backwards-compatible with the web's existing architecture, but provides new functionality that enhances it. It has been designed to work with the DNS as well as on its own, so that it does not have to completely replace the DNS in order for it to function effectively. As such, the RLS will only manage those resources that

are explicitly registered with it, giving the user the choice of whether they wish to use its advanced features or not, while maintaining full backwards-compatibility with the web's existing architecture. The service is designed to be deployed in an evolutionary way, becoming increasingly prevalent on the web until it eventually becomes the *de facto* name resolution service, leaving the DNS as the *Internet's* name resolution service.

3.2 Unconstrained Namespace

The RLS has been designed to accept any string as the name for a resource, allowing an infinite variety of naming schemes and namespaces to be used. As such, the RLS provides a technical solution to the constrained namespace problem, but does not define a way of avoiding namespace conflicts. However, this is a matter of policy rather than technology, and so will be left for future research.

We envisage the RLS being run along similar lines to the DNS, with individual nodes in the system being independently operated, but an organization such as ICANN managing the addition or removal of those nodes. However, unlike the DNS, no organization would have control over the namespace.

3.3 Transparent Resource Migration

The RLS helps to prevent link rot by providing a transparent resource migration service that maps a persistent name onto a dynamic location. Resources that wish to make use of the service must first register with it, after which they are free to migrate without breaking the links that reference them. Resources that do not register will still cause link rot should they migrate to a different location. As such, the RLS will not provide an immediate solution to link rot, but will act to retard its growth until all resources register with the service. Once this happens, link rot will only occur when a resource is no longer required and is deleted. However, we envisage third-party archiving services being employed to host such unwanted resources, with the RLS being used to maintain their persistent name. Although this does not guarantee the integrity of all links, we argue that it is the most appropriate level of integrity for the web, as persisting all resources forever would be impractical, and unwanted resources will not have many links referencing them anyway.

Resource migration enables the RLS to shift the responsibility for link management (and thus link rot) from the resource owner to an automated service. Because a registered resource's name persists across servers, the resource owner is not required to manage broken hyperlinks, as their integrity is guaranteed by the RLS. The owner must still inform the RLS of the new location, but we have automated the entire process through a new Resource Migration Protocol (RMP - see section 5.1), which remotely instructs the RLS to update the location of a migrated resource. To demonstrate this, our prototype includes a RMP client with a drag and drop interface for moving resources across servers using a style very similar to Microsoft's Windows Explorer (see section 7.1.3). This enables the resource owner to freely move registered resources across all web servers without any manual link management

3.4 Temporal References

The RLS preserves the web's history through the use of a new *temporal reference*, which enables a resource to be identified according to its time of creation as well as its name. In this way, different versions of the same resource can share the same name, while still being differentiated by the RLS according to the reference's temporal attribute. This will also enable resources to be searched for according to their time of creation as well as their subject matter, enabling temporal search engines to be developed that show the state of the web and its resources at different points in time.

Temporal references require the resource and its content to be bound tightly together and treated as an atomic unit. Should the content need to change, a new resource must be created to contain it. However, rather than requiring a new name, the new resource can use the

existing name, but with a different temporal attribute. In this way, different versions of the same resource can be uniquely identified according to their time of creation, without requiring different names. References without a temporal attribute are assumed to be *current references*, which always reference the most recent version of the resource. This enables the temporal reference to resolve the semantic ambiguity of the hyperlink, as the current reference can be used to identify a web *page* whose content changes frequently, while a fully defined temporal reference can be used to identify a specific *version* of the web page according to the time that it was created.

Note that although the RLS binds a resource to its name and content in a way that appears similar to the specification of the URN, its functionality is very different. For example, the namespace of the RLS is completely unrestricted, whereas that of the URN is rigidly constrained. In addition, a URN can persist *beyond* the lifetime of its associated resource, whereas the RLS will persist a resource's name only as long as the resource exists.

4. Architecture of the Resource Locator Service

4.1. The Locator Network

The RLS is a distributed database, deployed across a network of nodes called *Locators.* A Locator is analogous to a Resolver [36], but is able to locate a resource through time as well as space by storing a resource's name, current location, and time of creation. This *Locator Network* performs a similar job to the DNS, but with granularity at the resource level.

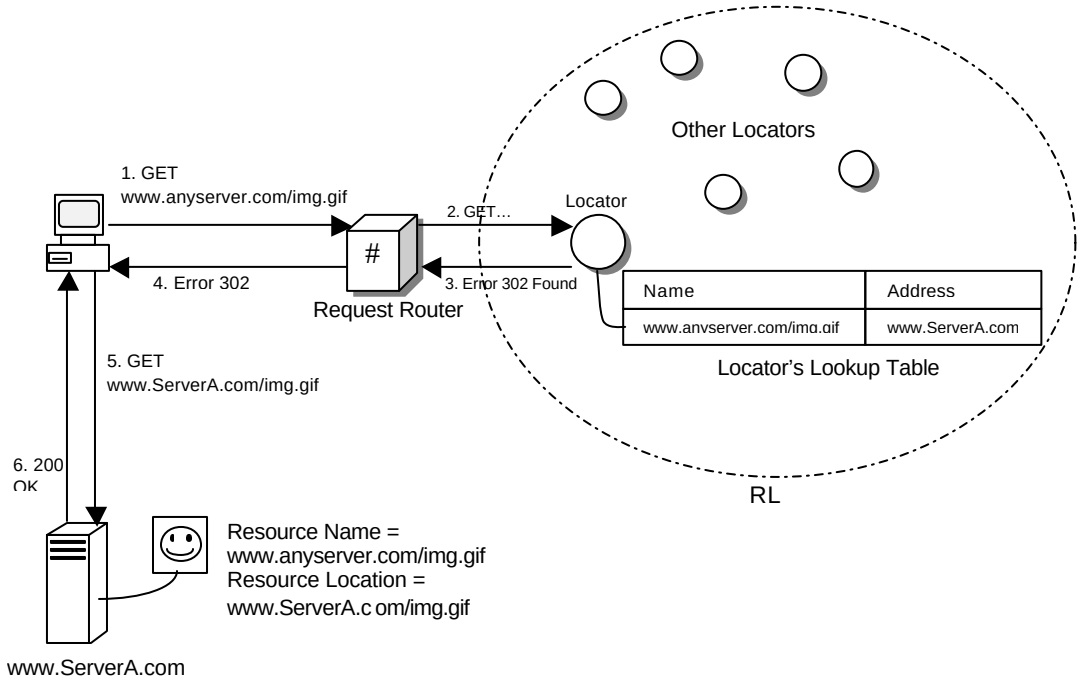

Figure 1 - Basic Architecture of the Resource Locator Service

Figure 1 shows a high level view of the RLS. The Locator network uses standard HTTP for communication, with the client being redirected to the correct location of the resource using the HTTP redirect mechanism [9]. Although this is not the most efficient approach, it facilitates backwards-compatibility, enabling all web entities to use the RLS.

The Locators take any string as the name of a resource and return that resource's location as a URL. The only constraint on the name used is that it must identify only one resource (although that resource may itself be replicated, and so have many values for its location in

the database). In this way, the RLS removes any technological barrier to future naming schemes, leaving policy alone to determine their structure.

4.1.1 *Migration Approach*

Of the five approaches described in section 2.1, the RLS uses the *name server* approach, as it is the most appropriate for the required functionality. Of the other approaches:

- the *lecture the user* approach advocates doing nothing as a solution, and so can be immediately discounted;
- *forward referencing* may provide referential integrity, but it still relies on the URL and thus the DNS's constrained namespace, and cannot support temporal references;
- the *search* approach will not scale to a system the size of the web [14], and the situation would be made worse if it had to manage temporal as well as spatial references;
- the *callback* approach provides referential integrity without providing a true naming service, and so the URL and the DNS would remain.

The name server approach, in contrast, is simply a distributed database, and so will scale to the size of the web and support temporal references. However, designing the system such that it interoperates with the web's existing architecture has been a major engineering challenge. Our solution to this problem is presented in the following sub-section.

4.2. Request Routing

Figure 1 is a high level representation of the RLS and depicts the client querying an appropriate Locator. However, in practice, a client must be made aware of exactly which Locator contains the required name/location mapping, but in a way that does not require the client or the hosting server to be altered. As such, some form of mediation is required between the client and the RLS that can transparently route the client's request to the appropriate Locator, without requiring any modifications in the client or server. This is difficult to achieve, however, as the constraints imposed on the RLS directly conflict with its distributed nature. For example, some distributed systems, such as the DNS or directory services, use the structure of the namespace itself to identify the correct node, but the RLS cannot, as the namespace must be left completely unconstrained. The alternative is a flat architectural configuration, with nodes arranged as peers and the namespace left unconstrained, but this requires the search approach to be used, which will not scale to a system the size of the web [14].

The IETF has also faced this problem with the design of the URN, and their proposal involves using the DNS to locate the correct node (termed a Resolver) in their Resolver Discovery Service [7]. A URN is sent to the DNS, which then forwards the URN onto the correct Resolver, which in turn resolves the URN to the correct URL. Essentially, the DNS's architecture is adapted so that it acts as an access system into a network of distributed name servers (the Resolvers), which perform the actual URN/URL mapping. Although this can work without requiring every browser and server to be changed, it has several disadvantages:

- It fundamentally changes the purpose of the DNS (from a name/address resolution service to an access service to another network).
- The DNS's importance in basic network routing prohibits its use in experimental work [3].
- URNs would still be constrained by the DNS namespace [7].

To resolve this problem for the RLS, we have developed a new system of mediation using an object called the *Request Router*, which transparently routes a standard HTTP request to the appropriate Locator in the RLS. The Request Router can work with any string as a resource identifier, and does not flood the entire RLS in order to locate the node with the required information. Furthermore, the system is fully backwards compatible with the web's existing architecture, and generic enough to provide mediation between the web and any distributed system.

4.2.2 The Request Router

The Request Router (RR) provides transparent, scalable mediation between the web and the RLS through the use of a *hash routing* algorithm. Specifically, a hash routing algorithm takes a string and maps it onto a hash space. The hash space is partitioned such that the string is mapped onto one and only one node in a distributed system [33, 40]. Using a hash routing algorithm as the basis for locating nodes in the RLS, therefore, enables any string to deterministically identify the Locator that contains the required name/location mapping. As the Locator is a database, it can be defined to store any type of information, and so the resource's location can be defined as any string. Thus, the hash routing algorithm solves the problem of how to use an unconstrained namespace for both a resource's name and location, whilst efficiently locating the correct Locator without flooding the system.

4.2.3 The Hash Routing Algorithm

The RR uses the same hash routing algorithm as the Cache Array Routing Protocol (CARP) [41], which maps a URL onto a specific cache in a distributed caching system, such that resources are distributed evenly across all caches in the system. The algorithm used in CARP works by mapping the URLs of resources that need to be cached onto a partitioned hash space, with each set in a partition being associated with one caching node [33]. The algorithm deterministically identifies the node as follows:

1. The URL of the resource is hashed.
2. The URLs of each of the caches in the array are also hashed in turn, with a weighting factor being applied that is set according to the physical characteristics of each node (see below).
3. The hash value of the resource and the hash values of the nodes are XORed together, producing a score for each resourceURL_hash/cacheNode_hash combination.
4. The cache node whose resourceURL_hash/cacheNode_hash combination scores highest is the one that hosts the resource.

Thus, given only the name of the resource and the names of all the machines in the array, the exact machine that holds the resource is uniquely and deterministically found. The resources are distributed uniformly across the system, but the weighting factor can be used to skew the distribution such that those nodes with a higher performance will host more of the resources.

4.2.3.1 Adapting the Hash Routing Algorithm for the Resource Locator Service

Although highly effective in large cache arrays, we have adapted the CARP protocol for use in the Request Router to better meet the needs of the RLS. Specifically, each node in a CARP system keeps a list of the URLs of all the caches in the system, and this causes a degree of network overhead. When applied to the RLS, however, every RR would need to know the URL of every Locator in the Locator Network, and periodically check for system configuration changes. This would create an unacceptable increase in network overhead, and would limit the types of device that could use the RR to those that could store and maintain the large lists of Locators that would be required.

The RLS avoids this limitation, however, by removing the weighting factor from the algorithm, and leaving the namespace of the resources open while restricting the namespace of the Locators. As such, a *URL pattern* is defined, which all Locators must use for their own name. The pattern encapsulates a number, which can be thought of as that Locator's identity number. Each number must be unique in the system, and all numbers must be sequentially ordered, starting from 0. An example URL pattern is:

http://www.nodeX.Locator.net/

where X represents the Locator's number in the system. For example, the first three nodes in the system (assuming a zero-indexing system) would have the following URLs:

http://www.node0.Locator.net/
http://www.node1.Locator.net/
http://www.node2.Locator.net/

Effectively, the URL pattern of the Locators acts as a *well-known* URL in a similar way to the well-known ports defined for TCP applications. Note that the URL pattern must be sequential, and there can be no gaps in the sequence. The URLs of a complete sequence of nodes, each of which has a URL that corresponds to the URL pattern, is therefore known as a *URL sequence*. In this way, the URLs of the Locators themselves become deterministic.

4.2.4 Updating the Request Router

Adding a new Locator to the RLS will cause the hash routing algorithm to implicitly re-map $1/n$ resources in an n-node system (note that n includes the new node) [33, 40]. As such, if a Request Router is unaware of this change in the system's configuration, then $1/n$ of its requests will go to the wrong Locator. However, the RR does not need to be synchronized with the configuration of the RLS, as the deterministic nature of the URL sequence enables it to detect any change automatically. Specifically, once the RR has the URL pattern for the RLS, it is a trivial matter for it to iterate along the resulting URL sequence, querying the existence of nodes at each point in the sequence. If a Locator fails to respond, then the RR has found the limit of the sequence. Thus, if a Locator cannot find a resource, the RR can simply query the existence of the Locators that have a node number that matches this limit (in case a Locator has been removed), or is one greater (in case a Locator has been added). If the limit remains unchanged, then the RR knows that the resource is unregistered with the RLS; otherwise, the RR simply rehashes the resource's name using the updated value, and sends the request to the newly calculated Locator. In this way, the RR is completely decoupled from the configuration of the Locator Network, and so any change in the configuration of the system does not result in a flood of update messages. Furthermore, the only information that the RR needs to store about the configuration of the system is the URL pattern and the number of nodes.

Note that the system's reliance on the URL sequence makes it vulnerable to node failure. Should a Locator fail, not only will its records not be available, but any RR that performs an automatic update during the failure will calculate the wrong number of nodes in the system, and will map most URLs onto the wrong Locator. However, the disruption can be limited if the RR continues to check for the existence of nodes beyond that at which no response is received, effectively enabling it to jump any holes in the URL sequence. Although the RR will still not be able to access the records in the failed Locator, it will at least know the correct configuration of the system, and so all other records will be available.

To further improve the resilience of the RLS, future work will look at introducing redundancy to the system, either by clustering several servers to provide a more fault-tolerant Locator design, or by using a *duplicated* hash routing algorithm, such as that proposed by [19]. Duplicated hash routing uses two hash routing functions and two cloned systems, one of which is a secondary system that acts as a backup in the event of a node in the primary system

failing. However, the benefits of this algorithm need to be determined, as although the reliability of the system is improved, the size and complexity are increased.

4.2.5 Integrating the Request Router into the Web

Because the RR is decoupled from the RLS, and needs only minimal information in order to function, it can be deployed virtually anywhere on the web. For example, it can be:

- *embedded into an HTML document as a Java applet, ActiveX control or even script.*
  When the user clicks on a hyperlink, the click event can be captured by the embedded RR, the hash routing operation performed, and the location of the Locator discovered. Thus, the node location process occurs within the HTML page itself. This ensures total transparency and maximum backwards compatibility, but permits only HTML documents to use the RLS;
- *built into a browser.*
  The browser automatically locates the appropriate Locator, allowing all servers to be unaware of the RLS, but requiring the client to be modified;
- *designed as a browser plug-in.*
  The browser is *extended* rather than redesigned, with the RR being downloaded by the user when required. This provides seamless evolution, and a solution that is more backwards-compatible than the previous example. Again, all servers are unaware of the RLS;
- *built into a server, or added as a server module.*
  The RR can be deployed on the server, which can perform the hash routing algorithm for each request it receives. This allows all browsers to be unaware of the RLS, and gives server owners the choice of whether to use the RLS or not;
- *embedded within a proxy server or a reverse proxy server.*
  The proxy server intercepts the request, and routes it to the appropriate Locator. This requires reconfiguration rather than redevelopment, allowing all browsers, servers and resources to be unaware of the RLS;
- *designed into a layer 4 switch or policy based router.*
  The switch or router contains the RR, transparently routing the request to the appropriate Locator without the knowledge of the client or the server. This provides total transparency and maximum backwards compatibility.

This allows the RR to be integrated into the web wherever it is required. In this way, the number of resources registered with the RLS can grow over time as more people decide they wish to use its services. As such, we envisage the adoption of the RLS to be evolutionary, rather than revolutionary, proceeding in a distributed way across different sectors of the web, as its services become useful to different types of user For example, to begin with, small numbers of web authors may embed a RR within a HTML document. After a short period, server owners may decide to embed a RR into their servers in order to use the RLS without affecting the clients. From this, plug-ins can be made available for existing browsers, allowing resources to be located directly in the browser, via both the DNS and the RLS. Once a reasonable number of people use the RLS, Internet Service Providers can embed a RR into their proxy servers. Eventually, the RLS will reach a critical mass of users, whereby a RR will become an integral part of a browser and server, and thus part of the web itself. In this way, the RLS's database becomes populated over time by resource owners who choose to register their resources with it. As such, the database does not need to be initialized, and because it freely co-exists with the DNS, does not prevent non-registered resources from being accessed.

5. The Functionality of the Resource Locator Service

5.1 Transparent Resource Migration

In order to help solve link rot, the RLS must be able to dynamically reassign the location of a resource while persisting its name. This can be done using a new protocol that we have developed specifically for this task, called the Resource Migration Protocol (RMP), which is briefly described in this section (see figure 2). A more detailed discussion will be the focus of a future publication.

RMP is based on WebDAV (Web Distributed Authoring and Versioning [11]), a new extension to HTTP designed to enable group authoring of web resources. WebDAV has been chosen because of its new file manipulation semantics, such as the ability to lock files, which are implemented via HTTP. However, RMP is flexible enough to allow existing HTTP servers that are not WebDAV compliant to participate in the migration process, but without the extra safeguards that WebDAV provides.

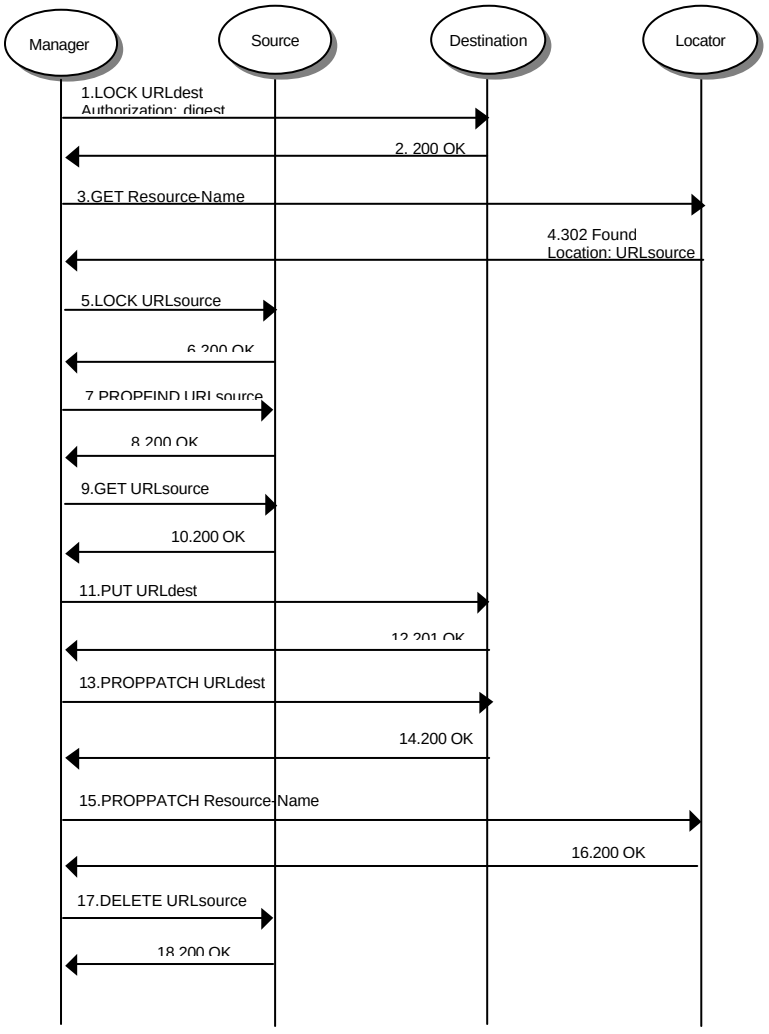

Figure 2 - The Resource Migration Protocol

5.1.1 The Resource Migration Protocol

The Migration Manager, defined as any entity that wishes to migrate a resource, manages the entire migration process, and is the only participant allowed to act as a client throughout, while the Source and Destination servers do not communicate with one another at all. The process begins with the Manager contacting the Destination in order to ascertain whether or not it is willing to host the resource. It does this by sending a WebDAV LOCK message (message 1) for the resource identified by URL_{dest}, together with authentication details (see figure 2). As the resource still exists on the Source server, there should be no resource physically located on the Destination that is bound to this location. Locking a null resource in this way has the effect of reserving the URL, ensuring that no other user can use URL_{dest} until the Manager unlocks the resource [11].

Upon a satisfactory response, the resource must be migrated from the Source to the Destination in such a way that it is accessible throughout the process. As such, the Manager must locate the Source using its own RR (messages 3 and 4), and LOCK the resource (messages 5 and 6), to ensure that it is not updated in the middle of the migration process. The Manager then sends the Source a WebDAV PROPFIND request (which allows a resource to be queried according to its attributes) in order to retrieve the resource's name, location and time of creation (messages 7 and 8), before sending a standard HTTP GET message to retrieve the resource itself (messages 9 and 10). The resource is copied to the Destination using a standard HTTP PUT message (messages 11 and 12), using URL_{dest} as the new location for the resource on the Destination. Note that although WebDAV provides MOVE and COPY methods for moving and copying a resource, they have not been defined for cross-server implementation; that is, a MOVE, for example, is only defined for moving a resource to a new location on the *same* server. This limitation prevents these methods from being used in the RMP, as WebDAV servers will not support moving a resource onto a different WebDAV server.

At this stage of the process, the resource is physically located on both the Source and the Destination. Before the resource on the Source is deleted, however, the Manager must update the appropriate Locator. It does this by sending a PROPPATCH message (message 15) to the Locator, which includes the resource's name and time of creation to identify the resource, and URL_{dest} as the property to be updated (i.e. its location). Once the Locator responds that the change has been successful, the resource located on the Source can be deleted using either a WebDAV DELETE message, or a standard HTTP DELETE request message. Once the DELETE response message (message 18) has been received, the migration process is complete.

5.2 Initializing and Updating the Resource Locator Service

The RLS is designed to be used in a way similar to that of the DNS; that is, a resource owner must first register the details of their resource in order to use the services of the RLS, and must then inform the RLS should the resource be deleted. The registration process acts to initialize the Locators' databases, and ensures that the resources that are managed are those whose owners explicitly requested the management service.

The Locators have been designed to enable a resource owner to automatically register and deregister a resource through the Locator's interface, using WebDAV messages. In order to add a resource's details to the RLS, a client can send a WebDAV PROPPATCH message [11] to the appropriate Locator, with the resource's name and time of creation properties encoded in the message body. The WebDAV specification defines PROPPATCH to set and remove properties, but the Locator cannot allow it to change the name of a resource or its time of creation (see section 3.2). As such, the Locator's interface restricts the client to adding or removing entire records only, with RMP used to automatically modify the location data. In this way, the complete functionality of the RLS is fully automated, while referential integrity is enforced.

In order to remove a resource's details from a Locator, the client must send it a PROPPATCH message containing a *remove* XML element (see [11], section 12.13.1), which acts to delete the details from the locator, but not the resource from the web. The Locator essentially delegates that responsibility to the resource owner, viewing the message as a request for the resource to leave the RLS, rather than for the resource to be destroyed. As such, the Locator will include the current location of the resource in a *location* header in the response message, allowing the client to send a HTTP DELETE message to the hosting server to physically delete the resource, if required.

Note that new HTTP headers could also achieve the same purpose as PROPPATCH, but the use of WebDAV messages is more consistent with RMP, and the semantics of the messages fit well with the needs of the Locator. Specifically, PROPPATCH "…processes instructions specified in the request body to set and/or remove properties defined on the resource identified by the Resource-URI" [11]. In this way, the Locator acts as a third party that manages the properties of the resource on behalf of its current host. In addition, the WebDAV error message *409 conflict* is used if a client tries to change the name of a resource within the Locator, as this message informs the client that it has "…provided a value whose semantics are not appropriate for this property" [11].

5.3 Temporal References

5.3.1 Defining the Temporal Reference

A client must be able to query a Locator for a resource according to its time of creation, and so some form of temporal identifier is required. As such, although the RLS can use any string as an identifier, we have specified two types of temporal reference in order that temporal referencing can be used immediately. Specifically, we have designed the Locators to work with standard URLs with a *timecreated* temporal component appended as a Query String (e.g. *http://www.aserver.com/index.htm?timecreated=Sun,%2006%20Nov%201994)*, which allows existing URLs to be used as temporal references; and with a new temporal URL scheme that we have defined as a more long term, architectural solution. The new temporal URL scheme conforms to the encoding rules defined in [4], and encapsulates the same semantics of the URL, but with the addition of a temporal component. Specifically, the new scheme, called TURL (Temporal Uniform Resource Locator), has been defined as:

turl://authority/path;time-created?query

The *authority* component of the TURL is identical to that of the URL (i.e. the domain name of the hosting server). The *path* component, too, is identical to the URL, but with one exception: a semi-colon separates the path that the server uses to locate the resource from the temporal information used to identify the time that the resource was created. The *query* component remains as it is defined for the URL, but the whitespace of the temporal component has been replaced with a dash (-) for clarity. Thus the URL:

http://www.aserver.com/index.htm?timecreated=Sun,%2006%20Nov%201994

can be re-written as a TURL as:

turl://www.aserver.com/index.htm;Sun,06-Nov-1994

In addition, as HTTP essentially forms the interface between the RR and the Locator, we have had to extend it in order to map the temporal component of the TURL onto a HTTP header. HTTP's existing headers already encode temporal information, but they are largely used for caching, and are normally sent by the server rather than the client. For example, the *Last-modified* entity header is used to represent the time at which the resource was last modified, which is another way of saying the time at which the resource was created.

However, it can only be used by servers in a response message, and cannot be used by a client at all. Equally, the *Age* entity header [9], which provides the estimated age of the resource on the origin server, is also a response header, only sent by a server (usually a caching proxy server). Alternatively, the *Date* header field is a general header, which can be used by both client and server, but only to represent the date and time at which the *message* was originated, not the resource [9]. Finally, the ETag header could encode the temporal information, as it provides a means of encoding user-defined values, but it, too, is a response header [9].

As such, faced with the decision of subtly altering the semantics of HTTP or defining a new general header, we have chosen the latter option, and defined a header called *time-created*, which can be used by both client and server, and which defines the time at which the resource was created. The value of the new header must be formatted according to [5], and it must map exactly onto the temporal component of the TURL. The new header provides the preferred means for querying a Locator according to a resource's time of creation, thus separating the temporal information from the resource's name. In this way, any appropriately specified namespace is able to become a temporal reference, enabling the RLS to retain its unconstrained namespace.

5.3.2 Defining the Scope of the Temporal Reference

A temporal reference supported by the RLS can enable one resource to persist across time, but not the resources behind any hyperlinks that might be embedded within it. For example, a HTML document registered with the RLS may contain several hyperlinks, but if the resources underlying the hyperlinks are not registered with the RLS, then they may not persist. As such, the RLS can only prevent link rot for those resources that it has been instructed to manage, and so web-wide link rot prevention can only be achieved if the RLS manages all web resources.

In addition, transient resources, such as dynamically created HTML documents, or streaming audio or video, are also not covered by the current design of the RLS and the temporal reference. This is because the TURL simply extends the existing URL protocol to encompass time, rather than adding any new functionality, and an existing URL references the object that creates a dynamic resource or a multimedia stream, rather than the transient resource itself. For example, a URL might identify an application behind a CGI (Common Gateway Interface) gateway, which in response returns a dynamically generated HTML document, but it does not identify the HTML document. Similarly, temporal references may enable the application to persist (although their definition does not cover persisting the application's state, merely its existence as a discrete file), but they do not cover its output, unless it is explicitly saved as a permanent web resource and given its own (temporal) URL.

6. Changing the Configuration of the Resource Locator Service

When the configuration of the RLS changes, the hash routing algorithm will re-map $1/n$ (where n = total number of Locators) of all records onto a different Locator. This will make the RR incorrectly route $1/n$ of all subsequent requests, unless the appropriate records are transparently migrated to the correct Locator. As such, the RLS must carefully manage *transparent record migration* (termed to reflect the fact that it is individual records in a Locator's database that must migrate) if it is to remain robust in the face of a changing configuration.

To manage this migration, we have developed the Locator Control Protocol (LCP), which allows all Locators in the RLS to be controlled such that the location of remapped records can be corrected without the knowledge of the RR. LCP is based on HTTP, ensuring its compatibility with existing web server technology. However, its full specification is beyond the scope of this paper, and so the following sections will present an overview of the protocol only. Section 7 will discuss the performance implications of the protocol.

The role of the LCP is to ensure that a Locator can be added to or removed from the RLS transparently, such that a RR is able to access all records throughout the system's

configuration change. The key to achieving this is to enable both configurations to co-exist for a short period by copying those records that must move, before the existing configuration is deleted to make way for the new one. In this way, all records are accessible whichever configuration the RR attempts to use.

6.1 Adding a New Locator

When a Locator is added to the system, a RR will only notice the change when it updates itself and the new Locator has adopted a domain name that complies with the appropriate URL pattern. In this way, the adoption (or removal) of a RLS-compliant domain name acts as a switch: with the domain name, a Locator is recognized by a RR as part of the RLS; without it, the Locator is not recognized, and so will simply be ignored. As such, by first copying all migrating records to their new locations *before* the new Locator adopts its new domain name (figure 3a), the LCP can enable both configurations to co-exist, ensuring that all records are accessible both before and after the new Locator is recognized by the RR.

The protocol requires the new Locator to act as the record migration manager, coordinating the migration process to ensure integrity of the records. While the migration is occurring, all Locators can still perform their standard name resolution service. Once the new Locator adopts a domain name that complies with the URL pattern, both configurations effectively co-exist. Those RRs that have not updated will be able to access the records in their existing location; those RRs that have updated, will be able to access the records at their new location (figure 3b). Once in this state, the old configuration can safely be deleted (figure 3c), causing those RRs that have not updated to receive an Error 404 when they try to access a remapped record, which will prompt them to update and thus to recognize the new configuration. In this way, no configuration updates need be sent to any RR throughout the entire process.

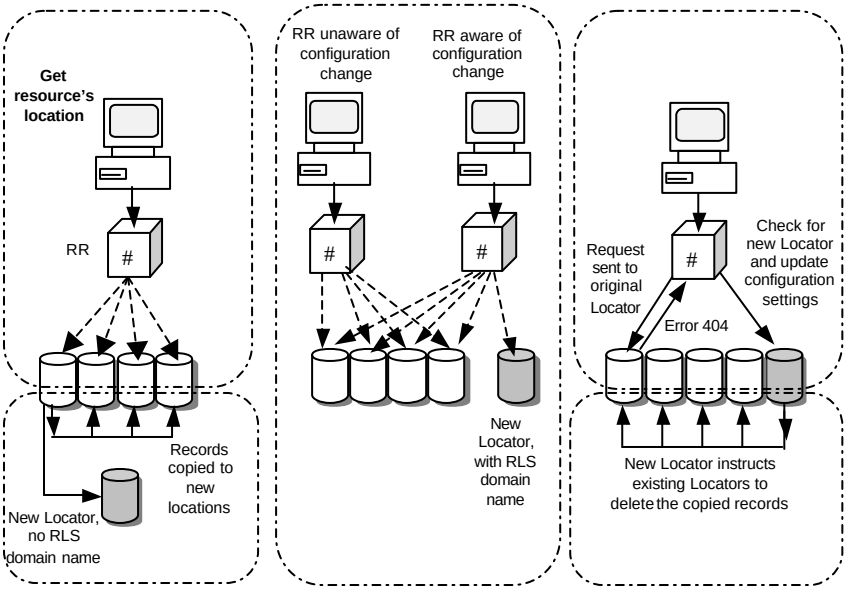

Figure 3a – migrating records are copied to new locations

Figure 3b – RRs in either state can access all records

Figure 3c – old configuration is deleted, forcing all RRs to recognize the new configuration

Figure 3a-c – Adding a New Locator

6.2 Removing an Existing Locator

Removing a Locator requires a different approach, however, as deleting its RLS-compliant domain name may leave a hole in the sequential numbering of the URL sequence, confusing the RRs. The process begins when the detaching Locator (*Locator*$_{detach}$ – coloured black in figure 4), acting as a coordinator, instructs all other Locators in the system that the configuration is about to change, thus causing them to copy the records that must migrate to their new locations. Note that some of the records will be copied onto Locator$_{detach}$, even though it is about to leave the system (figure 4a). Once this is complete, Locator$_{detach}$ remains in the RLS, and instructs the *last* Locator in the sequence (*Locator*$_{last}$ – coloured grey in figure 4) to detach itself from the system, even though Locator$_{last}$ is not the one that wishes to leave (figure 4b). In this way, the RLS shifts to the new configuration, with the existing configuration still operational (note that the RRs that have not updated may attempt to reach Locator$_{last}$, but will not receive a response; this will cause them to update automatically, however, thus moving them to the new state).

Once the new configuration has been reached, Locator$_{detach}$ will instruct the (now removed) Locator$_{last}$ to delete all of its records, before copying its own records over to make both Locators mirrors of one another. In addition, Locator$_{last}$ will also be given the same domain name as Locator$_{detach}$, making the two Locators identical clones (figure 4c). Once this happens, Locator$_{detach}$ is free to detach itself by removing its IP address from the DNS entry for its domain name, leaving the RLS in the new configuration. Again, the process of removing a Locator requires no synchronization messages from any Locator, as all RRs will automatically update themselves.

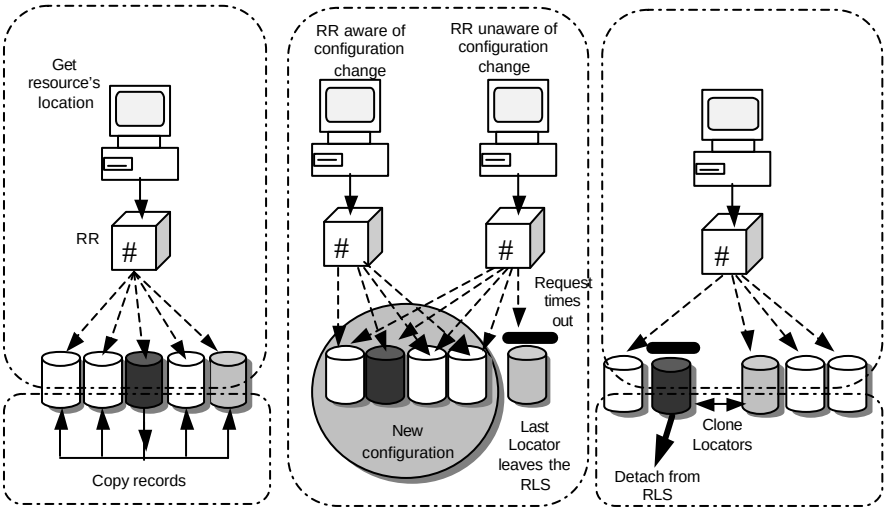

Figure 4a - Detaching Locator instructs other Locators to copy their records to the new locations

Figure 4b – Last Locator leaves the RLS first, forcing all RRs to update

Figure 4c - Last Locator and detaching Locator are cloned, before detaching Locator leaves the RLS

Figure 4a-c – Removing an Existing Locator

7. System Evaluation

To test the concepts discussed here, we have developed a prototype RLS, which comprises:

- a small network of Locators;
- a Request Router;
- a HTTP proxy server;
- a management interface.

This section presents the implementation details of the prototype, and performance data that we have gathered to demonstrate the scalability of the design.

7.1. Prototype Implementation

7.1.1 Prototype Locator

The Locator has been designed as a web server using Microsoft IIS on NT 4 Server, which uses Active Server Pages (ASP) to implement the functionality, and integrates with a Microsoft Access database that stores the name, location, and time of creation of each managed resource. The same resource name can reference multiple entries in the database, as each resource may have multiple locations (i.e. replicated resources) and multiple times of creation (i.e. when its content is changed). As such, the resource's name, time of creation, and location represent a compound key that together uniquely identify a single record, allowing the Locator to support replication and temporal referencing.

When a Locator receives a standard HTTP request, it looks up the resource's details in its database. If it contains the resource's name/location mapping, it returns a *302 Found* HTTP response message, with the current location of the resource contained within its *location* header; otherwise it returns a *404 Not Found* HTTP error message. In this way, a client can communicate with the RLS transparently, providing full backwards compatibility. If the Locator receives a HTTP request with a 'HEAD' method, it will simply return a HTTP *200 OK* response, enabling RRs to safely query for the existence of a Locator.

Finally, the Locator supports the RMP, using both WebDAV-enabled and standard HTTP servers as source and destination machines, enabling transparent resource migration to be implemented across all web servers.

7.1.2 Prototype Request Router

The Request Router is perhaps the most important part of the system, as it must be integrated into the web's existing architecture. To do this, we have created a Request Router object in C++ and embedded it into a simple HTTP proxy server, which routes the incoming request to the appropriate Locator, and then downloads the resource (if found) from the appropriate server. Any user who wishes to use the RLS can configure their browser to use the proxy server, enabling all legacy browsers and servers to use the RLS transparently. The proxy server application is also small enough for deployment on a client machine, allowing it and the browser to co-exist on the same machine if required. Future versions of the RR will include an ActiveX version, allowing the RR to be embedded into HTML pages, browsers such as Microsoft's Internet Explorer, or standard web servers.

The RR's client-side interface has two functions that are used to identify the correct Locator. The first, *RouteRequest()*, takes the name of a resource, and returns the appropriate Locator's URL with the resource's name appended onto it as a Query String (e.g. *http://www.node1.Locator.net/query?resourcename=http://www.aserver.com/aresource.htm*). This URL can then be sent directly to the appropriate Locator without the need for adding any new HTTP headers. The second, *GetLocatorByName()*, returns the appropriate Locator's URL *without* the resource's name being appended as a Query String. The resource's name must be encoded in a HTTP request header in a subsequent HTTP GET message. In both functions, the location of the resource is provided by the appropriate Locator via a HTTP redirect message (*302 Found*).

The RR also has functions that enable the URL pattern to be changed (thus allowing it to interface with other distributed systems on the web), and a function called *Update()*, which enables it to determine the number of Locators in a network by performing an automatic update.

7.1.3 A Prototype Management Interface and Resource Migration Protocol Client

In order to test the system, we have developed a management interface for controlling the Locators (see figure 5). The interface includes a RMP client, which enables it to act as the migration manager during a resource migration operation, and a Request Router object, enabling it to query the Locator Network.

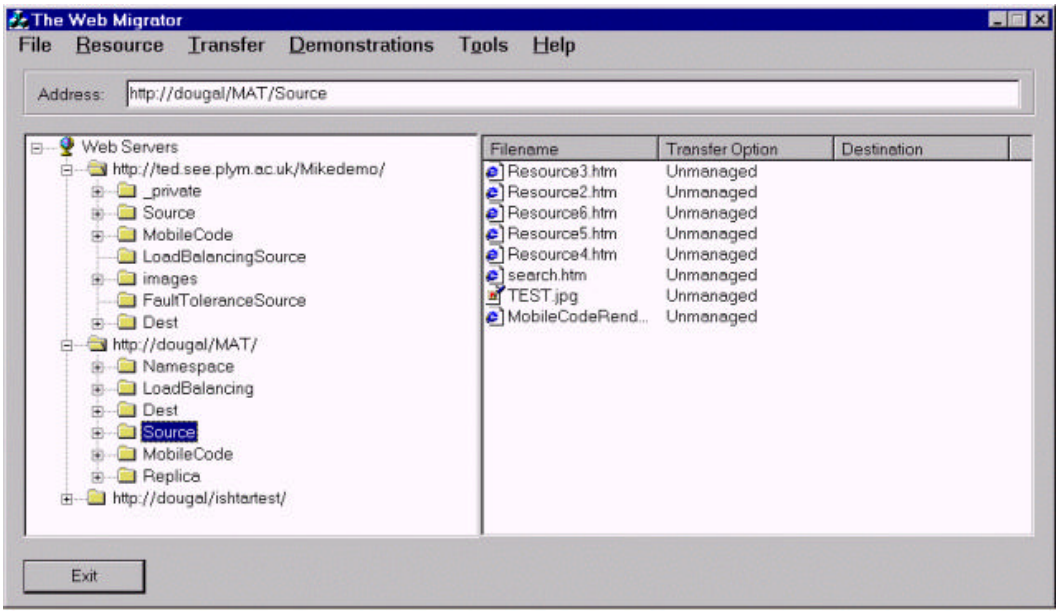

Figure 5 - Prototype Management Interface

The user interface is similar to Microsoft's Windows Explorer, allowing the user to drag and drop resources across web servers, while the underlying RMP functionality automatically updates the appropriate Locator's name and location details. The interface demonstrates the backwards-compatibility of the system, as the web servers shown in figure 5 have not been altered in any way, and are completely unaware of the RLS as a system.

7.1.4 Enhanced Web Services

In addition to the prototype RLS, we have developed several extra services that build on the services provided by RLS to extend the functionality of the web. These services take advantage of the RLS's resource migration mechanism, in order to provide transparent fault tolerance, load balancing, and mobile code functionality to existing web servers. Due to space restrictions, however, these services will not be discussed here, but will instead be presented in a future paper.

7.2 Scalability and Performance Issues

The prototype contains code that instruments performance, allowing us to measure its impact on standard web browsing. The results from our measurements, together with a general discussion on the scalability of the design, are presented in this section.

7.2.1 Network Overhead

Hash routing is a very fast algorithm for locating a node in a distributed system, providing a deterministic request resolution path through an array of machines, which results in locating a specific node in a single hop [24]. As such, the network overhead introduced by the RLS for both a successful and an unsuccessful resolution operation is *always* two additional HTTP messages (either a GET and an *Error 302 Found* response, or a GET and an *Error 404 Not Found* response).

If the RLS cannot find the resource, then a client application may contact the DNS if required, and if this is successful, the round-trip time to the RLS via the RR has been wasted. If, however, the RLS is completely integrated into the web, such that the DNS is not used to find the locations of resources, then all resources will be registered, and an *Error 404* means that the resource does not exist on the web, not just in the RLS. As such, there will be no added overhead, as the resource is unattainable.

Note that the RLS will not attempt to contact the DNS to find a resource, as it may not be appropriate in all cases. For example, a server hosting a RR may have registered all of its resources with the RLS, and so an *Error 404* means that the resource does not exist, not just that it is not registered. As such, it would serve no purpose for the RLS to contact the DNS in this situation, and so the RLS only manages its own *registered* resources, leaving clients to determine what to do with those that are unregistered.

7.2.2 CPU Overhead

The design of the RLS is such that the network overhead is constant, regardless of how many Locators are in the system, whereas the CPU overhead required by the RR scales linearly with respect to the number of Locators. As such, the scalability of the design is constrained more by CPU overhead than network overhead.

The linear scaling of the RR results from the hash function being used to distribute a set of records across many Locators, rather than to generate a unique value each time it is used, and so it does not have to worry about managing collisions, as the same result (i.e. the identified Locator) can be used many times for different resource names. The function distributes the records by hashing the URL of each Locator in the system, and as the time taken to hash each URL is virtually uniform (dependent solely upon the number of characters in the URLs that are hashed), the CPU overhead increases linearly with respect to the number of URLs (and thus Locators) it must hash.

We tested our Request Router on a Pentium Pro 200MHz with 64MB RAM, a Pentium III 400MHz with 128MB RAM, and an Athlon 1.1GHz with 128MB RAM. We set the number of nodes that the RR believed existed within the RLS to different levels, and measured the length of time that the RR took to identify the correct Locator. The results are presented in figure 6, which clearly reveals the linear relationship between time and the number of Locators. The results show that for small numbers of Locators, the time taken is insignificant, and that even with more Locators, the time taken is still small. As such, even with a relatively slow machine such as the Pentium Pro 200MHz, the RR can determine the correct Locator from a 10,000-node Locator Network in only 0.35 seconds.

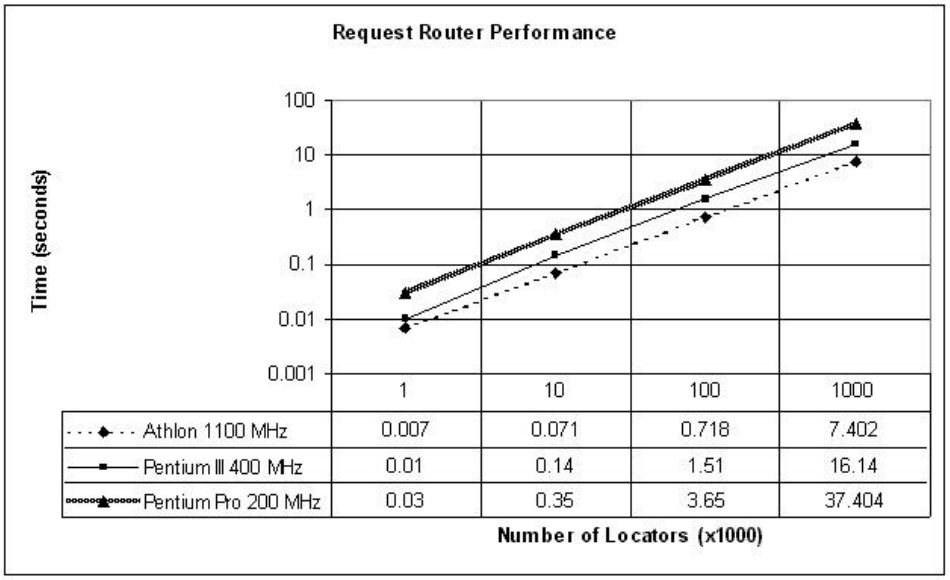

Figure 6 - Performance of the Request Router

In addition, the prototype RR was designed for experimental purposes, and is non-optimal. Specifically, it rehashes every Locator URL for every request that it routes, but unless the number of Locators changes, these hash values will remain static. As such, a more optimal design would cache the hash values in memory, and only rehash them when the configuration changes, thereby drastically reducing the length of time it would take to locate a Locator.

7.2.3 Total System Overhead

The total overhead introduced by the system was measured to provide a real-world indication of the system's performance. To do this, we first measured the time it took to visit the homepage of *www.lycos.co.uk* using a standard browser (Microsoft IE 5.0) and no RLS. The web site was visited 25 times, with the browser's cache deleted each time. We then connected the browser to our proxy server with the embedded RR, and visited the same sites again, once more taking 25 distinct measurements. The experiment was run using an Athlon 1.1 GHz PC with 128MB RAM, which acted as the proxy server with an embedded RR, and a Pentium Pro 200MHz PC with 64MB RAM, which encoded the functionality of the Locator. Both machines used Microsoft Windows NT 4 Server, and were connected via a 10Mbps Ethernet LAN.

The RR in the proxy server was manually configurable, allowing us to set the number of Locators according to requirements. For this experiment, we varied the number of Locators in the system from one to 1 million. However, to avoid having to physically deploy 1 million Locators, we reconfigured the proxy server so that it always forwarded the request onto the same Locator, regardless of which one the RR actually identified. The Locator would then return an *Error 404* message, which would cause the proxy server to redirect the request to the origin server, from where the resource would ultimately be retrieved. Because the overhead for the RLS is the same whether the resource is found or not (i.e. one extra HTTP request and one extra HTTP response), the measurements of the overall system overhead remained unaffected. In addition, this configuration removed any differences between servers that would have been introduced had each Locator been deployed on a separate physical machine.

The results presented in table 1 show the time taken to visit the Lycos web site both without the RLS, and with it, using one, 1,000, 10,000, 100,000, and 1 million Locators in the system. Each value is the 10% trimmed mean of 25 trials, with the overheard calculated by subtracting the mean from the value obtained without the RLS. The results show that the

overhead introduced by the RLS ranges from 0.869 seconds with only one Locator in the system, to 8.38 seconds with 1 million Locators. However, despite the large overhead for 1 million Locators, it remains small up to 100,000 Locators, with 1.582 seconds recorded.

Number of Locators	Download time for www.lycos.co.uk (time without RLS = 7.608 sec)	Overhead
1	8.477 seconds	**0.869 seconds**
1,000	8.483 seconds	**0.875 seconds**
10,000	8.546 seconds	**0.938 seconds**
100,000	9.190 seconds	**1.582 seconds**
1,000,000	15.985 seconds	**8.377 seconds**

Table 1 - Overhead introduced by the RLS with different configurations

The results show that the RLS introduces negligible overhead for a configuration of 10,000 Locators or less, and a relatively small overhead up to 100,000. However, it should be noted that neither the design of the RR or the proxy server are optimal, and that significant performance improvements can easily be made. We expect such improvements to enable the deployment of a 100,000 Locator system with negligible overhead.

7.2.4 Scalability of the Overall Design

In terms of growth, the hash routing algorithm can scale to over 4 billion (2^{32} = 4,294,967,296) Locators, performing single-hop resolution throughout [24]. Assuming each Locator can store the names and locations of 1 million resources (which, assuming an average URL of 50 characters, will require a database only 50 MB in size), today's web, with over 1 billion documents [15], would need the deployment of 1,000 Locators to fully manage all resources, which will take the RR on a Pentium Pro 200 machine just 0.03 seconds. However, even 10,000 Locators will only take 0.35 seconds, and this can accommodate ten times as many resources as currently exist on the web. Clearly, the configuration of the RLS can be better optimized, but this provides a good indication of the scalability and practicality of the design.

7.2.5 The Cost of Changing the Configuration

Changing the configuration of the RLS is not a time-critical process, as the name resolution service provided by the RLS is unaffected throughout the change. However, the change should still occur in a reasonable time-frame, and with a reasonable amount of network traffic, and so this section presents an estimate of the order of time that will be needed for a new Locator to be added to the system. Note that no estimations are provided for removing a Locator, as the operations are very similar, and so the time taken will be of a similar order.

The addition of a new Locator involves two steps that could significantly affect the time taken to update the system:

- determining which records need to move;
- physically moving the records.

The other steps involve data manipulation, such as deleting records, which will not negatively affect the scalability of the design or the time taken to change the configuration, and so will not be considered in the following calculations.

The first step involves every record in the system being processed by a RR whose node configuration is set at one node higher than its current value (i.e. set to $n + 1$). The time taken to do this can be significantly reduced if each Locator works in parallel with its peers, processing only the records contained in its own database. This is how the LCP operates. As such, ignoring the network overhead of the LCP, the time taken for step one will be:

$$\frac{Rt_r}{n+1}$$

where R is the total number of records in the system, and t_r is the time taken to process one record. Clearly, for the same R, the time will decrease with the number of Locators in the system, and as section 7.2.2 has shown, t_r is very small on even a low powered machine. Thus, the cost incurred by this first step is small. For example, suppose the existing configuration has 999 Locators managing 10^9 records, with an Athlon 1100MHz processor inside each Locator, giving t_r = 0.007 seconds (see figure 6). In this scenario, the time taken for step 1 will be:

$$\frac{10^9 \times 0.007}{999+1} = 7000 \text{ seconds (or 1 hour 57 minutes, 7 seconds).}$$

The cost incurred by the second step is dependent upon the number of Locators in the system and the number of records. When a new Locator is added to an n-node system, the records that are re-mapped will be evenly distributed across all Locators in the RLS [40]. If R is large compared to n, then every Locator in the system will contain records that are re-mapped. As such, each Locator will evenly distribute $1/(n+1)$ of its own records to the n other Locators in the RLS. This results in n Locators transmitting to n Locators (including the newly added one), resulting in the propagation of up to n^2 messages. This value represents the message limit, however, as each message can carry more than one record if required.

The time taken to transmit these messages can be shown to be acceptable for even the worst-case scenario, in which the configuration results in the maximum number of messages being sent (n^2), and only one message is in transit at any one time. For example, the configuration of step one is such that the addition of a new Locator requires the maximum number of messages to be sent for this configuration (i.e. n^2 = 1,000 x 1,000 = 10^6 messages). The time taken to send these messages can be estimated if the data transfer rate between Locators is known, as well as the number of bytes in each record. As such, if we assume that a conservative data transfer rate of 1.544 Mbps can be maintained between Locators, and the average record size is 150 bytes (50 bytes each for name, location, and time of creation), then the total time taken to transmit all messages (ignoring protocol overhead and converting bytes to bits) is:

$$\frac{1{,}000{,}000 \text{ x } 150 \text{ x } 8}{1{,}544{,}000} = 777.20 \text{ seconds (or 12 minutes 57 seconds).}$$

Thus, for this scenario, the *total* time taken to complete the addition of a new Locator is only 2 hours, 10 minutes, 4 seconds, which is entirely acceptable. Furthermore, this figure represents a maximum value, and can be decreased by sending messages in parallel, and by balancing the number of records per Locator with the number of Locators in the system, according to the available bandwidth and the performance requirements of the RRs. Finally, it is worth reiterating that a configuration change is not a time-dependent task, as the RLS is fully operational throughout the change, and it will not happen often, as the configuration of

the RLS should remain stable for relatively long periods of time. As such, the design of the RLS remains scalable and practical for a system the size of the web.

8. Discussion and Conclusions

Security is critical within the RLS, and is an area that needs further work before the RLS can be deployed. The RLS should use the HTTP digest authentication scheme [9], as it is stronger than the basic scheme, but both schemes have been determined too weak for the WebDAV working group, which faces similar problems, and is working on its own solution [10]. The RLS must use strong authentication techniques, as malicious use could potentially route requests to unwanted resources. As such, we anticipate future versions of the RLS to use the WebDAV Access Control Protocol [34] for its authentication requirements.

In addition, our prototype system has so far only been tested locally as a proof of concept. The next step is to build a bigger system and to stress test it under varying loads. However, its design is based on an existing, commercially proven distributed system (CARP), and so we expect it to stand up well to such a test. The LCP will also be redesigned to provide a more optimal solution that employs concurrent operation.

Overall, the Resource Locator Service provides an effective and elegant approach that addresses the problems of link rot, a shrinking namespace and a lost history, by providing a replacement for the URL and the DNS. We believe the RLS offers a better solution than existing systems, as it offers a single, coherent, architectural solution, which addresses all three problems at once. Further, the RLS is fully backwards-compatible with the web's existing architecture, yet extensible enough for it to be future proof. A web page creator, a browser developer, or a proxy server developer, can use the RLS today without affecting any other system within the web. Equally, the system should be scalable enough for it to become an integral part of the web's architecture in future years.

The resource migration aspect of the RLS will also enable it to become part of a nascent platform for distributed computing on the web, providing services such as fault tolerance and load balancing, and allowing new services to be deployed, including a temporal search engine, that are currently impossible to implement on a web-wide scale. We intend to develop these services to fully exploit the potential of the RLS in future work.

Acknowledgements

We would like to thank Paul Dowland for his help with the experiments that were conducted for this paper.

References

[1] Babich, A, Davis, J, Henderson, R, Lowry, D, Reddy, S and Reddy, S, DAV Searching and Locating, Internet Draft, http://www.webdav.org/dasl/protocol/draft-davis-dasl-protocol-00.html, April 20, 2000.

[2] Berners-Lee, T, Cool URIs Don't Change, 1998, http://www.w3.org/Provider/Style/URI.

[3] Berners-Lee, T and Fischetti, M, Weaving the Web – the Past, Present and Future of the World Wide Web by its Inventor, Orion Business Books, 1999, p. 133.

[4] Berners-Lee, T., R. Fielding and L. Masinter (1998), "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396, http://www.ietf.org/rfc/rfc2396.txt

[5] Braden, R, Requirements for Internet Hosts – Communication layers, STD 3, RFC 1123, October 1989.

[6] Briscoe, RJ, Distributed Objects on the Web, BT Technology Journal vol. 15 No 2, April 1997, pp.157-171.

[7] Daniel, R and Mealling, M, Resolution of Uniform Resource Identifiers using the Domain Name System, RFC 2168, June 1997.

[8] Evans, MP, Phippen, AD, Mueller, G, Furnell, SM, Sanders, PW and Reynolds, PL, Strategies for Content Migration on the World Wide Web, Internet Research, vol. 9, no. 1, 1999, pp25-34.
[9] Fielding, R, Irvine, UC, Gettys, J, Mogul, JC, Frystyk, H, Masinter, L, Leach, P, Berners-Lee, T, HyperText Transfer Protocol – HTTP/1.1, RFC 2616, June 1999.
[10] Franks, J, Hallam-Baker, P, Hostetler, J, Lawrence, S, Leach, P, Luotonen, A, Stewart, L, HTTP Authentication: Basic and Digest Access Authentication, RFC 2617, June 1999.
[11] Goland, Y, Whitehead, J, Faizi, A, Carter, S and Jensen, D, HTTP Extensions for Distributed Authoring – WebDAV, RFC 2518, February 1999, http://andrew2.andrew.cmu.edu/rfc/rfc2518.html.
[12] Hogarth, J, The National Register of Archives/ARCHON: a study to inform a development strategy for a National Name Authority File, for the Historical Manuscripts Commission, September 1999, http://www2.hmc.gov.uk/pubs/jhreport.htm.
[13] ICANN Announcement, November 16th 2000, http://www.icann.org/announcements/icann-pr16nov00.htm
[14] Ingham, D, Caughey, S and Little, M, Fixing the 'Broken-Link' Problem: The W3Objects Approach, in: The Fifth International World Wide Web Conference, Paris, France, May 6-10 1996.
[15] Inktomi and NEC Research institute, http://www.inktomi.com/webmap/.
[16] Internet Corporation for Assigned Names and Numbers, Uniform Domain-Name Dispute-Resolution Policy, http://www.icann.org/udrp/udrp/-policy-24oct99.htm, 24 October 1999.
[17] Kahle, B, Preserving the Internet, Scientific American, March 1997. See also the Internet Archive, www.archive.org.
[18] Kappe, F. A Scalable Architecture for Maintaining Referential Integrity in Distributed Information Systems, in *J.UCS* Vol. 1, No. 2, Springer, February 1995, pp. 84-104.
[19] Kawai, E, Osuga, K, Chinien, K and Yamaguchi, S, Duplicated Hash Routing: A Robust Algorithm for a Distributed WWW Cache System, in: IEICE Trans. Inf. & Syst., Vol.E83-D, No.5, May 2000.
[20] Kosters, M, Massive Scale Name Management: Lessons Learned From the .COM Namespace, in: The Workshop on Internet-scale Software Technologies, University of California, Irvine, California, USA, August 19-20, 1999, http://www.ics.uci.edu/IRUS/twist/twist99/.
[21] Lawrence, S and Lee Giles, C, Accessibility of Information on the Web, Nature, Vol.400, 8 July 1999, pp107-109.
[22] Lyman, P and Kahle, B, Archiving Digital Cultural Artifacts, Dlib Magazine, July/august 1998, http://www.dlib.org/dlib/july98/07lyman.html.
[23] Mealling, M, Requirements for Human Friendly Identifiers, Internet Draft, June 1998, http://www.ics.uci.edu/pub/ietf/uri/draft-mealling-human-friednly-identifier-req-00.txt.
[24] Microsoft Corporation, Cache Array Routing Protocol (CARP) and Microsoft Proxy Server 2.0, 1997, http://msdn.microsoft.com/library/backgrnd/html/carp.htm.
[25] Mockapetris, P, Domain names – concepts and facilities, RFC1034, November 1987, http://www.ietf.org/rfc/rfc1034.txt.
[26] Mockapetris, P, Domain names – implementation and specification, RFC1035, November 1987, http://www.ietf.org/rfc/rfc1035.txt.
[27] Moore, K, Location-Independent URLs or URNs Considered Harmful, Internet Draft, 7 January 1996, ftp://cs.utk.edu/pub/moore/draft-ietf-uri-urns-harmful-00.txt.
[28] Moore, R, Baru, C, Rajasekar, A, Ludescher, B, Marciano, R, Wan, M, Schroeder, W and Gupta, a, Collection-Based Persistent Digital Archives – Part 1, D-Lib Magazine, Volume 6 Number 3, http://www.dlib.org/dlib/march00/moore/03moore-pt1.html, March 2000.
[29] NetCraft WebServer Survey, http://www.netcraft.com/Survey/.

[30] Pitkow, JE and Jones, RK, Supporting the Web: a Distributed Hyperlink Database System, in: The Fifth International World Wide Web Conference, Paris, France, May 6-10 1996.
[31] Popp, N, Masinter, L, The RealName System: a Human Friendly Naming Scheme, Internet Draft, draft-popp-realname-hfn-00.txt. See also www.RealNames.com.
[32] RealNames.com, Internet Keyword Subscription Policy, January 2001, http://web.realnames.com/Virtual.asp?page=Eng_Policy_Subscribe_Agreement
[33] Ross, KW, Hash Routing for Collections of Shared Web Caches, IEEE Network, November/December (1997), pp. 37-44.
[34] Sedlar, E and Clemm, G, Access Control Extensions to WebDAV, Internet Draft, http://www.webdav.org/acl/protocol/draft-ietf-webdav-acl-01.htm, April 28 2000.
[35] Shafer, K, Weibel, S, Jul, E and Fausey, J, Introduction to Persistent Uniform Resource Locators, in: Proceedings of INET96, Montreal, Canada, 24-28 June 1996.
[36] Sollins, K, Architectural principles of Uniform Resource Name Resolution, RFC 2276, January 1998, ftp://ftp.isi.edu/in-notes/rfc2276.txt.
[37] Sullivan, D, Goodbye Domain Names, Hello Real Names, in: The Search Engine Report, May 2000, http://www.searchenginewatch.com/sereport/00/05-realnames.html.
[38] Sullivan, T, All Things Web, http://www.pantos.org/atw/35654.html.
[39] Sun, SX and Lannom, L, The Handle System: A Persistent Global Name Service – Overview and Syntax, Internet-draft, February 2000, http://www.ietf.org/internet-drafts/draft-sun-handle-system-04.txt.
[40] Thaler, D.G., and Ravishankar, C.V., "Using Name Based Mappings to Increase Hit Rates", IEEE/ACM Transactions on Networking, 6(1), Feb. 1998.
[41] Valloppillil, V and Ross, KW, Cache Array Routing Protocol v1.0, Internet Draft, draft-vinod-carp-v1-02.txt, http://www.cs-ipv6.lancs.ac.uk/ipv6/documents/standards/general-comms/internet-drafts/draft-vinod-carp-v1-03.txt, February 26 1998.