

Maintaining information resources

Sofus Macskassy* Leon Shklar
Computer Science Department, Rutgers University, New Brunswick, NJ 08903
Pencom Systems, 40 Fulton St., New-York, NY 10038

{sofmac,shklar}@cs.rutgers.edu

Abstract

With the proliferation of the World Wide Web, it has become very important to provide advanced tools for maintaining referential integrity of information resources. The growing tendency toward building increasingly complex Web sites makes it necessary to maintain not only physical files, but also logical resources, or views, which are composed of references to other resources and presentation programs. Our solution to this problem is to design an infrastructure of resource maintenance agents. It includes the Data Agent, which keeps track of files and supports third-party requests to notify them of changes that occur to these files. Another component of the infrastructure is the Repository Agent, which supports change notification requests for logical resources. Prototype implementation of the infrastructure is currently available and is discussed in this paper.

1 Introduction

The World Wide Web is rapidly becoming ubiquitous, and its rapid expansion comes increasingly in the form of logical information resources. Such resources are specified through references to other physical and logical resources, and it may be necessary to execute several layers of presentation programs to evaluate their content. Such a trend necessitates faster, better and more consistent ways of keeping track of both the resource specifications and the underlying data. In addition, the proliferation of full-text search engines has resulted in the wide-spread use of indices referencing resources that have either changed or been relocated since the index construction. New files and views are continually created, changed, moved and deleted, resulting both in obsolete references and in the decreasing efficiency of content-based search.

The problem of keeping references, resource specifications, and content-based indices current is only going to deteriorate, and there is practically no hope that

it may be addressed successfully by a centralized service. To make matters worse, it is impractical to require that a solution be accepted simultaneously by all information providers. The challenge is to come up with a design that would work with the participation of only a few information providers, and yet would supply a growing benefit for other providers choosing to adopt the design. In this paper, we describe the proposed infrastructure of resource maintenance agents. It includes the Data Agent (DA), which keeps track of files and supports third-party requests for notification of changes that occur to these files. Another component of the infrastructure is the Repository Agent (RA), which supports change notification requests for logical resources. It communicates with other agents to keep track of references to their resources. We have implemented a prototype Data Agent and a very basic Repository Agent as extensions of the *Jigsaw* [1] HTTP server from the World Wide Web Consortium¹ (W3C). The Repository Agent is the subject of our current work and is not the primary focus of this paper, which concentrates mainly on the DA part of the architecture.

2 Agent Architectures

In designing our solution, we evaluated and analyzed existing agent architectures as well as architectures that, to the best of our knowledge, have not yet been implemented. Every architecture comes with its own pros and cons, and, upon careful consideration, we have selected the one that is the most sound considering the scale of the Internet and limits of individual servers. A distinguishing feature of our design is that it is based on distinct distributed data and repository agents, while most of the alternative approaches utilize a main agent or process that resides on a single machine and does all the work.

*This research was supported in part by Bellcore, Morristown, NJ

¹W3C's homepage is located at <http://www.w3c.org>

2.1 Client- and Server-based Agents

In the client-based architecture, agents are closely associated with clients. Each client agent has to remotely check all files it wants to keep track of, which not only increases the net-traffic, but also puts additional load on the client. The checking would be performed by the client agent through remote *protocol servers* (http, gopher, ftp, etc.). In terms of the network load, the protocol servers would be much better off maintaining their own control over polling local files, instead of performing redundant one-time status checks in response to client requests. Another problem is the effect of multiple clients polling the same file, instead of the server sending out group notifications. The only obvious advantage of client-based designs is in freeing servers from necessity to install an agent.

Server-side agents are much more network-friendly. They poll local files and notify registered clients. There is still a potential for overload problems caused by reporting modifications to rapidly changing files. On-request notifications and limits on polling and notification frequencies, are some of the strategies that could help to remedy this problem. On the down side, protocol servers have to accept a DA, which may be unappealing to some server administrators.

2.2 Persistent URLs

The *Persistent URL* [6, 2] (PURL) mechanism was designed to address the problem of dangling links. It is based on designating a central server that supports URL registration and is notified about URL changes by their owners. To make sure that a registered link stays current, it has to point to the designated server, which performs the redirection. PURLs are an intermediate step toward the *Universal Resource Names* [4] (URNs). Both PURLs and URNs are intended to ensure that links do not move, go away or become outdated with respect to machine-names, relative file locations, etc. They do not provide any support for verifying the currency of URL content.

Our architecture would inter-operate with this approach, especially if the URL registration server could be persuaded to accept an agent. The DA would still function at the server and file system level and would not be affected. Instead of using the existing PURL mechanism, we could just as easily implement a superset of PURL functionality, with RAs keeping track of files the same way the central PURL server would. As with PURLs, the RAs would require that resource owners submit notifications but only when dealing with files that move across different machines. PURLs and URNs do not compete with our approach. When available, they would provide the added bonus of freeing the RAs from having to keep track of the network location of physical resources.

2.3 URL-minder: a Web-robot that keeps track of URLs

URL-minder [9] is a web-robot that accepts URL notification requests, polls these URLs, and sends out notifications when they change. If multiple users ask for the same URL, it is still only polled once. While it has much of the same functionality as our DA, it implements a client-based architecture, which has the drawback of needing to poll URLs all over the network. The web-robot does take care of some of the problems by supporting a single polling process per URL for multiple client requests. Unlike our approach, URL-minders do not support either complex notification conditions (e.g., modification by a certain user within a given time range) or flexible polling and notification policies (e.g., only poll once a week). These robots can not be used to keep track of logical resources. Even so, the URL-minder is an improvement over purely client-based agents.

2.4 Other Approaches

It remains to mention Ingham, Caughey, and Little's work on the broken link problem[5]. Their idea involves placing referential links along the relocation path of a physical resource (i.e., when the resource moves, it replaces itself with a link to its new location). Users are referred along the sequence of links to finally reach the current location. At that point, they may create a short-cut for future access.

Somewhat related is the work by Francis, et. al. describing *Ingrid* [3], which is based on a navigation tool. Here, the navigator goes through a *Forward Information Server* (FIServer) that caches the results of earlier queries. On posting a cached query, the FIServer returns links that it found during an earlier search. Since FIServers can communicate with each other, they form a distributed system that keeps track of information. When an information object changes, the change propagates through FIServers invalidating the affected parts of the cache and forcing re-evaluation of future queries.

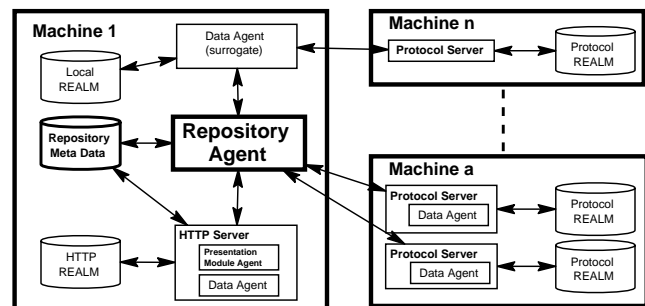


Figure 1: High-level architecture

3 Our Architecture

We propose a new agent architecture that is designed to maintain reference integrity and currency of content-based indices. As mentioned in the Introduction, the two main components of our architecture are the Data Agent and the Repository Agent. In sections 3.1 and 3.2, we provide a description of these components and their relationships (fig. 1). Section 3.3 serves to discuss requests and notifications. To further explain our approach, we provide a detailed description of the processing of a sample request (section 3.4).

3.1 Data and Repository Agents

The main charter of the DA is to keep track of files that are local to its associated protocol server. This agent knows only about file objects in its local file system and neither anything about what the files actually are (a text file? C file? binary file?), nor anything about their contents. The DA keeps track of file statistics known to the operating system and sends out notifications when changes affect the notification conditions. Ideally, every DA is associated with a *protocol server* (http, gopher, ftp, etc.), and the agent's realm is limited to that of the server. For example, an ftp-DA would not necessarily have access to a file retrievable through an http-daemon that has access to the same file system. Because current operating systems lack integrated notification mechanisms, the pragmatic solution is for DAs to periodically poll file-system objects to see if they have been changed, created, deleted, etc., and if so, to send notifications to the interested parties. Unfortunately, it may not be possible to install a DA at every protocol server. For this reason, we introduce so-called *surrogate* agents, discussed in section 3.2. In addition, it may be required to keep track of files that are local to the client machine but are not available through any usual protocol. Figure 1 serves as an illustration for these scenarios.

The RA helps to maintain repositories of resource specifications. It does so by sending notification requests to resource maintainers, which are referenced in the specifications. The resource maintainers may be either DAs or, in turn, other RAs, and the resources may be either local or remote. If the RA specifications reference only physical resources, as in figure 1, the agent sends notification requests exclusively to DAs, which ensure that both content and location of the files are up to date. In the general case, RAs are responsible for tracking changes to local resource specifications, because these specifications may be referenced by objects in other repositories and tracked by their agents. In the case of a data change, it is always first discovered by the DA and may then be propagated through a chain of RA notifications. It is up to every RA to figure out whether the change involves updating a content-based index, resource references or any other information. The RA

work was initially motivated by the need to maintain metadata repositories[7], and we expect the RAs to be closely integrated with repository servers.

3.2 Alternative Data Agents

The proposed approach would work best if every protocol server has an associated data agent, but for our architecture to make sense in the real world, we have to provide for the large legacy of existing servers. One possible alternative for a remote protocol server that does not have an associated DA is to accept mobile code. In this case, the first RA attempting to keep track of physical resources local to that server, would send over the DA code and then go on as if the DA was always there. As the last resort, we are proposing to use so-called *surrogate* agents that are local to clients. These agents perform network polling of remote physical resources, regardless of the protocol servers.

Surrogate agents are not very efficient but are necessary to support the transition to the new architecture; they take care of situations when server-side agents are not available. Surrogate agents are also used to keep track of files that are local to clients. In the worst-case scenario all work is done by the surrogate agents and the architecture degenerates into a client-based setup similar to the URL-minder. The important difference is in the obvious incentive for protocol servers to install their own DAs in order to decrease their network traffic. The advantages of our approach would increase continuously as more protocol servers install DAs, providing even higher incentives for the remaining protocol servers to join in.

3.3 Requests and Notifications

In order for our design to achieve the desired flexibility, it should be possible to specify a wide range of requests with flexible notification conditions. As mentioned earlier, information available to DAs is limited to file system statistics. Table 1 summarizes the supported notification conditions.

A request may contain any combination of these conditions, as well as notification frequency and other client-specific requirements. If all the client cares about is whether a file has been modified within the last twenty-four hours, there is no reason to poll that file every five minutes and hence bog down the server unnecessarily. To remedy the situation, we have provided for additional polling and notification conditions, which are shown in table 2.

A request is associated with a list of file system objects, be they files or directories, and may evolve over its lifetime, in accordance with the changing requirements. Along with conditions, notification requests also determine information to be included in every notification. For example, there may be no notification

Recursive Polling	If the object is a directory, is it necessary to recursively poll objects in all subdirectories?
Size Range	Send a notification if the object is inside/outside this range.
Does it exist	Send a notification if the object exists/does not exist.
Is it accessible	Send a notification if the object is/is not accessible.
Is it readable	Send a notification if the object is/is not readable.
Is it writable	Send a notification if the object is/is not writable.
Is it executable	Send a notification if the object is/is not executable.
Last modified	Send a notification if the object has been accessed by anybody within a fixed period of time (or since last notification).

Table 1: Notification conditions

Priority	The priority of this request. How soon after something happens to the object should a notification be sent out.
Time-Range	Whether the polling should only occur within a certain time-range.
Notify-Time	When is the client available to receive a notification.
Notify-Type	How to send a notification (e-mail, socket, on-request, etc.).

Table 2: Polling and notification conditions

conditions that depend on file size but it may have been requested to include file size information in every notification. Also, while most applications and agents are good at keeping track of submitted requests, it should still be possible for a client to find out which of its earlier requests are currently active at a DA.

As mentioned earlier, a way to avoid excessively frequent notifications is to support on-demand notifications. This may be the only solution for clients that don't accept incoming socket-connections and are not accessible through e-mail. On-demand notifications are much preferable to client-side polling because of greatly reduced network load – clients submit one polling request per DA, each of which may keep track of any number of physical resources. Table 3 shows the kinds of notification requests that support the described functionality.

Add-Request	Specify a new request using the above discussed notification contents and conditions.
Query-Request	Display contents and conditions of a request.
Update-Request	Update contents and conditions of an outstanding notification request.
Delete-Request	Delete a request.
List-Requests	List all requests made by the client.

Table 3: Types of client requests

3.4 Processing a Sample Request

Our prototype DA was implemented as an extension to the Jigsaw http server, hence we are using the http server in figure 2 to make it easier to explain the prototype. We assume that the client has enough information for building a notification request. The request processing steps may be summarized as follows:

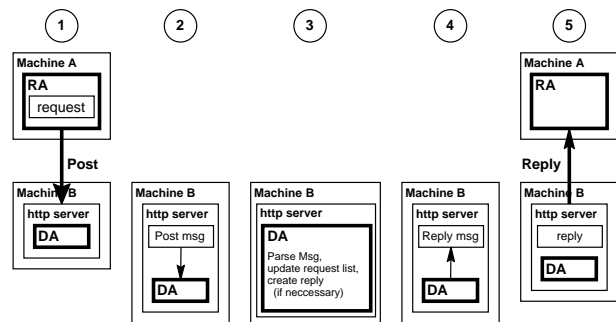


Figure 2: Processing a sample request

- 1 The RA generates a request and sends it to the DA (in this example, the DA is running on a remote machine). To get to the DA, the RA submits a POST message to its associated http server.
- 2 The http server receives the POST message, analyzes the URL, and determines that the message should be processed by the DA. The body of the message is passed on to the agent without additional processing.
- 3 The DA gets the message, parses it, and decides whether a reply is needed (if the POST request was submitted through a form, a reply would be sent, otherwise, only if the requester asked for a reply).
- 4 The DA sends a reply back to the http-server. Jigsaw-based implementations utilize multiple

threads, so the DA passes the reply to the thread, which is responsible for sending it back.

- 5 Finally, the http-server returns a reply back to the client (if needed or asked for).

4 Our Prototype

We have implemented the DA as an extension to the *Jigsaw* [1] HTTP server. Our primary reasons for basing the prototype on Jigsaw are in its object-oriented nature, extensibility, platform-independence, and its support for threading. In addition, Jigsaw supports persistent objects, which makes it a convenient foundation for implementing the next generation data modeling platform[8].

The DA accepts Add-Requests and Delete-Requests one file or directory at a time. The requests have to be submitted using the HTTP POST method, and a client may either create its own form, or use a URL that the DA makes available. The DA keeps track of notification requests in its own database, which it updates as soon as it receives a new request (be it add or delete).

We have also implemented a prototype of the surrogate DA. The surrogate DA accepts the same kind of requests as the DA and acts in the same way when notifying the client. Currently, the prototype DA only sends notifications using socket connections, but we plan to add e-mail capabilities to future versions. We have also supplied a dummy RA that can send requests to the DA and receive notifications. This prototype RA is not associated with a real metadata repository, but we are working on a more complete RA prototype that will be integrated with an HTTP server supporting logical resources.

Both the surrogate DA and the Jigsaw DA generate the same output and respond to the same requests, so it should not matter which one was used to generate the sample output that is discussed in this section. Figures 3 and 4 represent the output of the DA and the RA respectively. We had to remove some of the output to save space, but the dates and relevant information were left untouched, to give the reader a feel for what is happening.

In figure 3 you can see DA events, which include the DA's access to its database, receiving requests and sending out notifications (notice the timing of the DA events with respect to the timing of the RA events in figure 4):

- 1 The DA comes up on a particular port.
- 2 The DA reads its database to retrieve outstanding notification requests.
- 3 The DA completes reading the database and initiates a separate thread for polling files.

```

1) DataAgent: started on socket
   ServerSocket[addr=0.0.0.0,port=0,localport=4999]
2) Read: [/Sofus/data.c|20|paul.rutgers.edu|
   5999|9/12/96 13:35:53]
   DataAgentPoller added file /Sofus/data.c
   with priority 20
   Read: [/Sofus/data2.c|10|paul.rutgers.edu|
   5999|7/31/96 16:50:38]
   DataAgentPoller added file /Sofus/data2.c
   with priority 10
3) DataAgentPoller running...
   (log = log.data)
   (root = /grad/u0/sofmac/Jigsaw/DataAgent)
4) [9/12/96 13:38:07] Checking [root]/Sofus/data.c
   /Sofus/data.c last access: 9/12/96 13:35:53
   [9/12/96 13:38:07] Checking [root]/Sofus/data2.c
   /Sofus/data2.c last access: 7/31/96 16:50:38
5) DataAgent started...waiting for connect
   [9/12/96 13:38:16] Checking [root]/Sofus/data2.c
   /Sofus/data2.c last access: 7/31/96 16:50:38
6) DataAgent got connection... reading input
   Got addRequest. Handling it.
   DataAgent: read [http://paul.rutgers.edu:5999/
   Leon/files.lst 5]
   DataAgentPoller added file /Leon/files.lst
   with priority 5
7) AgentPoller: added this line to DB:
   '/Leon/files.lst|5|paul.rutgers.edu|
   5999|9/03/96 14:53:13]'
   [9/12/96 13:38:24] Checking [root]/Leon/files.lst
   /Leon/files.lst last access: 9/03/96 14:53:13
   [9/12/96 13:38:26] Checking [root]/Sofus/data.c
   /Sofus/data.c last access: 9/12/96 13:35:53
   [9/12/96 13:38:26] Checking [root]/Sofus/data2.c
   /Sofus/data2.c last access: 7/31/96 16:50:38
   ...
8) [9/12/96 13:45:00] Checking [root]/Sofus/data.c
   It has been accessed!!!!
9) Sending to Socket[addr=paul.rutgers.edu,port=5999]
   ...

```

Figure 3: Sample data agent output

- 4 The Polling thread checks on files specified in the outstanding notification requests.
- 5 The DA completes initialization and begins waiting for incoming notification requests, while continuing to poll files.
- 6 The DA gets a new connection from the RA (see step (3) in figure 4).
- 7 The DA reads a new request, adds it to its database and expands the polling.
- 8 The file [root]/Sofus/data.c was accessed and/or changed, and the poller notices this.
- 9 Following step (4), the DA sends a notification to the RA (or RAs) who asked it to keep track of this file.

In figure 4 you can see RA events, which include the RA's access to its database, sending notification

```

RepositoryAgent started socket on port 5999
1) Read in: [http://paul:4999/Leon/files.lst 5]
FileTracker: got http://paul:4999/Leon/files.lst
           priority 5
FileTracker: started to track
           [file /Leon/files.lst]
           [priority 5]
           [host paul:4999]
2) Added [file: /Leon/files.lst pri: 5] to DB
3) [ 9/12/96 13:44:23] RepositoryAgent: sending
requests to data agent
   Sending to Socket[addr=paul.rutgers.edu,port=4999]
   message:
   -----
   AddRequest
   http://paul.rutgers.edu:5999/Leon/files.lst 5
   -----
4) [9/12/96 13:45:00] RepositoryAgent got connection.
   reading input
   [9/12/96 13:45:00] RepositoryAgent: read in
   [/Sofus/data.c]

```

Figure 4: Sample repository agent output.

requests and receiving notifications (notice the timings with respect to the DA output in figure 3):

- 1 The RA gets started and checks logical resources in the metadata repository for references files.
- 2 For each referenced file, adds it to the RA tracking database.
- 3 For each file in the tracking database, sends a request to the proper DA.
- 4 The DA finds out that a file was accessed (see (8) and (9) in the figure 3) and sends a notification to the RA.

5 Other Applications

To design a general solution to the problem of maintaining consistency and integrity of information resources, we have made a clear distinction between logical resources, which are represented by object specifications (sets of references to other resources and presentation methods), and physical resources, which are represented by files. While the function of the RA is to keep track of logical objects, the DA is designed to keep track of files. Our primary motivation was to automate the maintenance of inter-dependent information resources. However, there is no reason that other applications and/or users should not utilize the DA mechanism.

Here are some examples that served as motivation for providing direct access to the DA from third-party processes:

1. A user may want to be notified when any of the static links off his or her homepage become obsolete. In order to take advantage of the DA framework, it would be necessary to list the links, and for each link, send a notification request to the DA that can keep track of the file (or files, if multiple links reference resources on to same machine). If and when one of the files referenced from the homepage gets relocated, an e-mail notification would then be sent back by the responsible agent.
2. The DA mechanism may be utilized to support a distributed software development environment. Here, the flexibility of the DA framework would be advantageous for setting up different notification policies, etc. (e.g., selective notification of only members of the development team that are likely to be affected by the changes).

We have described a small sample of possible applications hoping to provide an insight into the generality of the proposed architecture. It is important that the DA can communicate with any party, be it another DA, an RA, an end-user or a third-party application.

6 Composite Actions

So far, we have only discussed simple actions like **Add-Request** and **Delete-Request**. An important part of our current work involves designing and implementing an action language that would provide a mechanism for composing complex notification requests and enable agents to perform composite actions. Such language would simplify the construction of applications discussed in section 5. In particular, we would like to enhance agents with a wider repertoire of actions that may be performed in response to specific events. Examples of such powerful composite actions include automatic updates to local HTML pages in response to a resource relocation, and creation of an obligation to keep sending regular email notifications to the maintainers of an HTML page referencing the relocated resource until one the following events:

- the obligation expires,
- an acknowledgement is received from the page maintainer,
- there is an indication that the page has been updated to point to the new location (e.g., through the **Referer** header field in an HTTP request for the relocated resource).

A detailed discussion of the action language will have to wait. For now, let us briefly discuss some of the composite actions we would like to make available:

1. **If-then-else**: We would like to be able to specify conditional actions, instead of the basic “when

this changes, do that". The added power should serve the dual purpose of decreasing the number of requests and simplifying the request specification process.

2. **Wait-for-event**(*time-period*, *event*, *default-action*): With this action we in effect create an obligation for the agent to perform a default action unless a particular event takes place within a specified time period.
3. **Delay-and-recheck**(*condition*, *time-period*): The objective of this action is to avoid "jumping the gun" on spurious events (e.g., erroneous creation of a file that gets deleted almost right away; this could potentially lead to two notifications of file updates, when no notification should have been sent).
4. **Repeat**(*condition*, *action*): This specification would enable periodic repetition of a certain action while the condition holds. The condition may express a limit on the number of repetitions, a time limit, etc.

Adding composite actions serves a dual purpose of making agents more powerful and versatile in how they handle different events, while significantly simplifying their use. We are currently in the process of designing a convenient set of composite actions that would provide an intermediate solution while the full power of the action language is not available.

7 Discussion and Future Work

While we feel that the proposed architecture is very versatile and flexible, its components need further work. It is also necessary to provide more detailed specifications of requests and notifications. We are currently working on designing Java classes that extend W3C's Jigsaw server to support virtual resources[8]. The important part of our future work is integrating this server with the Repository Agent. The Repository Agent would use logical objects known to the server to generate notification requests to Data and Repository Agents serving resources that are utilized by these objects. In turn, the Repository Agent should also be capable of processing notification requests that are originated by other agents, which utilize its resources. The Repository Agent would be responsible for updating object specifications and, possibly, content-based indices based on the incoming notifications. Furthermore, we would like the Agent to serve its own control information in response to http requests.

An important issue that we yet have to tackle is security. For example, a malevolent client could ask a DA to poll a file every millisecond, thereby practically bringing down a server. This particular problem is easy to

avoid, by having the administrator specify a minimum polling time for a DA. The DA should have restrictions that an administrator can set to conform with local security measures. Also, a DA should only let a client be able to change, view, delete and query its own requests and prevent it from finding about about the requests of other clients. This needs an authorization scheme, which may be conveniently implemented using the action language. As a short-term solution, we intend to incorporate one of the authorization schemes that are publicly available.

We have not discussed this problem here, but it turns out to be very hard to keep track of moving files. If a user moves a file, the DA will see one file disappear and another one created. That these two have the same name (possibly), same size, etc., could be coincidental. If this file stays on the same machine and the same file system, we could define some heuristics to keep track of the file movement. However, this is more of a quick patch than a real solution. This problem becomes even harder when a file is moved to another machine altogether. There is no easy way to keep track of this other than through an explicit notification. With object-oriented operating systems, the DA could keep track of the OS file objects, regardless of their physical location. This still does not solve the problem of moving files to other machines, and URNs seem to be the best hope for addressing this problem.

8 Conclusions

Our architecture handles some important issues well: it maintains reference integrity keeping network traffic to a minimum and distributes the work. Overall, our architecture is more sound and generates less network load than other existing solutions. Moreover, the more servers begin supporting our approach the more efficient and attractive it would become for others. As more users use the Internet without consideration for how their use of resources affects those around them, solutions like this, which lessen the load, will benefit all. Faster machines and faster networks are not enough, as most people will use as much bandwidth as they can, and a faster connection just means faster graphics, speech, and information. We have to consider how we can most efficiently use what we have, and we feel this architecture is a step in the right direction.

9 Acknowledgments

Our stimulating discussions with L. Thorne McCarty have contributed to this work. We would also like to thank Sue Dumais, Dave Makower and the anonymous reviewers for their comments.

References

- [1] A. Baird-Smith, “ *Jigsaw: An object-oriented server, Design Document* ”, World Wide Web Consortium, 1996.
<http://www.w3.org/pub/WWW/Jigsaw/User/Introduction/wp.html>
- [2] Fausey, Jul, Miller, Shafer, Thompson, Tkac, Weibel, *The PURL Home-page* ,
<http://purl.oclc.org>
- [3] Francis, Kambayashi, Sato and Shimizu, “Ingrid: A Self-Configuring Information Navigation Infrastructure”, *Proceedings of the Fourth International World Wide Web Conference* , pp. 519–537.
<http://rodem.slab.ntt.jp:8080/home/index-e.html>
- [4] *The IETF Uniform Resource Identifiers (URI) Work-Group* ,
<http://www.ics.uci.edu/pub/ietf/uri>
- [5] D. Ingham, S. Caughey, and M. Little, “Fixing the ‘Broken-Link’ problem: the W3Objects approach”, *Computer Networks and ISDN Systems* , 28, (1996), pp. 1255-1268.
- [6] K. Shafer, S. Weibel, E. Jul, and J. Fausey, “Introduction to Persistent Uniform Resource Locators”, *INET96* , 1996.
- [7] L. Shklar, A. Sheth, V. Kashyap, and K. Shah, “InfoHarness: Use of Automatically Generated Metadata for Search and Retrieval of Heterogeneous Information”, *Proceedings of CAiSE’95* , Jyvaskyla, Finland, LNCS #932, Springer-Verlag, pp. 217-230, June 1995.
- [8] L. Shklar, D. Makower, and W. Lee, “Metamagic: Generating Virtual Web Sites through Data Modeling”, *In preparation* .
- [9] *The URL Minder Home Page* ,
<http://www.netmind.com/URL-minder/URL-minder.html>