This is to certify that the

dissertation entitled

MAINTAINING CONTEXTUAL LINK INTEGRITY
IN DISTRIBUTED HYPERMEDIA SYSTEMS

presented by

Ryan L. McFall

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science

Major professor

Date 12/8/2000

Maintaining Contextual Link Integrity in Distributed Hypermedia Systems

By

Ryan Lee McFall

A dissertation

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

Department of Computer Science and Engineering

December 8, 2000

Abstract

Maintaining Contextual Link Integrity in Distributed Hypermedia Systems

By

Ryan Lee McFall

Hypertext systems have proven to be an excellent method of organizing and distributing information. The main feature that distinguishes a hypertext system from other information systems is the inherent capability to describe relationships between various entities using **links**. The importance of links has been known for a long time, but the introduction of electronic hypertext systems has faciliated the increased role that linking holds.

The widespread availability of electronic information and the ease of publishing and modifying that information has led to a phenomenon referred to as **broken links**. This problem occurs when a previously created link can no longer be traversed due to a modification to or deletion of the referenced information resource.

Broken links have been a major annoyance in the most widely used hypertext system to date, the World Wide Web (WWW). With the introduction to the WWW of more sophisticated hyperlinking primitives, the problem will be exacerbated. In particular,

a new class of broken link we call a *contextually broken link* will arise. In this type of broken link, one or more endpoints of the link still exist, but the *content* located at the broken endpoint is no longer correct. This type of broken link is more subtle than the first type, since the system may not be able to notify the user that the link being followed is broken.

In this dissertation, we describe the problems associated with contextually broken links and describe algorithms and communications protocols that can solve these problems. Furthermore, we analyze the efficiency of these algorithms and protocols. We show that these algorithms are capable of detecting and repairing contextually broken hyperlinks automatically in the WWW without requiring significant changes to its established infrastructure.

ACKNOWLEDGMENTS

This dissertation has been the product of much work, sweat, tears and prayer. I would like to take a moment to thank all of the people who have been such a tremendous source of encouragement and strength for me. Without these people, this work would never have been completed.

First of all, I would like to thank Dr. Matt Mutka for overseeing my work as my advisor. I have appreciated his willingness to undertake this research even though my topic was neither of great personal interest to him, nor was it something that he had an extensive background in. I am grateful that Dr. Mutka was patient with me during the times that my teaching took up a significant portion of my time (often probably more than it should have).

A second word of thanks goes to the Graduate School at Michigan State University for providing the Dissertation Completion Fellowship that allowed me to complete this dissertation on time, even while staying home with my newborn daughter for the majority of my final semester on campus. Timely completion of this work without that funding would have been close to impossible.

My parents have provided me with financial and emotional support throughout my life. Without them, I could not have made it through 5 years of college, let alone another 6 years as a graduate student. I am grateful to you both for all the opportu-

nities you provided me while I was growing up, and for your continued, unwavering support while I have been pursuing this dream.

Bernie Holmes listened to my complaining more than just about any other person besides my wife, I suspect. I am grateful to Bernie for his support and suggestions, and for keeping me sane in general. I am sure that he will not forget the many hours I spent muttering about things not working as expected. Thanks for sharing your office with me for so many years!

Dr. Mark Urban-Lurain, Dr. Don Weinshank and Gary McCuaig were also instrumental in completing this work. I thank you for the opportunity to work with you in designing and implementing CSE 101. Mark, you have influenced the way I think about developing software in many ways during the approximately 5 years that we have worked together. I also appreciate your support in all of the decisions I have had to make regarding my graduate school experience.

The faculty at Hope College have been wonderful in their encouragement. Thank you for the excellent preparation for graduate school that you provided me as an undergraduate, and for your sustaining words of advice as I have slowly made progressed towards this point. I am particularly thankful to Herb Dershem for the way he has provided ideas and advice. I am in your debt!

There have been several fellow graduate students who have been crucial to the completion of this work. Stephen Wagner, where would I be without all of your help? I truly pray that you will be able to complete your dissertation in the upcoming year and can join be in finally being able to answer the question "Are you almost done?" with "No, I AM done!" Without Russ Hankey, Dave Hall and Dmitri Perkins, I would

probably still be done, but I would be in much worse shape. Our tennis exploits and weightlifting were a source of both physical and mental renewal. Russ and Dmitri, I thank you for your spiritual companionship over the years.

To the members of my small group and other friends from East Lansing Trinity Church, I am truly in your debt. You have been an incredible source of encouragement for me and have stood by me throughout all of the struggles of this time. I love you all and am so thankful that you have come alongside of me as I learned to trust in God rather than myself to complete this journey.

A very unlikely place to meet two people who would prove to be such a source of strength is Istanbul, Turkey. John Gillette, you have been a brother to me over the past years, and without your prayers and friendship I know that I could never have completed this work. Sara, I am grateful for your friendship and the joy and support you have brought to Leanne's life while I have been pre-occupied with this work. We thank both of you from the bottom of our hearts for the support you have given us throughout this difficult time.

Finally, most of all I would like to thank my wife Leanne. It is unbelievable to me the way that you have managed to live with me and even still like me throughout these past years. Thank you for being so faithful to pray for me and for always providing an encouraging word when I wanted to give up. I am grateful for the way that you have sacrificially spent so much time doing housework and other tasks when I have been too busy to help. Thank you for graciously finding things to do on Saturdays without me while I trudged along trying to complete this work. One thing that we

have learned throughout this time together is that God is definitely faithful to fulfill His promises! I will always love you, princess!

If you are reading this as you embark on this journey, I want to encourage you with these words that have been proven true to me countless times during these years: "Do not be anxious about anything, but in everything, by prayer and petition, with thanksgiving, present your requests to God. And the peace of God, which transcends all understanding, will guard your hearts and minds in Christ Jesus. (Phillipians 4:6-8, the Bible)"

Table of Contents

List of Figures

List of Abbreviations

- DNS: Domain Name System
- DOM: Document Object Model
- HTML: Hypertext Markup Language
- HTTP: Hypertext Transfer Protocol
- ISO: International Standards Organization
- NTP: Network Time Protocol
- PURL: Persistent URL
- SGML: Standard Generalized Markup Language
- TCP: Transmission Control Protocol
- TEI: the Text Encoding Initiative
- URI: Uniform Resource Identifier
- URL: Uniform Resource Locator
- W3C: World Wide Web Consortium
- WWW: World Wide Web
- XML: the eXtensible Markup Language
- XML: the eXtensible Markup Language

Chapter 1

Introduction

The amount of data available through open, distributed hypertext systems, such as the WWW [1], has increased dramatically in the last 5 years. Hypertext, a term whose origin is credited to Ted Nelson [2], is defined by Woodhead in [3] as the "generic class of, or approach to, computer-based materials linked by non-linear structures." Nelson himself defined it as "a body of written or pictorial material interconnected in such a complex way that it could not conveniently be presented or represented on paper." In other words, a hypertext system is one in which the data being presented is linked to other data which are related in some way. The user of the hypertext system does not have to follow a pre-determined, sequential path through the documents making up the hypertext system, but can follow links from "node" to "node" at will.

While data availability is important, it has long been known that data itself does not necessarily comprise knowledge. It is discovering and understanding the relationships between information that constitute knowledge and understanding, which is where the importance of easy to use hyperlinking mechanisms come into play.

The notion of the importance of linking relevant information goes back at least as far as Vannevar Bush's landmark paper "As We May Think" [4]. Bush describes a system where the user is able to create "trails" of related information, where a particular piece of data can be included in different trails in different ways.

Even with traditional knowledge collections, the importance of being able to identify consistently some portion of knowledge has been recognized. The ISBN system was created to provide a means that could uniquely identify a printed book, for example. Separating the Bible into chapter and verse is an example of the need for a consistent method of referring to a fine-grained portion of a work. Both of these mechanisms work very well due to the static nature of the objects being identified. The task of consistently identifying the same part of some resource becomes more challenging when the resource is frequently changing.

Many early hypertext systems were proposed and built in the early days of hypertext research. The main factors that held back widespread acceptance of these systems were their complexity and their dependence on proprietary data formats and software applications. It took the introduction of the WWW, a simple distributed hypermedia system, to exploit the potential of hypertext.

Very simple tools are used to create the content on the web (including the links) and to distribute (serve) that content. In the early days of the WWW, the tool used to author content was a standard text editor, and web servers were simple extensions to the traditional concept of a distributed file system. Indeed, many authors still prefer to use these simple tools for their authoring, and web servers have still not evolved into much more than networked file servers.

The simple hyperlinking model employed by the WWW is another reason that it has been so quickly adopted. Many hypertext systems (for example, [5], [6]) have hyperlinking models that are much more sophisticated than what the WWW allows. Forcing users to learn and cope with the complexity of these models may have contributed to the marginal acceptance of these systems. When it is quite simple to create a link between two information resources, authors are much more likely to do so.

Another reason that the WWW has flourished while earlier hypertext systems have not may simply be that the other systems were implementations of ideas whose time was yet to come [7]. This is evidenced by the fact that as the WWW matures, its users are starting to demand some of the functionality that existed in these more sophisticated hypertext systems [8].

Unfortunately, the simplicity of the WWW has also led to some of the most serious problems facing the web today. In particular, the familiar problem of *broken links* is one that many experts agree must be solved in order to facilitate the continued expansion of the use of the WWW. According to Davis [9], "If the issue of maintaining integrity in hypertexts is not successfully addressed then open hypermedia will continue to be limited largely to relatively static information publishing applications." The goal of this dissertation is to address the problem of broken links while still retaining much of the simplicity that led to the widespread popularity of the WWW.

The remainder of this dissertationis structured as follows. In chapter 2, we give a more complete statement of the problem we will be solving. Next, we explore other

work that has been done in relation to hyperlinking in general (chapter 3) and specifically the issue of maintenance of link integrity (chapter 4). Chapter 5 describes the algorithms we have created to detect and repair contextually broken links. Chapter 6 describes the software agent we have developed to allow the current generation of WWW browsers and servers to provide contextual link integrity, including describing the communication protocol these agents use. In chapter 7 we describe a method of extending the agent which allows a server to maintain contextual link integrity most of the time even without cooperation from other servers. Finally, in chapter 8 we give conclusions and outline further work that remains to be done.

Chapter 2

Problem Statement

As user demands for more sophisticated linking are met, the problems of broken hyperlinks will increase in frequency and cost. Since link integrity is so crucial to the success of a hypertext system, it is imperative that algorithms and protocols be developed which can maintain link integrity on the WWW. This is the fundamental goal of this research.

One of the more important capabilities that more sophisticated hypertext systems provide is the ability to define links that point to arbitrary locations within a text (for example, the Text Encoding Initiative [10, 11, 12]). This capability allows an author to point a reader to a particular part (called a sub-resource) of the referenced resource; in the case where the "reader" is a software agent, the agent can be directed to process only a particular portion of the resource in question. This capability is noted by Engelbart in [13] as one of the more important development paths the WWW must follow.

To facilitate fine-grained referencing of a sub-resource, one must come up with an addressing mechanism to determine the sub-resource that is the endpoint of the link. One such addressing mechanism is *structural addressing*, in which a sub-resource is identified by specifying a path to traverse through the structure of the document. For example, documents written using Hypertext Markup Language (HTML) [14, 15, 16] and the eXtensible Markup Language (XML) [17] can be viewed as a tree structure, and a structural address can specify a particular traversal of the tree from the root node to the desired node of the tree.

Introducing structural addressing as a means of identifying the endpoints of a link brings about a new notion of broken links. The traditional notion of a broken link is a link whose destination endpoint no longer exists. For example, an HTML link can break when the document it references is moved to a new location or is deleted. When this type of broken link occurs, the hypertext system is able to notify the user that the link is no longer valid, since the destination endpoint is not available.

In the new type of broken link caused by structural addressing, one or more endpoints of the link still exist, but the *content* located at the broken endpoint is no longer correct. This type of broken link is more subtle than the traditional type, since the system may not be able to notify the user when the link being followed is broken.

To distinguish between these two types of breakage, we will classify the first type of broken link as an *existential break*, while the second type will be called a *contextual break*, since it is only through examining the context of the link that we can determine whether it is broken.

Both types of broken links can exist in a hypertext system, whether or not it employs structural addressing. In a system that uses structural addressing, an existential break can occur if an endpoint of a link is deleted in such a way that the locator used to identify that endpoint is no longer valid. For example, if a link is created which addresses the fourth section of a four section book, and the second section of the book is deleted, then an existential break has occurred – the endpoint addressed by the locator does not exist. A contextual break occurs when the structure of the document is changed so that the content located by the link is not what the link creator originally intended. In the previous example, if the endpoint of the link was the third section of a book, and the second section was deleted, a contextual break would occur.

For a further example, consider the situation shown in Figure 2.1 This figure shows two XML documents, and the corresponding document trees that result from these documents. The first document is a sports article written about a baseball game that took place the previous day between the Stars and the Moons. The second document is a collection of all box scores for the Stars for the current season. The box score in the article is actually a link to the first box score element in the box score document. Now suppose that the Stars play their second game of the season, and the box score document is updated by placing the second game's box score above the first (perhaps to make it easy to find the most recent box score). Figure 2.2 shows the results of the modification to each of the documents. Because the link in the article document points to the first box score element in the box score document, its content will change due to the modification, and the link "breaks".

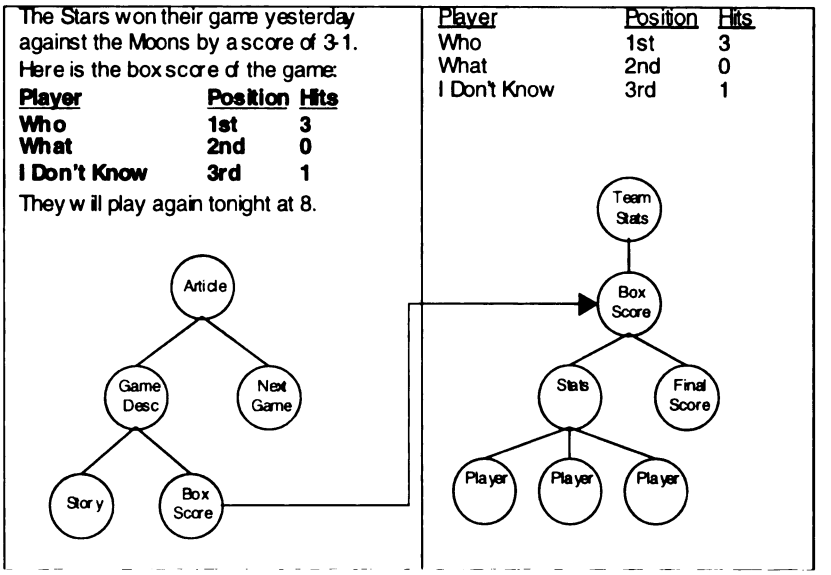


Figure 2.1: The article (left) and box scores (right)

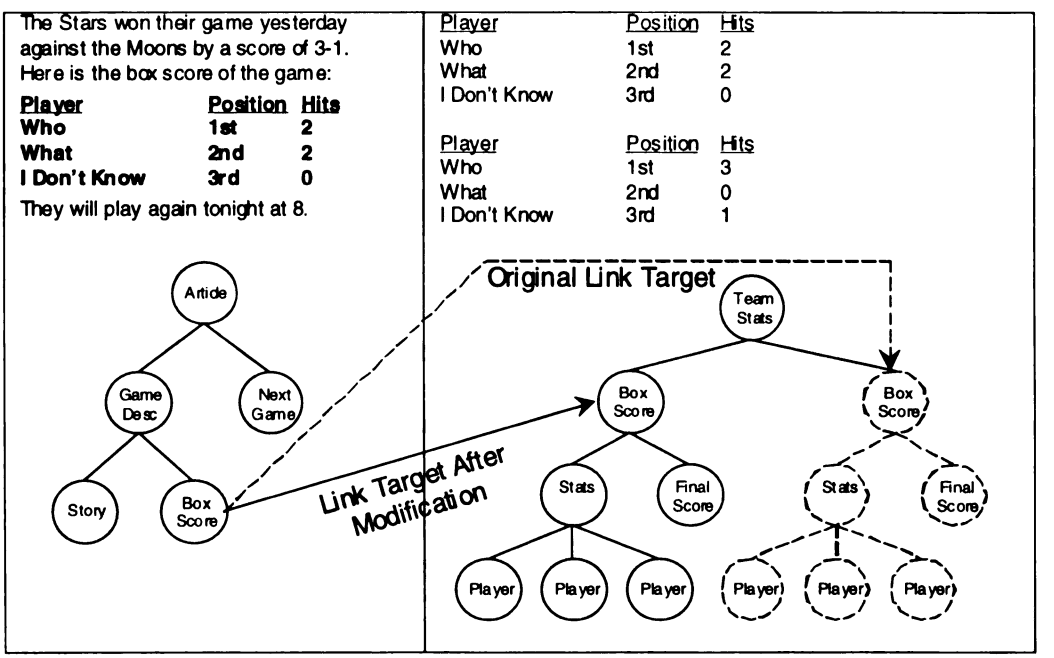


Figure 2.2: The documents after the update

The possibility of existential breaks in a system that does not employ structural addressing is fairly obvious. It is also possible for a contextual break to occur. In the WWW, for example, it is possible to replace some information resource with a resource that has completely different content. However, a contextual break occurs if the new resource is given the same name as the original resource. Contextual breaks are much less likely to occur, however, if structural addresssing is not employed.

The main focus of this work is to solve the problem of contextually broken links. When document modifications take place that cause a contextual break, our goal is to detect this fact automatically and repair the broken links. In particular, we will focus on finding and repairing broken links in documents written using XML in the context of the WWW. Our work is not necessarily restricted to this context, as it applies in any system using structural addressing where the documents in question are tree-structured.

XML was created by the World Wide Web Consortium (W3C) as a successor to HTML, the original markup language used to describe resources on the WWW. There were many reasons that the decision was made to replace HTML with XML. One of the most compelling reasons guiding the development of the XML specification was the desire to eliminate the presentational nature of markup data, which would allow the markup to provide semantic information describing the nature of the data. In order to do so, one of the first changes that needed to be made was to eliminate the dependency on a fixed set of elements (commonly referred to as tags) that are available to mark up data. In this sense, XML is not a specific markup language,

but rather a "meta" markup language that defines a common structure for an infinite family of markup languages.

Rather than completely building a new standard for this meta markup language, the designers of XML leaned heavily on the Standard Generalized Markup Language (SGML) [18] standard. SGML is very complex, which led to the designers of XML to eliminate many of the more esoteric features of the SGML language.

SGML provides a fairly simple means of creating links between elements in the same document [19]. Each element can have an attribute with a special type called an *ID*, which must be unique within a document. Any other element that wishes to reference this element does so by having an attribute with a specific type called an *IDREF* element. The SGML processor enforces the restriction that all ID's used within a document are unique, and that any attribute of type IDREF contains a value that points to an existing element, within the same document, containing the referenced ID value.

SGML does not provide any mechanism that can be used to provide inter-resource linking. In order to create an inter-resource link, an author must use a system that builds upon SGML, such as HyTime [20] or the Text Encoding Initiative (TEI). The designers of XML realized that there is a need for a middle ground between the simplistic linking models of HTML and SGML and the complexity that is inherent in standards such as HyTime. Therefore, in addition to creating the XML standard describing the ways that documents would be marked up, they authorized the work on a standard means of describing links between XML resources (the XLink standard

[21, 22]), and also a standard way of addressing sub-resources of those resources (the XPointer standard [23, 24]).

XLink allows links to be created that are multi-ended, bi-directional, and out-of-line. In this way it was influenced by the linking model specified in the HyTime standard. In order to specify links between documents, the standard Uniform Resource Locator (URL) reference means of locating a document on the WWW is used. If the endpoint of the link is the entire resource located at that particular URL, no further addressing is necessary. If, on the other hand, a sub-resource of the resource designated by the URL reference is to be the link's endpoint, a fragment identifier is appended to the URL, and the fragment identifier is then used to locate that sub-resource.

If the resource named by the URL reference is an XML document, then the fragment identifier portion of the reference is an Xpointer. For complete information on the ways that Xpointers[1] can be used to locate a sub-resource, see the specification. Xpointers can be used in a way similar to an HTML link or an SGML IDREF reference to point to a uniquely named element within the target resource. Additionally, they can specify a traversal of the target document tree using locator terms such as *ancestor*, *child*, *sibling*, etc[2]. In this way they are similar to the TEI and HyTime primitives.

[1] the current practice is to refer to the standard as XPointer, while a particular instance is called an Xpointer

[2] Although the XPointer Candidate Recommendation [24] has changed the syntax used to specify tree traversal, the semantics remain unchanged. The version in [23] is somewhat easier to read, and so we will use its notation in this dissertation.

In addition to allowing the specification on a particular node in the document tree, an Xpointer can also refer to a *range* of nodes, or to a specific portion of the content of a set of nodes.

The addressing mechanism outlined in the XPointer specification essentially uses structural addressing, and thus is susceptible to the problem of contextual link breaks. There are at least two major problems that must be solved in order to provide guaranteed contextual link integrity for a hypertext system.

The first problem is the necessity of algorithms that are able to locate the original endpoint of a link after the document containing the endpoint has been modified in a way that causes the original link to break. While simple string search algorithms would be sufficient to perform this task, they are not efficient enough to be practical when the referenced document is large and there are a large number of links referencing that document. The fact that the hypertext documents we are considering are very regularly structured allows other algorithms to be developed which realize considerable performance gains over a simple string searching technique. Furthermore, traditional string searching algorithms are meant to deal with plain text only, and would need to be modified to understand the difference between content and markup in a hypertext document.

The second problem that must be solved is to allow the introduction of link integrity guarantees into the WWW without requiring substantial changes in the protocols and document markup languages currently used to implement the WWW. As Bouvin states,

> Adhering to standards has a large part of the success of the Web and the very size of the Web has enormous inertia, so an attempt to replace the Web with something perhaps more advanced in certain aspects is, if not doomed, then up against trememdous odds [25].

Protocols must be developed that allow information providers on the WWW to remain as independent as possible. We must not require all authors to use a specific authoring tool along with a specific web server, since systems with these restrictions are not likely to gain widespread acceptance. Instead, a mechanism must be developed where the users are still free to use whatever tools they are accustomed to author and distribute documents.

The problem of maintaining link integrity while still allowing web systems to remain independent is a challenging one. In order for a system to successfully determine which links have been broken by a document modification, it must have knowledge of the existence of each of the links that point into the document in question. Acquiring this information independently (that is, without explicit notification of link creation by the referring document's author) has been proven impossible [26], so we must come up with an unobtrusive means of communicating this information between servers.

Part of any system that attempts to prevent contextual link breakage will require frequent parsing of structured hypertext documents. For a server that handles heavy traffic, fast and efficient parsing is of paramount importance. The widespread existence of multiple processors in today's computing environment leads to the possibility of parsing documents in parallel. Parallel algorithms must be developed to take advantage of the processing power that is available.

After surveying the capabilities of several historical hypertext systems and their approach to solving the broken link problem in chapters 3 and 4, we present our solutions in chapters 5, 6 and 7.

Chapter 3

Background

In this chapter, we discuss several previous hypertext systems and the capabilities that they provide in terms of linking and link maintenance.

3.1 Dexter Reference Model

The Dexter Reference Model [8, 27] is not a specific hypertext system. Instead, it is intended to be an abstract model of a generic hypertext system. While the specific notion of document and link can be very different among various hypertext systems, it is helpful to specify an abstract model of the system so that reasoning can be done about the properties and behavior of the abstract model. This reasoning will then be applicable for any actual system for which a mapping can be found from that system's data and linking model to the Dexter model. This idea is similar, for example, to the motivation behind the OSI seven layer network model [28].

A second goal of the Dexter model is to provide a common base of terminology for the hypertext community. Every hypertext system has its own set of terms and definitions that it uses, which can make comparison between systems difficult. The Dexter model was in part created so that hypertext researchers might be able to avoid communication difficulties due to a simple difference of vocabulary.

As is the case with any abstract model, the Dexter model uses the terms it defines in a way that is separate from any specific hypertext system. For example, the term assigned to refer to data objects in the system is a *component*, while common hypertext systems often refer to these data objects as *documents* or *nodes*.

Parts of the Dexter Model

The Dexter model separates an abstract hypertext system into three layers: the *runtime layer*, the *storage layer* and the *within-component* layer. These layers are listed in decreasing order in terms of level of abstraction in the software design process. This means, for example, that the *within-component* layer is closest to the physical representation of the data in the system, while the *runtime layer* is the view that the user of the hypertext system sees. Interestingly enough, it is the middle layer of this model (the storage layer) that the Dexter model chooses to focus on. The within-component and runtime layers are deemed to be too diverse for the creation of a general model which can cover the range of possibile implementations for these layers.

The purpose of the storage layer is to model how "nodes" and "links" are put together to create a network of information. As previously mentioned, the Dexter term for node is component. A component is an abstract data object, with no restriction of the type of object (text, graphic, sound, etc.) or the format in which the object is stored.

Along with defining the notion of component, the model also specifies two functions that can be used to identify and access components as part of the storage layer. The first function is the *accessor function*, which is responsible for taking a unique ID (UID) which all components have and finding the component in question. In the WWW, for example, this corresponds to the idea of the GET method operating on a URL.

The *resolver function* is reponsible for mapping a more general description of a component to the UID of the component. This function might be responsible for mapping a request such as "The component describing the life of Martin Luther King" to a specific UID of a component matching that description. The WWW does not have an analogue to this function; the URL of a document is generally the only way to retrieve its contents, and corresponds to the specific UID in the Dexter model.

To facilitate addressing parts of components, the Dexter model defines the concept of *anchors*. Anchors are also made up of two parts, similar in functionality to the roles of the accessor function and the resolver function. The first member of an anchor object is an *anchor ID*, which is a unique identifier within the scope of the component the anchor is associated with. The value of the anchor ID remains constant throughout the lifetime of the anchor. The second member of an anchor is the *anchor value*. This

is an arbitrary value which can be used by the within-component layer to resolve to a specific part of a component. In order to maintain the separation between the within-component and storage layers, the anchor value has no meaning to the storage layer.

For this dissertation, it is interesting to note that the model states that:

> As a component changes over time (e.g., when it is edited within the run-time layer), the within-component application changes the anchor value to reflect changes to the internal structure of the component or to reflect within-component movement of the point, region, or items to which the anchor is conceptually attached. The anchor id, however, remains constant, providing a fixed referent that can be used to specify a given structure within a component [8].

It is the implementation of this behavior within the within-component layer into which the algorithms defined in chapter 5 fit.

Once we have the notion of components and anchors, it becomes possible to describe the mechanism used to describe an endpoint of a link. This mechanism is called a *specifier* and its main feature is a (component specification, anchor ID) pair. A link in the Dexter model is simply a set of two or more specifiers, providing links that are multi-ended as well as bidirectional [1].

A further aspect of the Dexter model is a specification of a set of abstract functions that can be used to operate on the model contained in a hypertext. Some of these operations are of particular interest to this dissertation:

[1] the direction of an anchor is also contained as part of a specifier

- The function LinksToAnchor takes an anchor and its containing component as input, and returns the set of links in the hypertext that refer to the anchor in question.

- The function LinksTo takes a hypertext (which is a enumerable set of components and links) and a component ID as input, and returns the set of links resolving to that component.

- The function ModifyComponent modifies a component, ensuring that the component remains consistent, and that the resulting hypertext remains *link consistent*. Here link consistent means that the component specifiers of every link specifier must resolve to existing components.

Although not the major focus of the Dexter model, we briefly outline the functionality of the runtime layer. This layer is responsible for presenting the nodes and links of the hypertext to the user, facilitating browsing and authoring of the hypertext. It is through this layer that the user *instantiates* components of the hypertext, making cached copies of those components currently in use during that particular session.
In particular, the run-time layer is reponsible for making the presence of anchors and links within the hypertext known to the user, and controlling traversal of those links. It is also responsible for handling session related information, such as a history of the components visited, and providing run-time versions of the storage layer's resolver function. Finally, it provides the means through which components are instantiated (created from their storage layer models) and written back into the storage layer, a function known as *RealizeEdits*.

Applications of the Dexter Model

At the time of the development of the Dexter Model, the editors of the reference state that "The model as currently stated is far more powerful than any existing hypertext system. The provisions for n-ary links and for composite components, for example, are intended to accomodate the design of future hypertext systems."

The design of the linking and addressing specifications for the second generation of the WWW proposed in coordination with the XML specification fulfill some of the advanced design goals of the Dexter system. However, some aspects of the Dexter model have been left as unimplemented by the WWW. In particular the link consistency requirements stated by the model are unfulfilled, a deficiency this dissertationaddresses.

3.2 Intermedia

The Intermedia system [29] is significant because it is one of the first hypermedia systems to recognize the importance of two hyperlinking concepts. Of these concepts, of particular importance to this dissertationis the idea that link anchors should be constructed as "any entity that the user can select in that particular application." Of course, if link anchors are to be allowed to be specified in this general a way, there must also be some way of identifying the portion of the resource that the user selected. In the Intermedia system, it is the responsibility of each individual editing/browsing application to implement this function. Since the data needed to identify an anchor

in a text document may be significantly different than what is necessary to identify an anchor point in a graphic, this is a reasonable model.

The second important concept introduced by the creators of the Intermedia system was that it should be possible to link to any information resource that the user could access. The Intermedia creators found that forcing hypertext users to remain in a "closed" world as part of a hypertext system was one of the main reasons that acceptance of early hypertext systems was not as widespread as it might have been. They envisioned a world where any information resource may be the endpoint of a link; for example, a word processing document could be linked to a spreadsheet that was linked to a sound clip.

The Intermedia system means of accomplishing this "universal linking" is through integration of linking services into applications through the use of an object-oriented programming framework [30]. Link creation is performed through menu operations "Start Link" and "Complete Link", a metaphor quite similar to the "Edit/Paste" operations users of graphical user interfaces are quite familiar with. Most of the details of creating links and the initiation of following links is handled by the application framework. This, of course, limits the degree of "openness" related to linking present in the system, since it is only those applications that have been derived from the application framework that can participate in linking.

The development of the Intermedia project ended in 1990, and as a result a study [30] was performed which evaluated the usefulness of various linking capabilities such as bi-directional and multi-ended links. This study established their importance as necessary for any hypertext system in which effective work can be performed.

3.3 Hytime

The HyTime standard [5] is an International Standards Organization (ISO) document outlining a standard way of describing the structure of time-based hypermedia documents [31]. As such, it can be used to implement hypermedia systems that are made of much more than simple hypertext documents. By using just a subset of the facilities HyTime provides, however, the standard can be used to describe a hypertext facility.

Linking in HyTime

HyTime provides a linking model that includes two different kinds of linking constructs: *contextual links (clinks)*[2] and *independent links (ilinks)*. A contextual link is used to provide the type of link that users of the WWW are familiar with. In this type of link, one of the endpoints of the link is the actual linking element. The link asserts a relationship between the linking element itself and the element it references. A normal use for this type of link is as a cross-reference. For example, a link might be created between a term in a document and the definition of that term in a dictionary document.

An independent link is a link that connects more than two locations, is stored in a document other than the documents it is connecting, or both. One use of this type of link might be to connect an article to a series of reviews of that article by different

[2]It is important to note that the use of this term is quite different than the notion of contextually broken links that we introduced in chapter 2

reviewers. Or, an ilink might be created to connect a specific word in a document to entries in a dictionary, a thesaurus, and an encyclopedia.

Links that are located in a resource other than the resources connected by the link are called *out of line* links. These types of links are useful when the resources to be connected are not owned by the entity wishing to create the link, and are therefore read-only to that entity. For example, suppose an instructor uses a particular on-line text for an operating systems course. A student in the course wishes to create links from the textbook used in the course to several different online operating systems texts, so that the student would be able to examine several different discussions of the same topic. Since the student most likely does not have write permission on the course textbook document, it would be impossible to add the link directly into the document. However, using out-of-line links the student could create a separate document containing the links. While reading the course textbook, the hypertext system is responsible for determining which of the nodes in textbook have links pointing to other textbooks.

Addressing in HyTime

HyTime provides three different ways of addressing sub-resources within a particular resource. The three types of addressing include addressing a node by name (using an ID attribute), specifying a series of counting terms that specify the location of the sub-resource, and through a general-purpose querying facility.

In general, specifying a sub-resource by name is similar to ID/IDREF mechanism provided by SGML. It is also possible to use this type of addressing to locate a named entity. An entity is a peice of data that can be referenced using a name explicitly declared for it. The entity itself may be something as simple as an abbreviation for a literal string, or may represent an external data object such as a file of SGML or non-SGML data [19].

When using counting to find a sub-resource, HyTime provides an extremely general counting mechanism. A counting locator in HyTime is called a *dataloc*. A dataloc divides information content into components (called *quanta*), and then counts the quanta [31]. Different means of breaking the content into components are possible. For example, one quantity that might be counted would be the bytes of the digital representation of the document. Another quantum might be time divisions in a digital representation of a video recording. Or, the divisions might be the markup tags indicating the elements of an SGML document.

A specific type of dataloc called a *treeloc* is of particular interest. A treeloc is used to specify a traversal of a tree representation of a resource. HyTime does not specify that resources addressed using a treeloc must be SGML documents. Any resource that can be represented using a tree can be addressed using a treeloc locator. However, they are particularly useful for SGML documents since all SGML documents can be conveniently represented as a tree.

The treeloc element specifies a traversal that starts at some location source, and follows a specific path down the tree. A second type of tree location term is a *relloc*, which is a more general form of treeloc that also allows traversal up the tree, and also

across the tree. It does this through the specification of one of five types of traversal as part of the relloc locator. These five types of traversal are *anc*, *esib*, *ysib*, *parent* and *children*, which specify the ancestors, elder siblings, younger siblings, parent, and children of the location source, respectively.

A further type of counting locator is the *fcsloc*, which is used to specify some location using a coordinate system. This type of locator is useful for identifying portions of graphical images, for example.

The final category of locators that HyTime provides are *query locators*. This allows the specification of a query, in some query language, that is executed to find the desired sub-resource.

The querying facility of HyTime is perhaps the most robust, but coming up with a query to identify a particular sub-resource is difficult, since computer understanding of text is still not very reliable.

3.4 Hyper-G

The designers of the Hyper-G network information system see the WWW as a first generation attempt at a world-wide distributed information service [32]. They recognize several problems with the current state of affairs on the WWW that they intend to address [33]:

- Provide orientational and navigational aids
- Provide automatic structuring and maintenance

- Reduce fragmentation across servers
- Support user identification and access control
- Support multilinguality
- Maintain interoperability with existing systems

Hyper-G addresses each of the fundamental shortcomings of the WWW. Like the WWW, Hyper-G is a client-server architecture. One of the fundamental differences between the WWW and the Hyper-G system is that in Hyper-G, the underlying mechanism used to store information within a hypertext is an object oriented database, as opposed to the WWW where the storage mechanism is (usually) a traditional file system.

Placing all information available to the system under control of the Hyper-G server makes the maintenance of link integrity much simpler, since the server is able to determine which other information objects will be affected by a change, and can notify those objects of changes necessary to maintain consistency. Change notifications are made through a scalable flooding algorithm [34] so that all other servers are kept up to date.

However, perhaps the major disadvantage of the Hyper-G approach is that

> ... documents have to be imported into the environment from the development environment, they have to be translated into a format for which a Hyper-G viewer has been written, and more generally that they have to be owned by the database, i.e. brought into the authors domain of authority **in other words, you can't link things you don't own (emphasis mine) [35].**

Chapter 4

Related Work

4.1 Maintaining Link Integrity

Davis [9] gives an overview of the various strategies that an open hypermedia system can take when attempting to solve the broken link problem. In the following paragraphs, we give an outline of these strategies along with an analysis of the strengths and weaknesses of each approach.

4.1.1 Existential Breaks

The simplest approach to preventing broken links is to put the burden entirely on the user to repair any links that might become broken. Although this approach sounds overly simplistic, it is in fact the method used to maintain link integrity in the largest hypertext system currently in use, the WWW. Indeed, the simplicity of this approach may well have led to the rapid widespread acceptance of the WWW. However, there are at least two serious problems with this approach. The first of these problems

is a human factors issue: people are simply not consistent in performing the work necessary to maintain link integrity. A recent study by the W3C [36] found that between 5% and 8% of links referenced by WWW users are broken.

The second problem with this approach is that it requires that document authors are aware of the existence of links that point into their documents, and have some mechanism of notifying the owner of the referring document to change the link. It is this second issue that makes the practice of manual link maintenance infeasible.

In [37], Creech describes an extension to this mechanism by which the system provides some support to the user. He defines the concept of a *Change Log Table*, which records the significant changes that have been made to a document during an editing session. Periodically a process called the *Web-Walk* process reads the Change Log Table and either automatically makes a change made necessary by the operation, or notifies the owner of the referencing document if the appropriate action cannot be determined automatically. The entries in the Change Log Table are made by the authors themselves, which allows them to use whatever editing tools they normally use to author their documents. Unfortunately, since the logging of changes is a manual process, dependent upon the authors doing it (and more importantly, doing it correctly), this approach is quite susceptible to error and very difficult to do correctly. Furthermore, there is a period of time during which broken links exist, dependent upon the length of time between Web-Walk's.

Another approach to dealing with the problem of broken links (either contextual or existential breaks) is to attempt to provide a permanent name, independent of location and structure, for the object that is to be identified. This type of approach

can be further broken down based on who/what is expected to maintain the mapping between the current location of the object and the permament name. Generally, this can be either the human owner of the object or a software program acting on the owner's behalf.

Persistent URL (PURL)'s [38] are one example of the former strategy. The relationship between an object's PURL and its current, location dependent name (URL) is similar to the relationship between a host name and an IP address in the Domain Name System (DNS) [39]. A PURL server provides functionality similar to a DNS server. It takes requests for a PURL and redirects those requests to the URL that is the current location of the object identified by the PURL. If the owner of the object decides to alter the location of the object, it is up to the owner to send a message to the PURL server(s) in charge of mapping that PURL.

The main advantage of providing this level of indirection is that it does not require a document to know about the documents containing references to it. By providing the redirection, the notification of a change in location for a resource only needs to occur once, and the owner of the resource that has changed location should have permission to make this change. There are two main disadvantages to this type of approach. The first disadvantage is the requirement of manual update of the link resolution service (the PURL server), since we know that a repetitive manual task is likely to be forgotten or ignored by a human operator. The second disadvantage of this system is that it only provides link integrity for links that point to resources that have been deemed important enough to have been assigned a name by the author

of the resource. This does not allow users to create links to sub-resources of the the resource that the author may not have anticipated.

The PURL approach centralizes the management of the names used to identify resources, but still allows those resources to be maintained in any way the user desires. Another approach that can be taken is to centralize the management of the information stored in the hypertext system in such a way that the system is able to easily maintain the integrity of links. To do so requires that users of the system interact with the hypertext system through specific tools such as specialized editors and servers. Through the use of these tools, the system can be made aware of any changes that are taking place that may break existing links, and update any references to links that would be broken by the modification in question. Hyper-G [6], described in section 3.4, is one such system that takes this approach. The W3Objects architecture [40] is another system that adopts this approach.

The main disadvantage with this approach is the closed nature of the system. Users do not like to be told what tools to use, and often may not be able to use the tools due to their unavailability on the specific computing platform on which they are working.

4.1.2 Contextual Breaks

The approaches mentioned above work best to deal with existential breaks. Further approaches are necessary to deal with the problem of contextual breaks. Some of the approaches that have been proposed include versioning, link aware editing tools, "just-in-time" link repairs and using queries to specify the endpoints of links.

In the versioning approach, the problem of broken links is eliminated because when a link is created, it points to a specific version of the referenced resource. If that resource is subsequently changed, it is not replaced by the new version of the resource; instead the new version and the old version co-exist. Since the link refers to the old version of the document, it will not be broken by the introduction of the new version. Of course, in a versioning approach, the amount of storage space required for the resources stored in the hypertext system increases in proportion to the number of times the resources are changed. In addition, the risk of users editing or referring to old versions of the resources is present.

Davis mentions the "diff" approach, which could be used instead of maintaining separate versions of the document. In this system, the difference between an original version of a document and a modification of the document is computed. Then a "just-in-time" repair method can be used which uses the difference file to resolve a link that is known to be broken. Implementations of diff exist that can be used to create a byte-by-byte difference file, which would be useful if links use byte addresses to specify link anchors. Byte addresses are one of the least robust means of identifying link anchors, which means that a large fraction of the links might be broken by a minor edit to the document. Furthermore, they are very unintuitive and difficult for users to work with [31].

When using structural addressing mechanisms, a difference algorithm that is able to operate on tree-based structures is needed. In [41], the authors give an algorithm that can do so in $O(nd)$ time. However, even with efficient algorithms that compute a useful difference, determining the difference between two documents involves more

work than really needs to be performed, since we are only interested in the changes to the parts of the document that are currently the endpoints of links.

More recently, in [42], Phelps and Bilensky describe a system using what they call "Robust Locations." A robust location consists of a series of location methods, including a core set of methods made up of unique ID's, tree-walks and context. Further sets of location methods can be added by implementors if so desired. Robust locations are complex enough that the authors suggest they always be machine generated.

This method requires that repair of broken links be left up to the client, using the information encoded in the robust location. They propose a specific repair algorithm that attempts to take advantage of the tree-walk portion of the location and does a "spiralling" search of the content around the original location attempting to find the originally intended content. If the tree-walk portion fails according to some predefined threshold value, a string matching algorithm is used to try to match the context of the link.

A main point in which this dissertationdiffers from the Robust Location proposal is in where the computation providing link integrity is performed. Phelps and Bilensky argue that reattachment should be performed by the browser. This requires significant new implementation by the browser, and also means that it is impossible for an organization to guarantee that all users will be prevented from following a broken link. Instead, only those users that use a reattachment capable browser will be guaranteed contextual link integrity.

It is our opinion that allowing the server to perform link reattachment is a better option for information providers. Once the server implements link maintenance pro-

cedures, any user browsing that server's documents will be ensured that links pointing to that server will be correct.

Using queries to specify the anchors of links seems like a reasonable approach, and in some cases is a very good approach. For example, if one is creating a link to a list of items in a catalog that fall into a certain price range, a query mechanism seems perfectly reasonable. When links are created based on the content of the resource being linked to, rather than being based on *attributes* of the that resource, however, querying is not really a possibility, since it is difficult to construct a query that uniquely identifies the desired content (and only the desired content).

Another useful classification of the various approaches to maintaining link integrity is one that determines how "open" the solution is. That is, to what degree is the user of such a system able to interact with the system in any way that he/she desires? It is our goal to be able to provide a system that provides a high degree of correctness in an environment where the user is completely unaware of the existence of the link integrity problem. Furthermore, if we require the user to have a small amount of responsibility, then we should be able to guarantee the integrity of links in all cases. Davis provides a categorization which classifies integrity maintenance approaches into the following classes:

- Don't Bother: In this approach, the user makes links, and if the links break it is determined not to be that important.

- Avoid the Problem: Links are expressed declaratively or in terms of an algorithm that will hopefully resolve to the intended anchor.

- Loosely Coupled: The system itself provides no means of guaranteeing link integrity; but it provide a set of tools that the user can use to do so. If the user chooses not to use the tools, link integrity is compromised.

- Automatic Link Repairs: The system knows that links can be broken, and takes responsibility of detecting and repairing broken links.

- Tightly Coupled: The most restrictive class, systems that adopt this approach integrate link management responsibility with authoring software and servers.

4.2 Our Approach

Our approach to maintaining link integrity combines many of the concepts mentioned in the above work, while attempting to provide solutions to many of their disadvantages. In Davis' categorization scheme described above, the approach falls somewhere between "Loosely Coupled" and "Automatic Link Repairs." We hope to be able to provide automatic link repairs that are guaranteed to work if the user uses a (minimal) set of tools to notify the system about the existence of links and inform it of changes to the documents it manages. When the user neglects the responsibility of using the tools, the system should be able to gracefully deal with the situation and still provide link integrity with a high probability.

We have developed algorithms that can efficiently detect and repair broken links. These algorithms require knowledge of the links pointing into a document and prior

notification when a document is about to be replaced with a new version and are described in detail in chapter 5.

We have additionally developed a communication protocol and WWW agent service that can be used to provide a "link mapping" service so that existing web servers can take advantage of the algorithms described above in order to provide link integrity.

One thing that we are not proposing to deal with is the possibility that the content specified by a specific link may have been modified in some way, or perhaps even deleted. Strategies for solving this problem would involve devising some sort of semantic distance metric that could be used to measure the distance between a portion of the content in a modified version of a document and the original content, and then deciding whether any portions of the modified document are "close enough" to be considered the original content. Indeed, the recent "robust locations" work cited previously falls into this category. While this is certainly possible, we do not consider it in this dissertation.

Recently, many systems and methods have been proposed for implementing "open hyperlinking", which allows the users of a hypermedia system to create links between information of widely varying types (for example, DLS [43]). Bouvin provides a good summary of these approaches in [25]. Providing guaranteed link integrity in systems where the endpoints of links can be arbitrary types of information is a more ambitious goal than this dissertationwill address.

Chapter 5

Finding and Repairing Broken Links

This chapter outlines algorithms that efficiently detect which links will be broken by a document modification. Furthermore, once it is determined that a link is broken, we describe algorithms that can locate the referenced node in the modified document in some restricted cases.

5.1 Detecting Broken Links

Definitions

$H_s(N)$: the height of the sub-tree of the document rooted at node N, where height is defined in the traditional sense.

$E_s(N)$: an encoding of the element structure of the sub-tree of the document rooted at node N.

Link target: A node in a document tree which is directly pointed to by a link

Contained links: A link L_1 is **contained by** a link L_2 if the link target of L_1 is a descendant of the link target of L_2.

Document tree: The tree representation of an XML document. The representation we will use is the Document Object Model (DOM). We give a simple description of the DOM here; for a more complete description, see the standard [44]. In the DOM, a document tree is made up of several different types of nodes, with the most significant types for this discussion being the **element** and **text** nodes. Element nodes represent the element markup within the document; text nodes encapsulate the document content contained by the elements and by definition are leaf nodes. Note that all document content is contained by some element.

Link tree: A tree whose nodes are the set of nodes that are targets of links for a particular document. The nodes of this tree are a subset of the nodes in the document tree representing that document. The link tree has an artificial root node as its root which is not a node in the document tree. This is necessary in order to guarantee that the link tree is tree.

Node equivalency: Node comparison is an overloaded operation. Two element nodes are considered equal if their tag names are the same and they have the same number of children. Two text nodes are considered equal if a string com-

parison of the text contained in the text nodes shows them to be equal. We will use the symbol $\equiv$ to denote the node equivalency test.

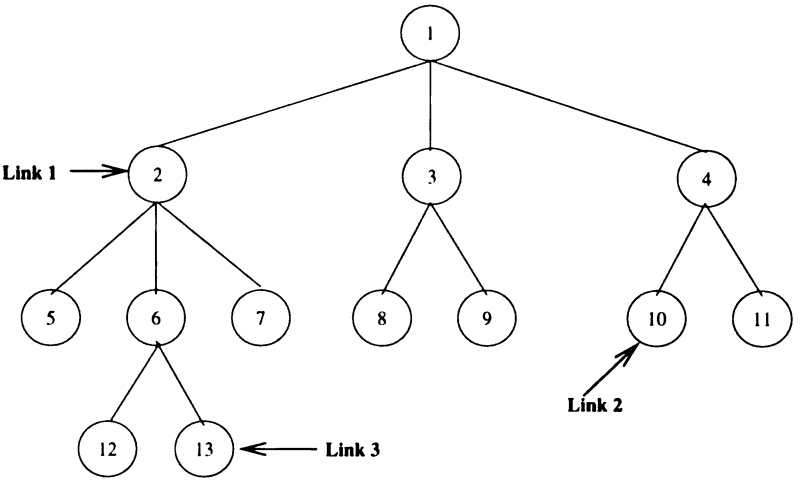
5.1.1 Increasing Detection Efficiency

The brute force method of detecting broken links involves checking each of the links to the original document and deciding if the content of the node(s) referenced by the link has changed in the modified document.

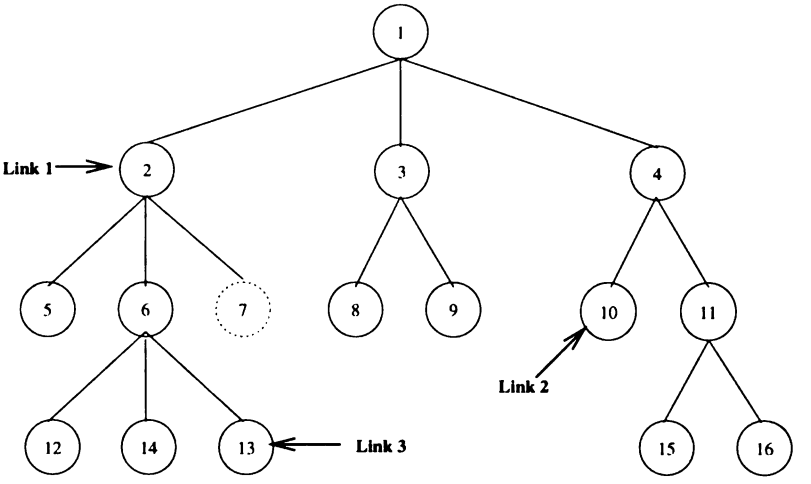
However, it may be the case that hundreds or even thousands of links to the document exist. A sequential walk through the link list, checking each link independently, could take a substantial amount of time. We can certainly do better than this due to the hierarchical structure of XML documents.

One way of reducing the amount of work required to detect broken links is structuring the order in which the links are examined. When there are a large number of links to a document, it is likely that some of the links will contain other links. Since a containing link should be considered broken if any of its contained links are broken, we can reduce the amount of work done by the link detection algorithm by checking broken links in a "bottom-up" fashion.

Consider the example shown in figure 5.1. In figure 5.1(a), we see a tree representing the original version of a document. Figure 5.1(b) shows the document after one node has been modified, and 3 new nodes have been inserted. Figure 5.2 illustrates the nodes that would be checked by a "brute-force" algorithm that compares the content



(a) The Original Document



(b) The Modified Document

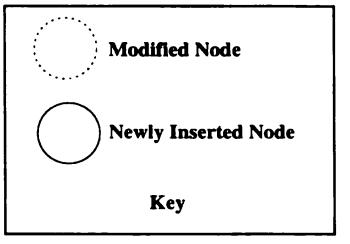


Figure 5.1: A document and its links before and after modification

of each explicit target in the original tree to the corresponding nodes in the new document tree.

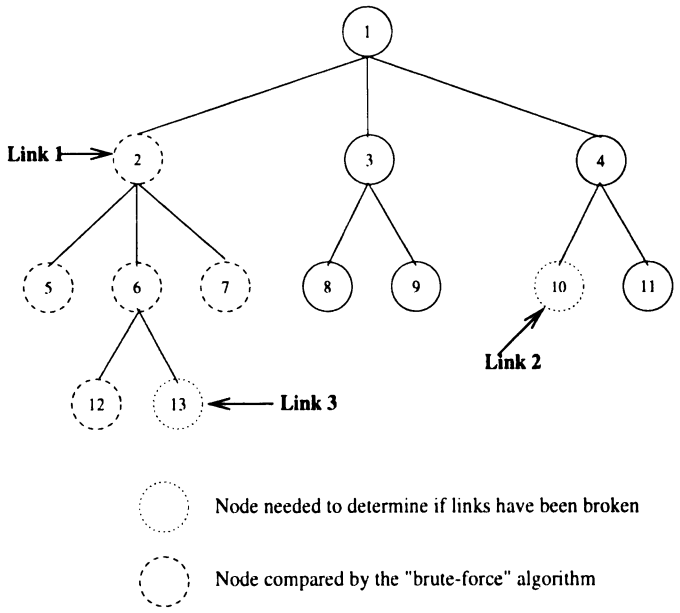


Figure 5.2: Nodes examined by the brute-force algorithm

We can see that this algorithm clearly does more work than is necessary, due to the hierarchical nature of XML document content. If the first comparison performed by a broken link detection algorithm was to determine if link 3 is broken, we could avoid comparing nodes 2, 5, 6, 7 and 12 altogether, since the target of link 3 is the second child of node 6, and that node is an implicit target of link 1.

In a large document, structuring the link database so that links are checked in a "bottom-up" fashion can eliminate large amounts of work that could be spent comparing nodes unnecessarily. This link structuring is the fundamental idea in our broken link detection algorithm.

5.1.2 The Algorithm

Let $Tree_{Old}$ and $Tree_{New}$ be the tree structures representing the old version of a document and a new version of the document which is about to replace the old version, respectively. Let $Tree_{Links}$ be a link tree for $Tree_{Old}$. The links in $Tree_{Links}$ are structured in a way such that a node *child* is a child of a node *parent* if *parent* contains *child* (We give an algorithm for creating $Tree_{Links}$ in algorithm 5.3).

Our goal is to determine which of the links in $Tree_{Links}$ should be marked as modified, where a link is considered modified if the explicit target of the link or any of its descendants are not equivalent in the new tree structure. We wish to do this in such a way as to perform as few node comparisons as possible.

Algorithm 5.1 is used to decide if the target of a link has been modified. This algorithm is used by algorithm 5.2 to examine the targets of each link in $Tree_{Links}$. At the conclusion of algorithm 5.2, all of the explicit targets have been correctly marked as either modified or unmodified.

5.2 Analysis

We claim that algorithm 5.2 is an optimal solution to the problem of determining which links of a set will be broken by a document modification. Our definition of optimality consists of the following two criteria.

First, an optimal broken link detection algorithm must compare any given node in the document tree at most once. Secondly, when determining whether a particular

Algorithm 5.1 Comparing a node

```
/*
Input: node is the node to be compared
Returns: true if node has been modified, false otherwise
Tree_Old (node) ≡ Tree_New (node) means to apply the equivalency operator to the
nodes in the same positions in Tree_Old and Tree_New
*/
if node is already marked then
  return marking of node
end if
set ismodified = Tree_Old (node) ≡ Tree_New (node)
if node is not modified then
  for all descendants of node do
    if Tree_Old (current_descendant) ≢ Tree_New (current_descendant) then
      set ismodified=true
      Break out of FOR ALL loop
    end if
  end for
end if
return (ismodified)
```

Algorithm 5.2 Determining which links have been broken

```
set modified = false
for all nodes in a post-order traversal of Tree_Links do
  if node_current is unmarked then
    use algorithm 5.1 to determine if node_current is modified
    if node_current is modified then
      set modified = true
      set temp = node_current
      while parent(temp) exists do
        mark temp as modified
        set temp = parent (temp)
      end while
    end if
    if not modified then
      mark node_current not modified
    end if
  end if
end for
```

Algorithm 5.3 place_node

```
/*inserts a new node new_node into subtree rooted at parent_node*/
for all children of parent_node do
  if new_node is a descendant of child_current then
    place_node (child_current, new_node) return
  end if
end for
insert new_node as first child of parent_node
for all siblings of newnode do
  if current_sibling is a descendant of new_node then
    Remove current_sibling from parent_node's child list
    Insert current_sibling into new_node's child list
  end if
end for
```

target is modified, an optimal broken link detection algorithm will compare at most **one** modified node in the subtree rooted at the target.

Algorithm 5.2 has the optimality properties described above. For the first property, we see that nodes are marked at the time they are compared. If a node is already marked, it will not be compared again (line 3 in algorithm 5.1).

For the second property, the post-order traversal of $Tree_{Links}$ guarantees that any links contained by *link* will have already been visited at the time *link* is visited. If any descendants of those contained links are modified, then *link* will have already been marked as modified as a result of executing the code in lines 7 through 11 of algorithm 5.2. If *link* does not contain any other links, then the break statement at line 10 of algorithm 5.1 guarantees that no other descendants of *link* will be visited after the first modified descendant is visted.

5.3 Repairing Broken Links

For each of the links marked broken in algorithm 5.2, we must find a node in the modified document tree that corresponds to the original target node. This problem is difficult if arbitrary document modifications are allowed. For example, if changes to the element structure of a linked-to node or changes to the actual content of the resource are made, it will not be possible to find an exact match in the modified document. However, in certain restricted circumstances, the problem is not as hard, and we can devise efficient algorithms to solve it.

Suppose a link L points to a node N_1 in document D_1. Editing D_1 in some way produces document D_2. If the following three conditions are met, then it is a fairly trivial matter to produce a new link L' which points to a node N_1' whose element structure and content are the same as the structure and content of N_1:

1. The element structure of N_1 is not changed
2. The content of N_1 is not changed
3. The XPointer for link L is not a spanning XPointer

One way of thinking of these restrictions is that the referenced sub-tree has been "cut" from its original location in the original document and "pasted" in its entirety into a new location in the modified document. Figure 5.3 shows examples of edits that do and do not meet satisfy these restrictions.

A simple brute force algorithm for finding N_1' would be to examine each node in D_2, comparing its content and element structure to that of N_1. However, given the

assumption that the element structure has not changed, many of the nodes of D_2 cannot be the node we are seeking. Only nodes N for which the sub-tree rooted at N has the same height as the node referenced by L could be the node we are looking for. Using this knowledge, we greatly reduce the number of nodes that are considered candidates for being the new target node.

We can further prune this set by comparing the element structure of the sub-tree rooted at each node in the set to the structure of the original target N_1. If the element structure is not the same, we can remove it from the set. Once we have pruned the set as much as possible, we can compare the element content of each of the nodes in the set to the content of the referenced resource, and return the set of nodes that match.

To implement this algorithm, we maintain a couple of data structures as the modified document D_2 is parsed. First, for each sub-tree height H we keep a list of nodes in D_2 for which the sub-tree height of the node is H. Second, an encoding of the sub-tree structure for each element is maintained to allow us to quickly compare the structure of two elements.

When we discover a broken link, we use algorithm 5.4 to search for a new target node in the modified document tree:

We can further optimize this algorithm by maintaining a pointer into each set of nodes indicating where in the set the last search completed. The next time this set is searched, the search will start at the pointer rather than at the beginning of the set. This is helpful if link targets that were close to each other in the original document

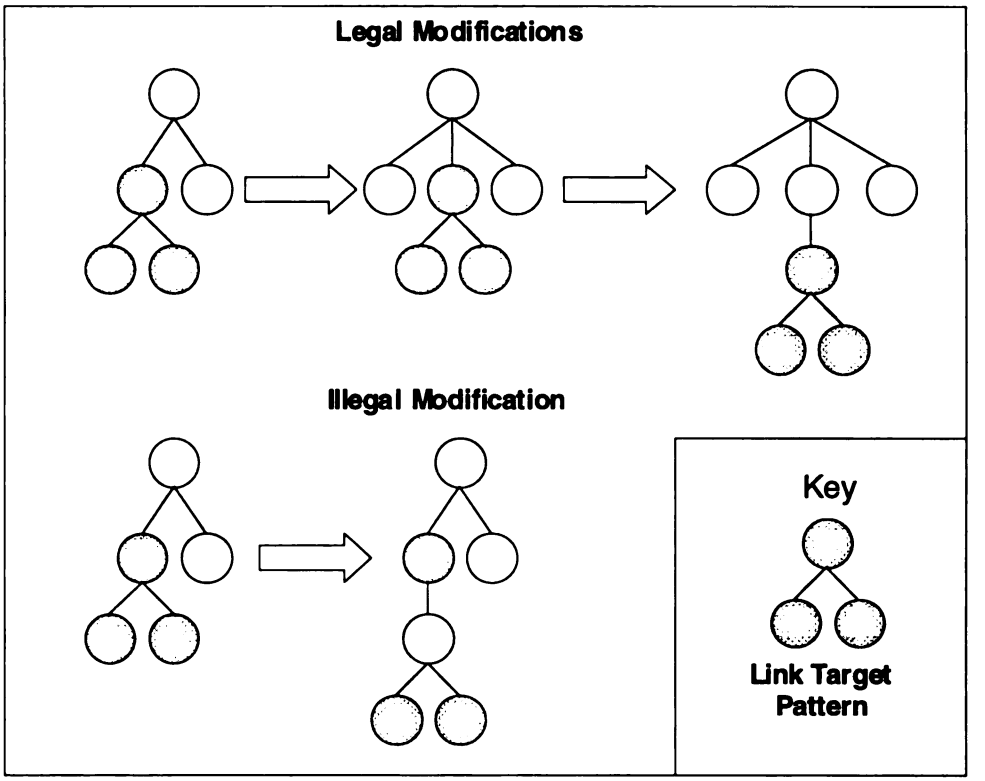


Figure 5.3: Types of document modifications that can and cannot be repaired

Algorithm 5.4 Finding a new target node

```
Let H = height of the original target node T
Set C = set of nodes in modified document tree of height H
For each node N in C
if E_S(N) = E_S(T) then
  if Content(N) = Content (T) then
    N is new target node; return locator
  end if
end if
```

tree remain close in the modified document tree. This will often be the case, such as when a common ancestor is moved from one location to another.

5.4 Evaluation and Testing

We have shown in section 5.1.2 that the broken link detection algorithm described in algorithm 5.2 is optimal. We now wish to evaluate the link repair algorithm given in algorithm 5.4.

5.4.1 Test Documents

Finding good documents to use as test cases was difficult due to the short life span of the XML and XPointer specifications. We chose to use two sets of documents for our testing. The first document set consists of a series of randomly generated documents. For the second document, we have chosen to use the XML version of the New Testament provided by Jon Bosak [45]

A major factor in the success of our link repair algorithm is the distribution of the heights of nodes in the tree. Suppose we determine that a link L_1 has indeed been broken and we wish to find the target of L_1 in the modified document. If a large majority of the nodes have the same height, we will have to search most of the tree in order to determine a new link to replace L_1. On the other hand, if the heights of the nodes in the tree are more evenly distributed, we will see considerable improvement in the performance of our algorithms.

5.4.2 Generating Random Documents

Since we decided to generate documents randomly, we needed to have some idea of what reasonable distributions for the variables described above are. Due to the lack of a large body of XML documents to use, we chose to examine HTML documents from our department and college's web sites and attempt to find a model that characterized the element height distribution from these sample documents. There were approximately 8800 documents in the collection that we looked at. For each of these documents, we calculated the percentage of nodes at each height, from 0 up to the height of the root node of the document.

Figure 5.4 shows the distribution of element heights obtained with outliers discarded. The data suggests that the distribution of element heights follows a hyperexponential distribution. The hyperexponential distribution is similar to the exponential distribution except that there is more area in the tails of the distribution.

Since HTML does not provide a method of pointing to arbitrary elements within the document, we decided a uniform distribution of link targets would be appropriate. That is, every element in the tree is equally likely to be chosen as a link target. Given that we have no empirical evidence to support any other model, this model seems reasonable.

The final task in using random documents as test cases is modeling changes to the documents. We allow two basic types of document edits: insertions and deletions. We have chosen to follow the model of [41] in defining these operations. Again, without any empirical evidence to guide us, we have chosen to treat each of these operations

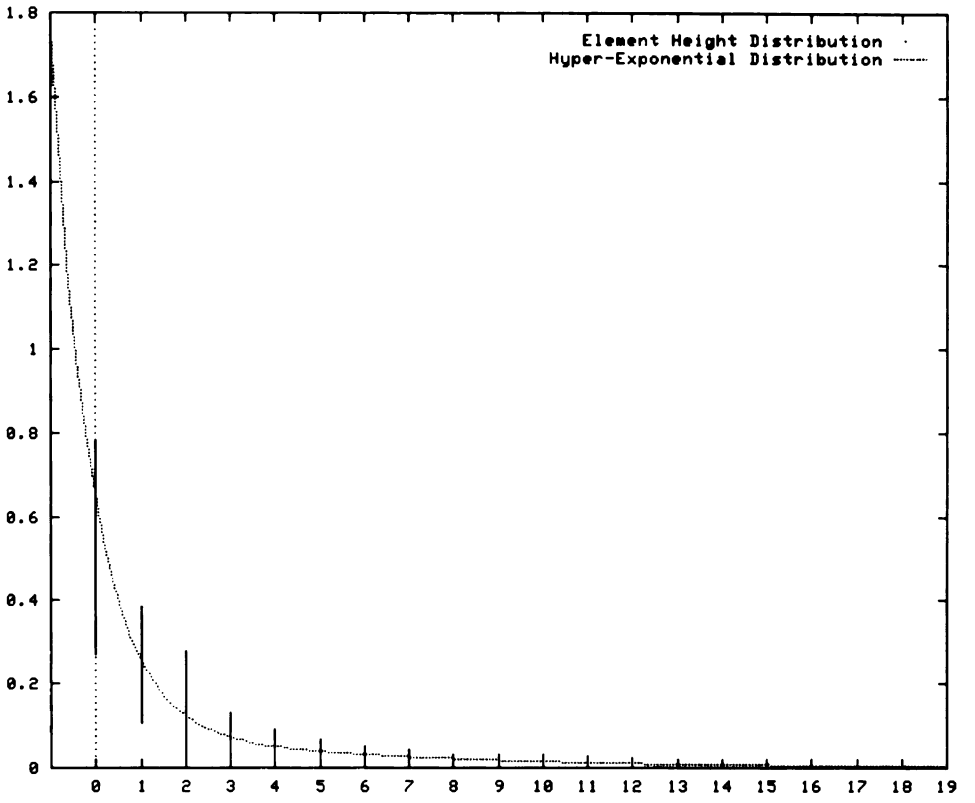


Figure 5.4: Element height distributions

as equally likely, and also consider each node in the tree equally likely to be the target of an editing operation.

5.4.3 The New Testament Document

This document is an XML marked up copy of the New Testament. It is approximately 1.1 MB in size, and contains close to 25,000 elements. For this document, rather than choosing links randomly, we have chosen to use a set of 270 links taken from the Life Application Bible [46] These links connect passages in the first four books of the New Testament that relate similar events. This set of links could very easily be found in a document whose goal is to show the commonalties between these first four books.

In order to come up with a modified document, we chose to rearrange the books in alphabetical order. The content of the document itself has not been changed.

5.4.4 Testing Environment

We have implemented the link repair algorithm described in section 5.3 and tested it. Due to the availability of tools such as XML parsers, the algorithms were implemented in the Java programming language. The current work seeks only to evaluate the effectiveness of our pruning algorithms. The algorithms may be implemented in a more efficient programming language as part of future work. Our implementation uses the SAX parser interface [47], and so can be used with any SAX-compliant parser.

We performed our evaluations on a Sun Ultra-2 with 2 300 MHz CPU's and 512 MB of RAM, running Solaris 2.6. Since measuring absolute performance is not our goal, the choice of hardware is not critical. This machine is used as a compute server, so it is difficult to know exactly what other activity was present at the time the tests were run. The machine usually has about 200 MB of free RAM at any given moment, however, so it is likely that our experiments had plenty of room in which to work.

5.4.5 Performance on the New Testament Document

After running some preliminary performance evaluations, we discovered that this set of links and documents is essentially a worst-case data set for input to our algorithms. This is due to the fact that the mark-up of the document follows a highly regular

pattern. Every verse is marked up in exactly the same way, as is every chapter and every book. All of the links in the link set point at leaf nodes. Each verse is located at exactly the same height in the parse tree, and possesses exactly the same encoding of the element structure. Furthermore, a disproportionate percentage (approximately 85%) of the nodes are leaf nodes.

To more accurately evaluate our algorithms, we generated random test documents in accordance with the distributions described in section 5.4.2. We chose to generate documents of three different sizes in terms of the number of elements in the document. The small document category was made up of documents that contain approximately 100 elements; the medium document category was made up of documents containing approximately 1000 elements, and the large document category was made up of documents containing approximately 5,000 elements. We randomly chose places in the generated documents to insert/delete elements, and varied the number of modifications made. We also varied the percentage of elements in the document that were the target of links from 5% to 50%.

Table 5.1 shows performance statistics for each of the test cases. These statistics are average values for all numbers of links. These results show that our algorithms achieve good performance for small documents, and become increasingly efficient as the size of the document increases.

Class	Number of Modification	Average Elements searched
Small	5	8
	10	9
Medium	5	62
	10	64
Large	5	188
	10	213

Table 5.1: Performance of Algorithm 5.4

5.4.6 Summary

In this chapter, we have described algorithms that can be used to efficiently detect link breakage when document modifications take place. In some circumstances the algorithms are able to repair the broken links quickly by searching the modified document for the referenced content and calculating a new link. An advantage of the algorithms is that they do not require authors to provide explicit information about how a document has been modified. To our knowledge, these algorithms are the only solutions that can repair broken links automatically.

We have shown that the broken link detection algorithm is optimally efficient according to the conditions we set forth. These optimality conditions are reasonable for any implementable (i.e. non-oracle) algorithm. Our link repair algorithms have shown that by restricting the search for new target nodes based on the target sub-tree height and element structure, we can significantly reduce the number of nodes in the modified document that must be searched to find the target.

In chapter 6, we discuss the design and implementation of a software system that utilizes the algorithms described in this section. Using this system, we can achieve our goal of providing link integrity automatically.

Chapter 6

A Software Agent to Provide Link Integrity

6.1 Introduction

In chapter 5, we described algorithms that can be used to compute a new locator efficiently to repair a contextually broken link. In this chapter, we describe a web proxy service utilizing those algorithms to ensure that, at all times, a user agent receives the correct data as the result of following a hyperlink.

In order to provide this guarantee, our system makes use of an agent, which is a particular implementation of the general WWW idea of a proxy server [48]. Each web server wishing to ensure referential integrity of links pointing into its document space must be serviced by an agent. Agents communicate through a protocol we have developed, allowing them to notify each other when links have been created, deleted or broken.

An outline of the chapter is as follows. First, we describe the requirements for the agent and how we have addressed them in our prototype. Next we describe the agent communication protocol in detail, and describe its operation in the presence of unreliable communication networks. We then analyze the resource requirements and performance effects of adding our system to the current web infrastructure. Finally, we summarize the chapter and give conclusions.

The necessity of inter-agent communication is motivated by Mendelzon and Milo [26], in which the authors give a proof for why it is not possible to compute the answer to the query "Are there any documents pointing to document D" (or equivalently, "Find all the documents pointing to document D"). In order to guarantee 100% link consistency, it is necessary to require some amount of cooperation among agents.

6.2 Overview of the Agent Proxy

For clarity's sake, we define some terminology. Let S be a web server, addressable using the URL U. We say that S's *document space* is the set of all documents whose URL is of the form U/path, where path is a string locating a document on server S. The agent has two distinct functions. The first is its role as an Hypertext Transfer Protocol (HTTP) proxy. In this role, it must accept incoming HTTP requests, forward them to the server for which they are destined and relay the response from the destination server back to the user agent (typically a browser) that initiated the request.

The agent's second role is to maintain a database with information about all links that point into the web server's document space, and make sure that incoming requests following those links locate the originally designated part of the document. The agent is able to perform this role because in its role as a proxy, it will receive and examine all requests made to servers it is responsible for, allowing it to redirect requests generated by following a link the agent knows to be broken.

In order to allow the agents to work with previously existing web servers, the agent must be configured to listen on the same IP address and Transmission Control Protocol (TCP) port the web server used. Of course, this means that the address and port number of the server must be changed. Since the agent is acting as a proxy, to the outside world, the change of addresses will be transparent.

In our current implementation, there are actually two processes involved in implementing the functionality described above. One we call the *lightweight agent*, while the other is the agent proper. The lightweight agent fulfills the role of the HTTP proxy, and the agent is responsible for the link maintenance functions.

Agents and lightweight agents communicate with each other through the use of the Java Remote Method Invocation (RMI) distributed processing architecture [49, 50]. While it is necessary to have one lightweight agent for each web server (since each server requires a unique IP address and TCP port) an agent can actually service multiple web servers.

The lightweight agent receives HTTP and document maintenance requests, and forwards the document maintenance requests to the agent process responsible for that server. In addition to the request, the lightweight agent includes information about

which server the request pertains to so that the agent can act accordingly. Figure 6.1 gives an illustration of the system architecture.

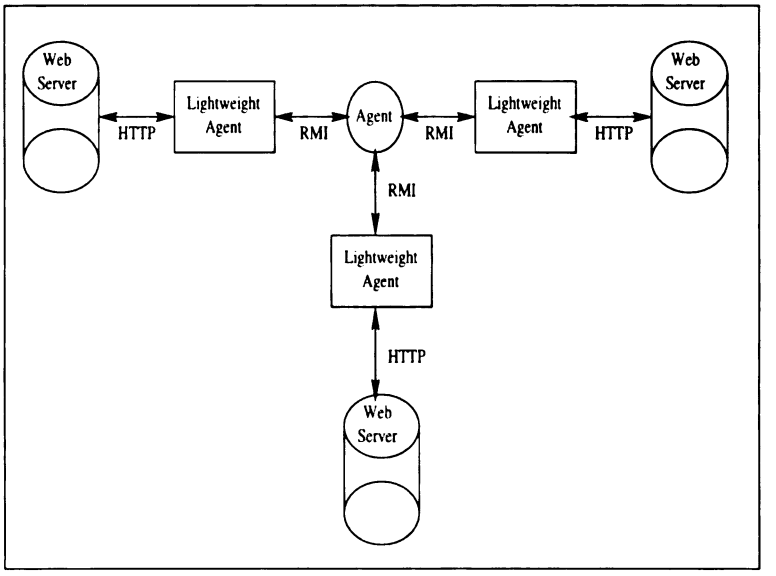


Figure 6.1: System architecture for agent, lightweight agent and web servers

6.2.1 The Agent in Detail

The agent maintains a database of information about each document for which it is responsible. For each document, the database contains a list of documents (URL's) that contain references to the document in the database. Each entry in the database also records the specific locator (series of XPointer location terms) the referencing link is using.

Figure 6.2 gives an illustration of some of the messages used by agents to communicate with each other. In the figure, we have two web servers S_1 and S_2, and two agents $Agent_A$ and $Agent_B$. $Agent_A$ is responsible for server S_1, while $Agent_B$ is responsible

for server S_2. Further suppose a document D_2, to be located on server S_2, is created and that D_2 contains a link L_2 pointing to some location in document D_1, located on server S_1. In order to guarantee the contextual integrity of L_2, $Agent_B$ must send a message to $Agent_A$, indicating the URL of document D_2, and the locator information of L_2.

Suppose document D_1 is now updated. We assume the entity performing the update informs $Agent_A$ of its desire to perform the update and asks $Agent_A$ to perform the update on its behalf. $Agent_A$ must determine which of the links pointing into D_1 will be broken by the update. Furthermore, for each link that is broken, $Agent_A$ must compute a corrected locator and notify $Agent_B$ of the new locator.

We must not make any assumptions about the actual amount of time it takes $Agent_A$ to contact $Agent_B$ and notify it of the change.[1] Therefore, there will be a period of time when D_2 will contain the old (incorrect) locator. During this time, $Agent_A$ maintains a **link map** containing a mapping that indicates the corrected locator that should be mapped to the broken locator. This mapping is indexed by both the combination of the referring document's URL and the locator. During the (perhaps unbounded) period of time until $Agent_B$ fixes the broken link in D_2, $Agent_A$ will intercept all HTTP requests which have a *referer*[2] header of D_2 and the incorrect locator and cause them to use the new locator information. $Agent_A$ does this by sending an HTTP redirect response (status code 302) with the correct locator to the

[1] In fact, it may be the case that $Agent_A$ chooses to delay notifying $Agent_B$ for performance reasons.

[2] This unfortunate misspelling is in fact part of the HTTP standard.

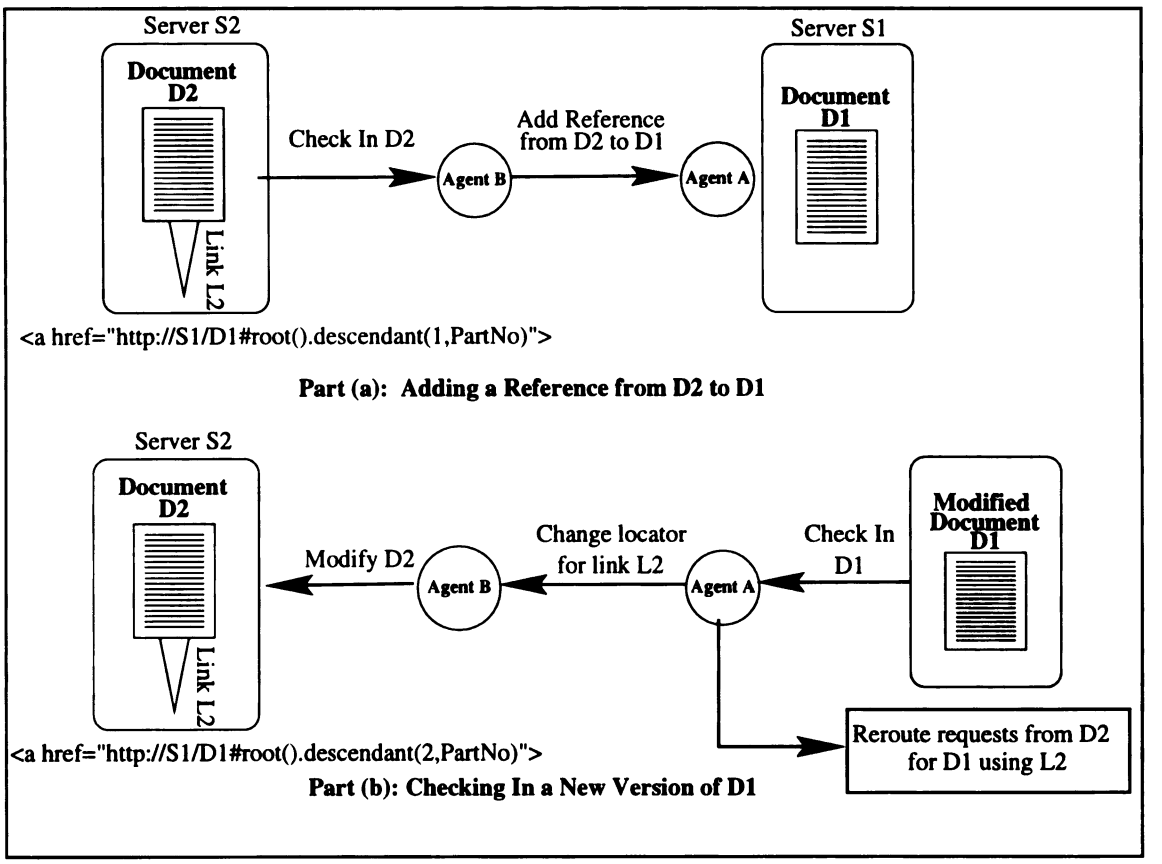


Figure 6.2: Messages passed between agents

client that made the incorrect request. The client will then re-issue its request, using the correct locator, and will receive the correct information.

The requirement of knowledge of the locator means that for our system to work with current generation browsers, we cannot use the traditional '#' character as the separator between the URL and the fragment identifier. This is due to the fact that browsers do not send along the fragment identifier with the request. This is unfortunate, but hopefully a case can be made that the browser should send the fragment identifier along, perhaps as a separate header, in extensions to HTTP.

As implied in the preceding description, part of the agent's role is to serve as a gateway for document modifications. When a document author wishes to make modifications to that document, she must send a request to the agent containing the new content of the document. The agent will then make the change when it has determined which of the links pointing at the document will be broken and how to correct them. We are explicitly **not** requiring that the authoring software (or the author, for that matter) maintain a record of the changes that have been made to the document; it simply sends the modified document data to the agent. While this policy may generate more network traffic than sending only the change description, it places less burden on the authoring software. By utilizing a separate "check-in" program, any currently existing authoring software can be used. The authoring software simply works on a local copy of the document in question, and then the user invokes the check-in program when editing is complete.

6.3 Protocol Description

In this section, we describe the protocol used by agents to communicate with each other. We first describe the set of messages that are used, and discuss the normal behavior of the agent when a broken link is followed. We then examine issues having to do with network failures and delays.

6.3.1 Messages

AddServer (LWAAddress, LWAPort, WebServerAddress, WebServerPort)

This messages indicates to an agent that it should begin servicing the web server whose lightweight agent is located at *LWAAddress* and listening on *LWAPort*. It also informs the agent of the actual address and port of the web server that is being serviced This is necessary so that the agent can communicate with the web server to update web content using the HTTP PUT method or some similar mechanism if the agent and the web server do not share a common file system.

RemoveServer (ServerAddress, Port)

Indicates to the agent that it should no longer handle protocol messages regarding the lightweight agent located at *ServerAddress* and listening on port *Port*.

AddReference

(ReferencedURL, ReferringURL, Locator, [Checksum or LastModified-Date])

This message indicates to the receiving agent that a link has been created that points into its document space. The document containing the link is located at *ReferringURL* and uses locator *Locator*. The purpose of the remaining parameters are described in section 6.3.3.

RemoveReference (ReferencedURL, ReferringURL, Locator)

Receipt of this message by a link agent means that a previously registered link has been removed and should be removed from the agent's link database.

CheckInURL (DocumentURL, SearchList, DocumentSize, DocumentData)

This message is sent whenever either a new document with URL *DocumentURL* has been created, or the document has been edited. *SearchList* is a list of document URL's indicating a set of documents that should be used by the algorithms described in chapter 5 to perform broken link detection and repair. *DocumentURL* is always an implicit member of *SearchList.* This feature is useful for the (fairly common) editing operation of splitting a large document into several component documents.

LinkUpdate

(ReferencedURL, ReferrerList, NewLocator,NewURL, NewLocator)

This message tells the receiving agent that all links contained in documents listed in *ReferrerList* pointing to *ReferencedURL* using locator *Locator* should be updated to point to *NewURL* and use *NewLocator.*

Allowing a list of referring documents to be passed reduces the amount of network traffic and processing that must occur if the receiving agent has several documents in its document space that use the same address for a link.

The majority of the messages generated by the protocol come about due to the receipt of a CheckInURL message by an agent. The agent must check the new version of the document to determine if any new links exist in its content, and if so send AddReference messages to the appropriate agents. If the document is a new version of an existing document, it is also possible that links have been deleted in the new version of the document, in which case a RemoveReference message should be generated.

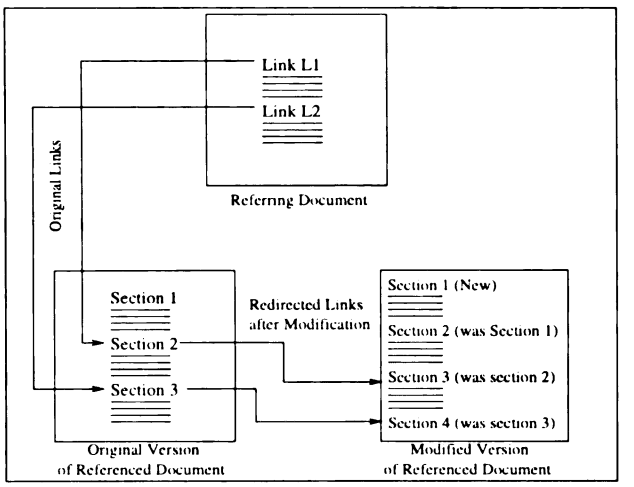


Figure 6.3: Possible ambiguity when sending a redirect message

If the agent detects that some links have been broken, it generates the appropriate LinkUpdate messages and sends them to the agents responsible for the documents containing the broken references.

6.3.2 Deciding When to Redirect

When the lightweight agent determines that a broken link has been followed, it sends an HTTP redirect message to the user agent that followed the link, telling it the correct locator to use. For correct operation in certain situations, the lightweight must be careful about how it goes about doing that. Take for example, the situation shown in figure 6.3.

Here we see a referring document that contains two links into the referenced document. After the document is modified, we see that the locator originally used by link L_1 is now mapped to the locator originally used by L_2.

When the agent receives a request following link L_1, it will respond with a redirect asking the client to follow L_2 instead. When the agent receives this request, it should NOT be redirected. Unfortunately, the agent has no way to distinguish this request from a request that is actually following L_2. We must be able to determine the relationship between two requests – that is, did the request to follow L_2 come about as a result of a previous request following L_1? Unfortunately, due to the stateless nature of the HTTP protocol, this is impossible. While HTTP 1.1 contains a mechanism for persistent connections (that is, multiple HTTP requests from a client to a server can be sent along the same connection), either party is free to terminate the connection at any time. Therefore we cannot force both requests to be made over the same connection, which would allow us to determine causality.

The Cookie [51] mechanism allows a server to introduce state into HTTP requests and responses. By utilizing cookies, we can solve the problem described above in the following manner. When the link agent receives the request following link L_1, it includes a Set-Cookie header. The Path field of this cookie specifies the referenced document's URL, the name of the cookie is "NoRedirect" and the value of the cookie is set to the value of L_2. Finally, the age of the cookie is set to be 1 second. Setting the age in this way makes it likely that the cookie is only valid for one request by the user agent.

If the link agent receives a request containing a cookie of the form described above, it knows this request should not be redirected, and so forwards the request unmodified to the web server. The link agent forwards the response from the server to the user

agent, including a Set-Cookie header that invalidates the cookie generated by the original request.

Creating an association between two HTTP requests using cookies is the preferred mechanism to ensure that redirected requests are not subsequently redirected, and is the only completely correct means that we can do so. Unfortunately, due to the potential for misuse of the information conveyed using cookies [52], some users direct their user agents not to accept cookies. If we desire, we can still do quite well without using cookies. Let us assume that we have sent a Set-Cookie request to a browser, and it has rejected this cookie[3]. We still need some way to make sure the second request is not redirected.

The lightweight agent must redirect the original request in such a way that it can identify the subsequent request and refrain from performing redirection on that request. The only part of the second request the agent has control over is the Request-URL. The agent has a couple of options in constructing the redirected URL:

- redirect the request to a different port number
- redirect the request using a different HTTP Host header
- change the path to the requested resource

Unfortunately, any of these methods require sending a different URL back to the user agent that made the original request. If the user then views the resulting document

[3]Determining if a cookie is rejected requires two redirections: first to a URI that is specifically set up to examine if a cookie is present, and then the actual redirection to the appropriate site, depending on whether the cookie is present.

and decides to create a new link to a portion of it, the URL the user sees for the document will be in the form for which redirection is restricted from occurring. This would then prevent the system from maintaining the integrity of this new link.

We must deal with this problem at a higher level than HTTP, in the link agent communication protocol. If a link agent receives an AddReference message using the restricted-redirection form of a URL, it must nack (send a negative acknowledgment for) that message, including the correct version of the URL as part of the message. By sending this nack, we ensure that links using the restricted-redirection form of the URL are not created. This of course does not prevent incorrect links from being created without an AddReference message being sent.

6.3.3 Operation in the Presence of Network Failures

In the absence of network delays or failures, the protocol described in the previous section is sufficient to guarantee link integrity. However, communication networks are not perfect, and so we must address the issue of network failures and delays. The presence of caches along the path between web clients and servers further complicates the situation. In this section, we address problems caused by unreliable networks and the presence of caches.

Lost or Delayed AddReference Messages

Suppose an AddReference message is delayed for a significant period of time.[4] It is possible for the URL being referenced to be modified during the time the AddReference message is being transmitted. This modification could possibly change the content pointed to by the locator used by the referencing document. In this case, not only will the link will be broken initially, the agent protocol will perpetuate the broken link as further changes to the document are made! Figure 6.4 illustrates this problem. In order to avoid this problem, it is necessary to know either the exact content the user wishes to link to, or the last modification date of the referenced document, **at the time that the user downloaded it.**

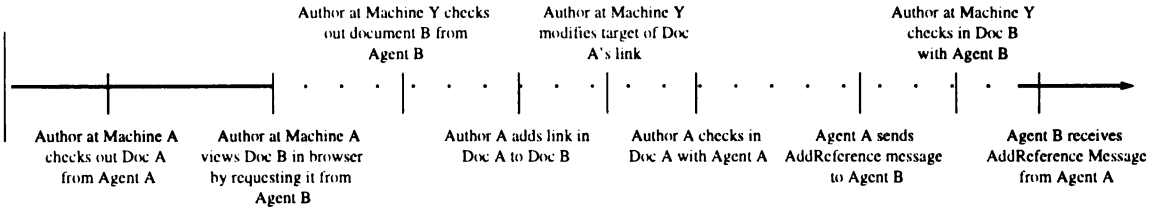


Figure 6.4: Problems due to network latency and AddReference messages

One possible way of avoiding this problem using knowledge of the intended content is to compute a checksum on the data the author intends the link to reference. This checksum is then included as part of the AddReference message. On receipt of the message, the receiving agent retrieves the current version of the document and makes sure the checksum on the intended content matches the one in the message. This might be implemented by providing a simple program into which the user pastes the

[4]In fact, since network delivery takes non-zero time, this situation can occur no matter how long it takes for the message to be delivered.

portion of the referenced document and have this program calculate the checksum. The check-in program later prompts the user for the checksum. Unfortunately, this adds a significant amount of complexity to the check-in process, and would be very tedious when checking in a document with a large number of links.

Another way of attacking this problem involves the use of the last modification date of the referenced document. A somewhat naive approach is to timestamp the AddReference message with the current date and time when it is sent. The receiving agent compares the timestamp in the message with the last modification date of the document, and if the modification date is later than the timestamp in the message, a nack is sent back, warning the user that the link might now be incorrect. There are two significant problems with this approach. First, using the time the AddReference message is sent as the reference time is not necessarily an accurate measurement of the time the user examined the content of the document and decided to create a link. Thus, it is possible that the protocol would accept a message containing a locator that references content other than that desired by the author.

A second problem with the naive approach is its reliance on synchronization of separate clocks. If the clock on the author's machine and the agent's clock are not synchronized, it is again possible for the agent to accept a message containing an incorrect locator. Algorithms for determining temporal relationships between machines with unsynchronized clocks, such as Lamport's Algorithm [53], are not helpful in this situation, since as can be seen in figure 6.4, there is no communication between the machines sending the AddReference and the CheckInURL messages, and so no reasoning can be made about the order of events. While HTTP 1.1 urges using a

method such as the Network Time Protocol (NTP) [54] to synchronize clocks, the standard does not require it. Therefore, requiring synchronization of physical clocks is too significant a restriction to impose on the potentially globally distributed set of agents.

Thus, we need a system that does not require synchronization of clocks or significant user input. If the last modification date of the referenced URL at the time the user decided to create the link could be known, then this could be included as part of the AddReference message. Since this date is generated by the server, and the Agent receiving the AddReference message can request this information from the server, we can use it as a basis for comparison.[5] The question is, how does the check-in process know what the modification date of a referenced document was at the time that the user looked at it?

We could simply require that the user supply this information. Most current browsers provide a way to retrieve the HTTP LastModified header when viewing a document. While this is less obtrusive than requiring the user to create a checksum, it still is user intervention we would like to avoid.

Another possibility is illustrated in figure 6.5. In this situation, we have placed a proxy on the client side. The client is configured to use this proxy, and thus all outgoing HTTP requests are sent through the proxy. The proxy is then able to record the requests made by a client machine, recording the URL's and last modification date

[5]For robustness, a web server is often replicated. In order for this scheme to work, the clocks of a web server and any machines replicating it would need to be kept synchronized. This is not as unrealistic a requirement as keeping the clocks of machines administered by different entities synchronized.

for each document the user requests. When the check in process is initiated, the agent communicates with the proxy[6] to find out the last modification dates of all the documents being referenced by the new version of the document and includes these as part of the AddReference messages it sends.

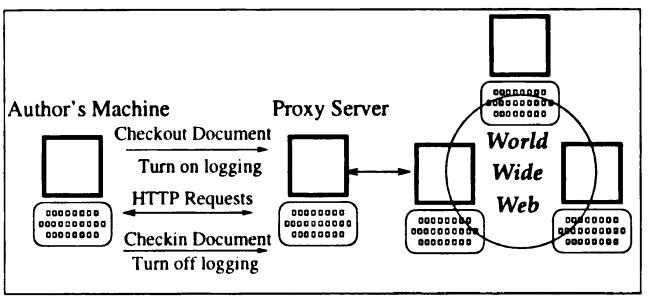


Figure 6.5: The operation of the client proxy

Logging all requests by a client in the proxy uses a significant amount of storage, and it is difficult to know when to flush this data. Therefore, we need some way of knowing when to turn the proxy recording mechanism on and off. One way of doing so would be to incorporate this action into the check-out and check-in procedures. When the user checks out a document, we turn the recording mechanism on, leaving it on until the time the user checks in the document. Logging of all requests is performed by the proxy during the time between check in and check out.

Another option would be to provide "begin authoring" and "end authoring" programs that the user executes. This avoids the problem using check-in and check-out where the user might do some browsing between checking in of a first document and checking out of a second document.

[6]in fact, the agent could very well **be** the proxy.

Lost or Delayed LinkUpdate Messages

A second situation we must consider occurs when multiple LinkUpdate messages are sent to an agent, and the order the updates are received is not the same as the order in which they were sent. We cannot rely upon the in-order delivery guarantees of TCP here because the messages will be sent across different TCP connections, and TCP only guarantees in-order delivery of messages on the same connection.

The example below outlines why receiving link update messages out of order could be a significant problem:

1) Document D_1 links to document D_2 using locator L_1

2) D_2 is modified by a check-in with Agent A_2, transforming L_1 into L_2

3) Agent A_2 sends a message M_1 to Agent A_1, asking that it replace L_1 with L_2. This message is delayed indefinitely.

4) D_2 is modified again, in a way that causes L_2 to be replaced by $L3$.

5) A_2 sends message M_2 to A_1, asking it to replace L_2 with L_3.

If M_2 is received before M_1, this would confuse A_1; D_1 may not hold any links using L_2 as a locator.

This situation could not occur if updates are not permitted while there are unacknowledged LinkUpdate messages. Then, we will be sure that LinkUpdate messages will be delivered in the correct order, since only one set of messages can be in transit at any one time. This solution is not practical since we have no idea how long it might take for any given LinkUpdate message to be received and acknowledged. Instead, our solution buffers unacknowledged LinkUpdate messages at the sender. Every time a

LinkUpdate message is sent, it includes ALL unacknowledged LinkUpdates intended for the receiver. This is not an extra buffering burden for the sender, since this data is already being maintained as part of the link map stored by the lightweight agent. The sender must include some sort of sequencing mechanism in its messages, and the receiving agent must know which messages it has already received. For the sequencing mechanism, we use the date/time of the sending agent. Since the receiver will be using the date contained in the message as part of the acknowledgment, and specifically not the value of its own clock, there is no issue of coordinated clocks.[7] The receiver only needs to maintain in its memory the sender's timestamp for the last message successfully received from that sender. Using the date as the sequencing mechanism avoids the complexity involved in avoiding duplicate sequence numbers in the event of a machine failure.

If the receiver receives a message containing LinkUpdate information it has already processed, it disregards that portion of the message. When the receiver sends an acknowledgment to the sender, that acknowledgment acknowledges receipt of all outstanding LinkUpdate messages sent before the timestamp contained in the acknowledgment message.

Browser and Proxy Caches

The presence of caches in the network further complicates the agent communication protocol. Suppose a LinkUpdate message is sent to an agent. The agent receives the

[7]Again, replicated agents used for performance and robustness **would** need to keep their clocks synchronized.

message, corrects the broken link indicated in the message, and sends an acknowledgment to the sending link agent. In the absence of caches, the lightweight agent for the referenced document could immediately remove the mapping of the old locator to the new locator from its link map, confident that no request using the old locator will be received in the future[8].

However, when the referring document can be retrieved from one of several caches, the situation becomes more complex. How does the the receiving lightweight agent know whether or not the request should be redirected? And how does it know when the mapping can be removed from its link map?

To answer these questions, the links agent sets the "Expires" header field of any response that is sent by the link agent to a user agent according to the following formula:

$$Expires := min(\text{Max-Expires, Server's Expires value})$$

where Max-Expires is a constant set as part of the lightweight agent's configuration. When an agent acknowledges the receipt of a LinkUpdate message, it includes its value of the Max-Expires parameter as part of the acknowledgment message.

By setting the Expires header field in this way, a lightweight agent knows the maximum amount of time that a cached copy of a referring document containing a broken link can exist. It must keep the mapping for that broken link in its link map for that period of time. If during that time it receives a request following the broken link, it sends back an HTTP conflict response (response code 409) to the user agent,

[8]unless, of course, a new link is created using the same locator, in which case requests using that locator should of course not be redirected

notifying the user that the link is potentially broken and the document should be explicitly reloaded from the origin server before attempting to follow the link again. To make sure that the request made by the user after doing so does not again receive a 409 response, the link agent again makes use of a cookie.

6.4 Costs

As with any modification that adds functionality to a system, we must determine what additional resource requirements will be imposed in adding the new functionality. In this section, we first examine the resource requirements for the agent, and then discuss how the performance of the web is affected by the addition of the agent service.

Memory Requirements

There are two data structures the agent must maintain in memory. The first of these data structures are the link databases for each document in the document spaces of the servers serviced by the agent. Although these database structures could be maintained in main memory, it is more likely they would be contained in some sort of mass storage system. These databases must be maintained in persistent storage so that the agent can recover if the agent process crashes for some reason. Also, they only need to be consulted during a document check-in, a relatively infrequent occurrence, so speed of access is not of particular interest.

The second data structure is the link map the lightweight agent uses to map broken links to their corrected counterparts. This structure is much smaller than the link

databases, and will be referenced more frequently, since each time a link containing an XPointer is referenced, the agent must verify that the link is correct. Therefore, it seems likely that this structure would be stored in main memory. How large might this structure get, and what may be done to limit its size? There are several factors that contribute to its size:

- the frequency that document modifications are made causing links to break.
- the policy used to decide when to request that the agents for referring documents, whose links have been broken, update their documents. The immediate notification policy sends a LinkUpdate message as soon as the broken link is detected. Delayed notification policies such as periodic notification and notification on reference are also possible.
- if a delayed notification policy is chosen, further variables are present. If periodic notification is chosen, then the period must be chosen. If when-referenced notification is chosen, the amount of time between references to a link becomes important.

Even with an immediate notification policy, we cannot determine a bound on the size of the link map, because it is possible that we may not be able to contact the agent that needs to be notified. Even so, it is likely that the amount of memory the link map requires will be reasonably small.

If we assume the average length of a locator is 32 bytes, each entry in the link map structure will consist of 64 bytes (since each entry consists of a mapping from one locator to another). There may be some additional overhead involved in the link map

(for example, one way of implementing it would be as a hashtable of hashtables – the first level of hashtable is indexed by a referrer/referred pair, and each hashtable in the second level contains the locator mappings), but this overhead should be fairly negligible. Allocating 1 MB of RAM for the link map structure then allows storage of mappings for about 16,000 broken links.

The notify on reference policy is desirable since it has the potential to reduce network traffic generated by unnecessary update notifications. However, it is also likely to be the policy that allows the link map to become the largest. A simple modification to this policy where notification messages are sent when the structure becomes too large would be a reasonable compromise.

Effects on Web Performance

In order for link requests to be serviced correctly, we must funnel all HTTP requests through the lightweight agent responsible for the server that owns the document. If clients are allowed to communicate directly with the web server, the server will not be able to determine if the link being requested is broken, since only the agent possesses this knowledge.

If the agent process is running on the same machine as the web server, then the cost of requiring the agent process will simply be that of communicating between the two processes. However, it is likely that the two processes will run on separate machines for performance reasons. In this case, every HTTP request to the server will require

at least one extra network hop. This performance cost is typical when there is a proxy between a web server and the rest of the network.

If a broken link is followed, the browser will be redirected to the correct location, which will essentially be an extra round trip between the browser and the server. Again, this cost is due to the use of the HTTP redirect mechanism, a mechanism already in place and generally used.

In general, the costs of adding the agent proxy are minor, and are already present to a great extent in the current web architecture. Given all the processing that goes on in a web server today, the cost of processing the extra network traffic generated will be negligible.

A side effect of requiring the agent is perhaps more costly. This side effect is the fact that server replication is not possible if all of the requests must go through the agent. It is becoming more and more common to see a single logical web server be serviced by multiple physical servers, with the content on these servers replicated. The load on these servers can be distributed through methods such as DNS Aliasing [55], Magic Routers [56], and HTTP redirect [48].

Forcing web clients to communicate with a single agent introduces a single point of failure into the system, and eliminates the load balancing advantages of having replicated servers. The simplest way to deal with this problem it is to replicate the agents. With this approach, the same methods used to distribute the load between conventional web servers can be used to distribute the load between the agent processes. If this approach is adopted, steps will need to be taken to assure that the link databases

and link maps will be kept synchronized between the agent processes. We intend to explore this approach more carefully in future work.

6.4.1 Increasing Document Processing Efficiency

An agent servicing a frequently updated web server will need to process a large number of documents concurrently. Most implementations of XML processors are written in a single-threaded fashion. Since a good portion of the time spent processing an XML document is spent doing I/O (especially for large documents), a large percentage of the time it takes a single-threaded processor to process a document will be spent in a blocked state. With an appropriate threads library a multi-threaded, single process XML processor will be able to process documents much more efficiently than its single-threaded counterpart.

We have modified version 2.0.15 of the XML4J processor from IBM [57] to transform it from a single-threaded to a multi-threaded implementation. As can be seen from 6.4.1, the time that it takes to process a set of large documents is considerably improved by adding more threads.

We have actually modified the XML4J processor in a much more significant way than to simply add multi-threading capabilities. The changes we have made allow the processor to be used in a distributed fashion. Multiple processes running on distributed machines are then used cooperatively to process the same document. Implementing the system in this way allows resources such as memory to be aggregated to facilitate processing of large documents that perhaps a single machine might not be able to

handle. It is this implementation, running on a single machine with the number of processing threads varied, that the figures in 6.4.1 refer to.

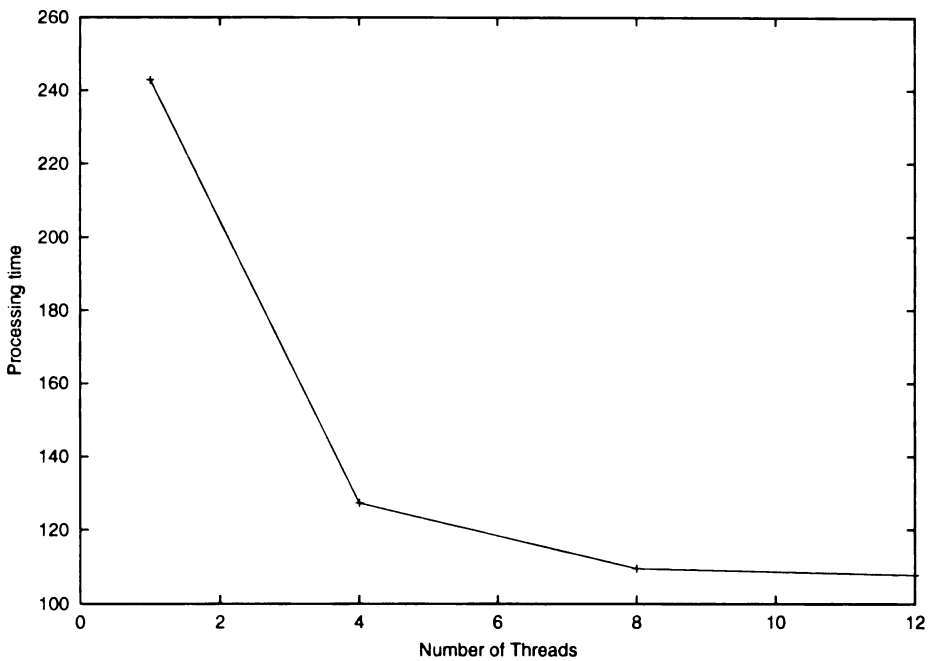


Figure 6.6: Performance of modified parser using multiple threads

The distributed implementation works in the following way. When a client wishes to have a document parsed, it submits a request to the processing cluster by making a remote procedure call to one of the members of the cluster, specifying the URL of the document to be parsed. This processor parses the root node of the document, storing the text nodes of the document tree locally. As it processes the root, it keeps track of the byte offsets of any children that it encounters. When the root node is completed, the processor picks another processor from the cluster and sends it a message, asking it to process the children of the root node.[9] The processors recursively

[9]Originally, the processor would allocate each child as it was encountered to another processor. Although this allows a greater degree of parallelism, due to the relatively high costs of remote procedure calls, this was found to be impractical.

continue this process, treating the node currently being parsed as the root node of the document, until the document has been completely parsed. Figure 6.7 illustrates this architecture.

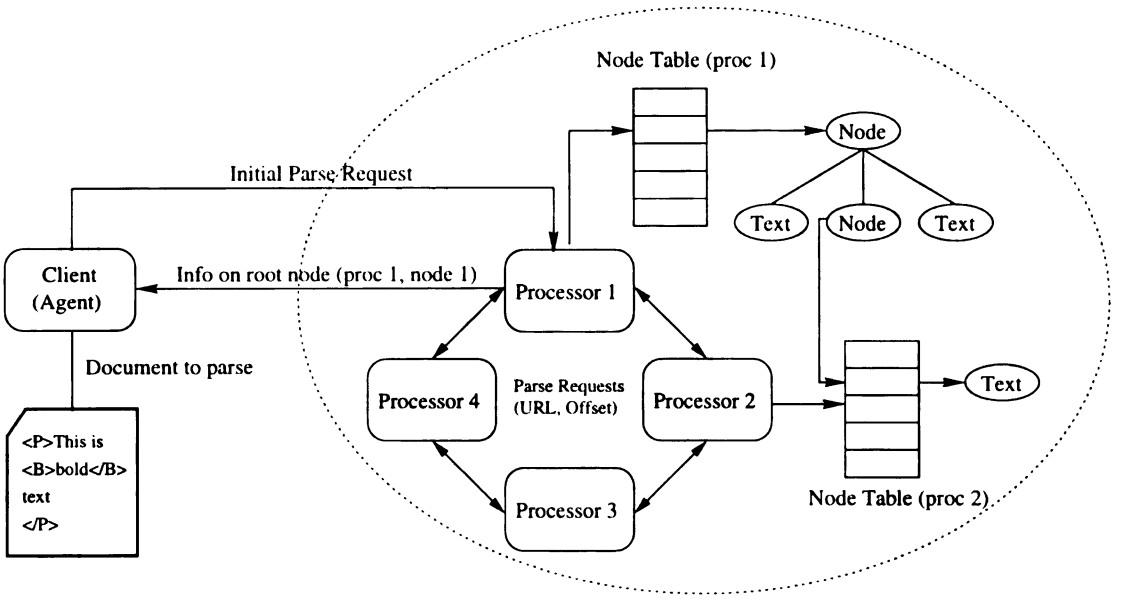


Figure 6.7: Distributed parsing architecture

The client interacts with the processor cluster through the standard DOM API [44]. Each time a request is made for data associated with a node, a remote procedure call is made to the machine that holds that node. It is possible that a request for information about a node could be made before that node has been processed. In that case, the processor waits to respond to the client until the requested data arrives. List structures such as child lists contain a list of references to remote nodes, which are used to contact the processors which actually hold the data associated with the node.

For reasons we have been unable to determine, however, the performance of the distributed system is unacceptably poor to be practically useful in comparison to

the single processor model. We believe that the potential improvements for the distributed system merit further study of the problems we have found in our particular implementation.

6.5 Prototype Implementation

We have implemented a prototype agent which works with current web browsers and servers without modifications to them. The only thing we have had to do that deviates from standards is to separate the URL portion of a link element from the XPointer portion using a | character, rather than the standard # character. This is because current browsers do not send anything after the # sign as part of the request; for our agent to work, it must know the locator the requester is using.

The prototype agent and the link repair algorithms that it uses were both implemented in the Java programming language. Obviously, one of the main benefits of using Java is that this allows the agent process to execute on any platform, so long as there is an implementation of the Java Virtual Machine for that platform.

Since there are not yet any browsers that implement the XLink and XPointer proposed standards, our implementation does all of the XPointer processing at the agent. When a link containing an XPointer is followed, the agent retrieves the portion of the document located by the link and returns only that portion to the requester, rather than returning the entire document and allowing the browser to process the XPointer.

We have created a simple message editor to facilitate the creation and communication of link notification and "check" in messages. A simple "check in" program that checks in a document located in a file has also been written as part of our prototype.

6.6 Summary

This chapter has described an agent acting as a proxy for the web server(s) that wish to guarantee contextual link integrity. For guaranteed correctness, the agents for each web server must communicate with each other when links are created or documents are modified.

The costs of implementing such a system have been shown to be comparable to those of systems that are already in use, such as firewalls and employing HTTP redirect to keep traditional links from breaking.

A prototype implementation has been developed using current web tools, including web browsers and servers. The fact that we were able to implement the prototype without changes to current browser and server software demonstrates the potential it has for quick deployment.

In the future, we plan to explore the issues of fault tolerance and performance discussed in section 6.4. Allowing agents to be replicated would greatly increase the usefulness of the system. We would also like to continue to reduce the amount of cooperation needed between agent processes, which we discuss in chapter 7.

Chapter 7

Independent Operation

7.1 Overview

We now turn to work which has as its goal eliminating the need for cooperation between authors of documents in web sites administered by different entities. In particular we are interested in eliminating the *link notification* requirement. This requirement states that authors must notify the agent associated with a web server every time a link pointing to a document in that server's document space is created or deleted. Eliminating this requirement is desirable since it will expand the set of documents for which contextual link integrity can be maintained.

As we stated in section 6.1, the requirement for agents to communicate the creation of links with each other exists since it is not possible to enumerate the document space of the WWW. While we cannot exhaustively search the universe of web documents to discover the existence of links, information is available that can inform an agent of

the existence of a link. This information is provided in the form of the HTTP*referer* header.

By monitoring the receipt of HTTP requests containing referrer information, we can discover the existence of new links and record them in the agent's link database. We will call links that are recorded in an agent's link database due to this process *discovered links*.

This method of discovery is not foolproof, however. Consider the example shown in figure 7.1.

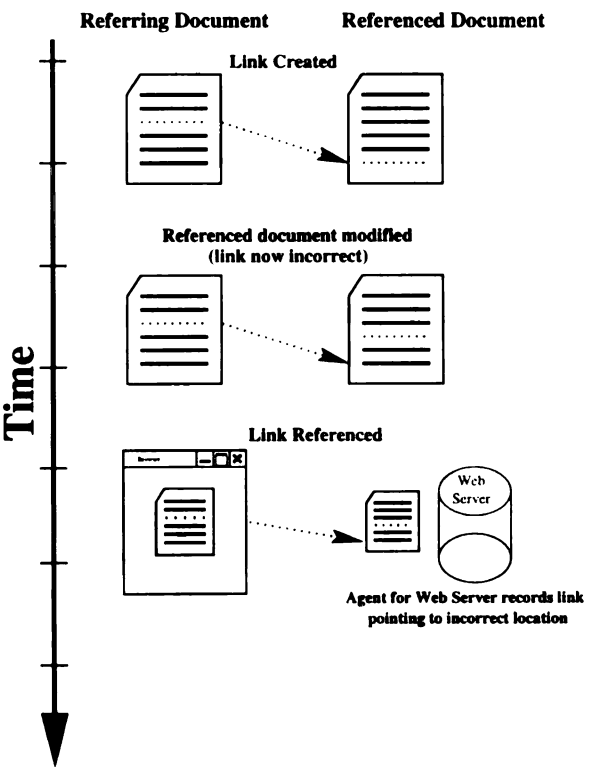


Figure 7.1: Discovering an incorrect link

In this example, a document containing a link is created. The author of that document uses the current version of the referenced document to create the locator for the link.

The author of the referring document does not notify the agent responsible for the referenced document of the link's existence. Subsequently, the referenced document's author makes a change to that document which contextually breaks the referring document's link. It is significant that this change occurs before the link in question is ever followed.

Some time later, a user browsing the referring document decides to follow the link. It is at this time that the agent for the referenced document learns of the existence of the link and records its existence in its link database. Unfortunately, the agent has no way of knowing if the link is contextually broken, and therefore it assumes the link is correct. Not only will the user browsing the referring document be pointed to an incorrect portion of the referenced document, the incorrect resolution of the link will be perpetuated by the referenced document's agent after any significant subsequent changes to the referenced document. In other words, once the link is incorrectly recorded in the database, it will be "fixed" by the link maintenance process so that it stays broken.

We call a link which is discovered by the link agent in this situation an *invalid discovered link*. Adding an invalid link to an agent's database does not really cause much harm. When the user follows the link, the result will be that the target of the link is not the same as that originally intended by the document author. If the agent did not record the invalid link, then the target of the link will also not be the author's intended target. The actual target the link resolves to will most likely be different in the two cases, but in both cases it will be incorrect. If at some time in the future the owner of the referencing document does start sending explicit

AddReference notifications, then it is important at that time to verify the correctness of any discovered links held by the agent. This is the case since at this time authors at the referencing site begin to have an expectation that links they create will remain correct.

The rest of this chapter describes the methods and results of an investigation into how frequently invalid discovered links are encountered. The results of this investigation give us an idea about how well an agent can maintain contextual link integrity without cooperation from document authors.

7.2 Methods

Determining the frequency of discovering an invalid link is a difficult proposition. We must be able to answer the question "How frequently is a referenced page modified between the time a link is created and the first time that link is followed?" Ideally we need to be notified immediately whenever any link pointing into an agent's document space is created. However, the impossibility of obtaining such information in an unrestricted environment is the very reason that the notion of discovered links came about in the first place.

In order to approximate the answer to the question, our search will be restricted to the document spaces of a specific set of agents. We periodically poll the document space of this set of agents and look for links that point into the document space of other agents in the set. This requires the ability to enumerate the document space of

each agent in the set; access to the configuration files of the server(s) associated with the agents is necessary to perform this enumeration.

In addition to tracking the creation of links, we must track two other activities. The first of these activities is modifications made to the documents in the selected agents' aggregate document space. Again we ideally would like to be notified immediately when a document modification takes place. Without operating system support, however, this is impossible and so we resort again to periodic polling.[1]

The second activity we must track is referencing of the links in question. This activity can be precisely monitored, since most web servers provide the ability to log referrer information at the time they encounter it. Of course the permission to monitor these log files is required for the set of agents in question.

We tracked link creation, page modifications and link references for a period of six weeks. Due to administrative difficulties, the set of servers used in our study contained a single server, our departmental web server. Studies have shown [59] that documents located on servers in the .edu domain are the least frequently modified documents, so the bias of studying this particular server must be taken into account when analyzing the results.

We have divided the links discovered during the data gathering process into two categories – links that point to pages that were generated by some program, and links that were not. Generated links were typically generated by the **javadoc** [60]

[1]The Win32 API used by Microsoft operating systems provides something close to this ability. Unix operating systems in do not generally provide such a facility, although one has recently been proposed in [58].

program, and their large number appears to be due to an instructor requiring students to use javadoc frequently as part of their assignments.

7.3 Results

During the six week tracking period, we observed the creation of 2678 links that pointed from within one document in the departmental server's web space to another document in that same document space. 526 of these links were referenced during the experimental period.

For each link referenced at least once during the period of time during which data was gathered, the chart in figure 7.2 indicates how many times the referenced document was modified **before** the link was referenced.

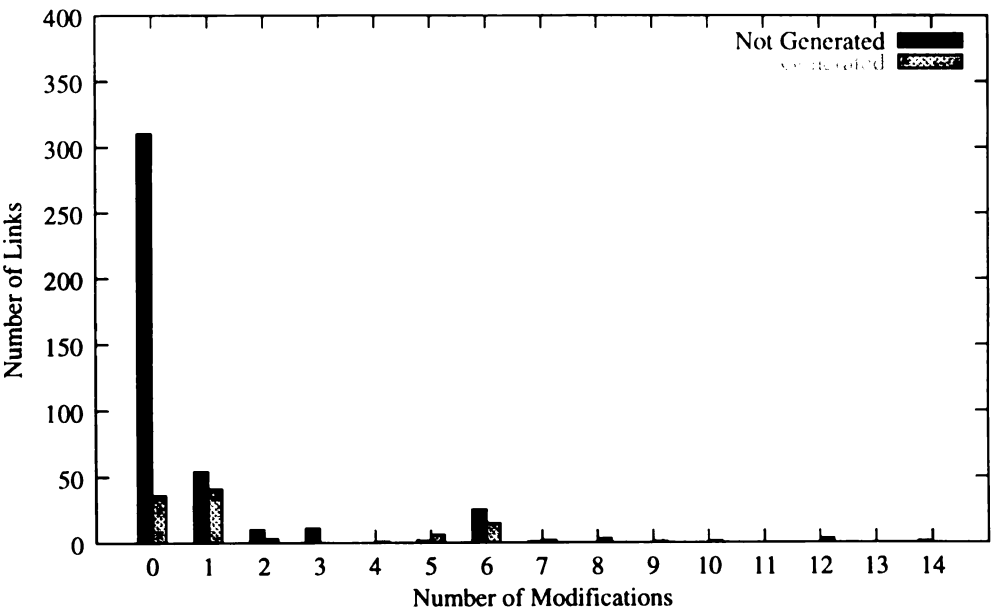


Figure 7.2: Number of times modified before first reference occurs

The entry in the chart where the x-axis is 0 indicates links that, if discovered by an agent during a reference, would be valid. There are 346 links that fit this category, indicating that approximately 2/3 of the links referenced at least once would be valid at the time of discovery.

Pages that were modified one or more times before they were referenced **may** lead to invalid discovered links, depending on whether the modification made at that time caused the link to be contextually broken. At this time, we do not have any data on how likely a document modification would contextually break a link. This data depends on the types of modifications that are typically made to documents, and the types of locators users typically create.

A substantial number of the links created during the trial were not referenced during the trial. Figure 7.3 shows the distribution of modification counts for those pages during the trial period.

For the generated links, we can argue that invalid discovered links are unlikely to occur when one of these links is referenced. This is because both the referring and referenced documents are typically generated by the program responsible for generating the pages, and it is likely that the program would correctly maintain any links it creates. While the user is able to add additional links to the generated documentation, creating candidates for invalid discovery, this seems unlikely in the case of javadoc, since these links would not be maintained between versions of the documentation; users would soon learn not to create links in the referring pages.

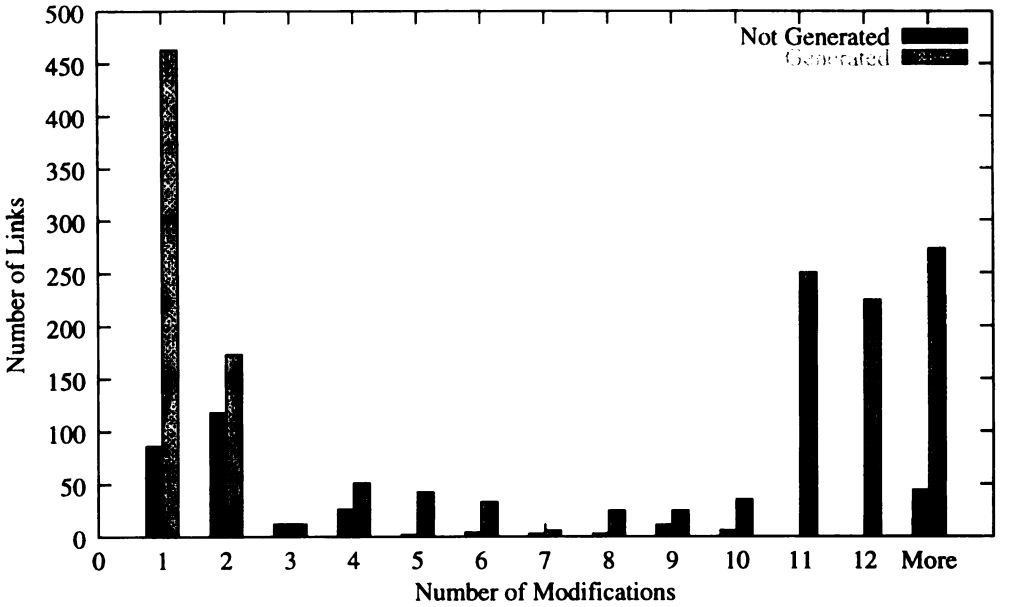


Figure 7.3: Number of times modified without being referenced

7.4 Improving the Results

The statistics in the previous section compare the time between the first modification of a document and the the first reference of a particular link pointing to that document. We can improve the performance of the algorithm by a slight modifcation to the discovery process. As we stated in section 7.1, when the agent receives a request for a document which has the referer header set, and the agent does not already know about the existence of the link, it "discovers" that link. At the time the agent receives the request containing a referer header, it can in fact discover the existence of **all** links in the referring document that point into its document space. In other words, the agent is really discovering the existence of the **referring document**, not of the link itself.

Making this modification to the link discovery process changes the question we must ask to determine how frequently discovered links will be invalid. The question becomes "How frequently is the referenced page modified between the time the link in question is created and the first time **any** link in the referring document pointing into the referenced agent's document space is referenced?" Figure 7.4 shows how the results change when using this modified discovery strategy.

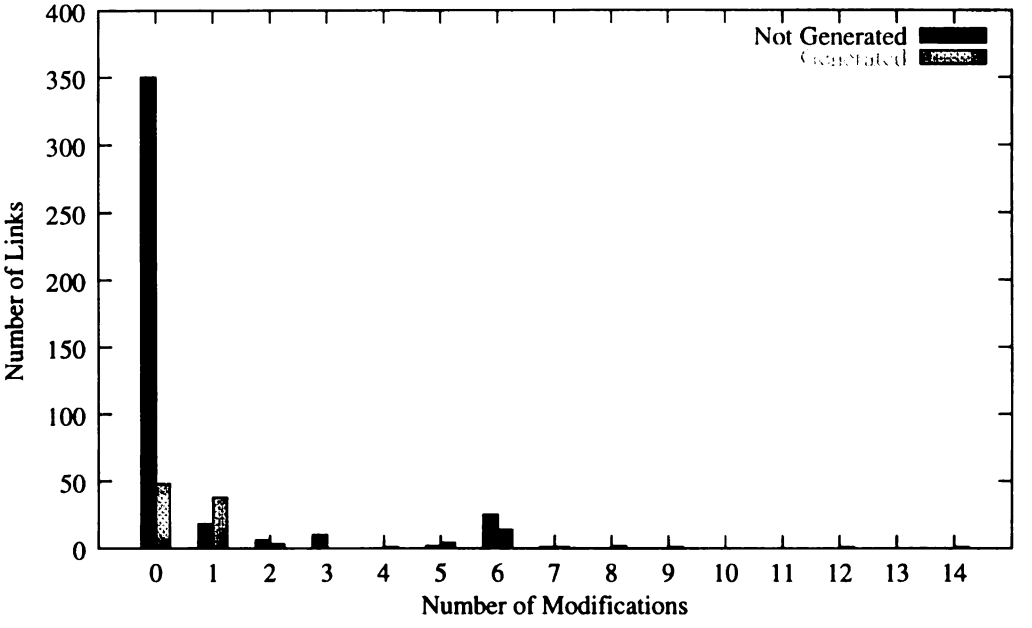


Figure 7.4: Results using modified algorithm

These results are based on the data that we gathered for this study, which only track intrasite links. The improvement is realized because most pages contain multiple links (and, since they are intrasite links, they must point to the same server). The likelihood of multiple links in a specific page to a single agent's aggregate document space is probably lower than what we have observed in this data.

We have attempted to estimate this likelihood by reexamining the data on our departmental web server. Each document in the server's document space was analyzed, and we recorded the web server associated with each link pointing to an offsite server. This data was summarized by computing, for each document, the number of links pointing to each unique server. Figure 7.5 shows the results averaging these per-server link counts for each document.

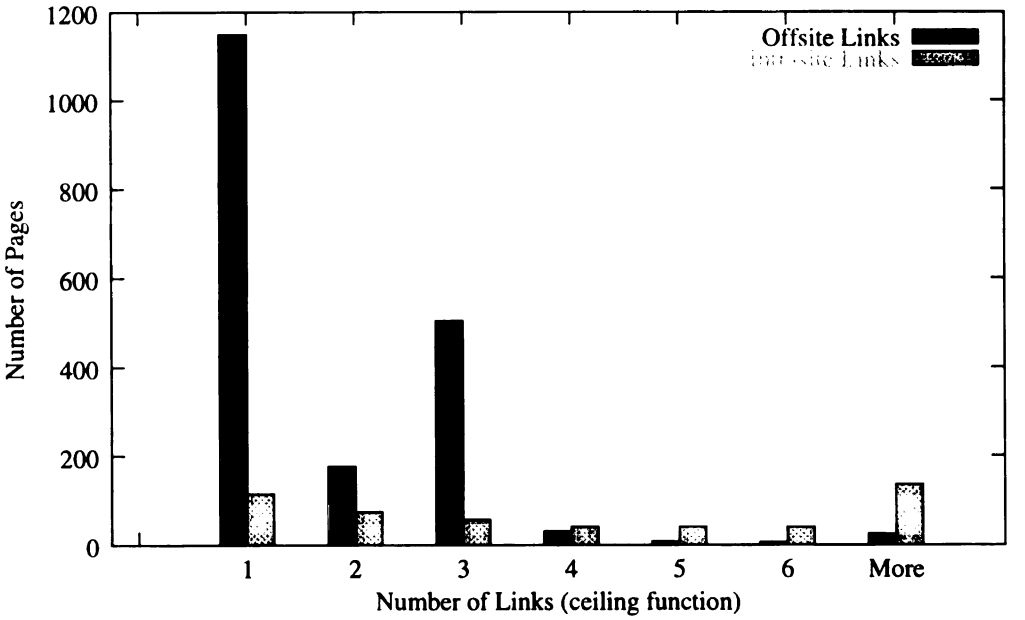


Figure 7.5: Average number of references per server in a document

It is possible that as the use of XPointers as locators becomes more prevalent, the likelihood of multiple links in a page pointing to an agent's aggregate document space will increase. We say this because users may create more links when they have the ability to more accurately specify the target of a link.

7.5 Conclusions

We have seen that discovering links through the use of the HTTP referer header can allow an agent to maintain contextual link integrity for referring documents residing on a server not serviced by an agent with a reasonably high degree of success. The possibility of discovering an invalid link is a risk that the agent must take if it decides to add discovered links to its database. We have seen, however, that discovering an invalid link does not necessarily result in behavior that differs significantly from not discovering the link.

Discovered links may be treated differently by the agent than links whose existence has been explicitly communicated by another agent. For example, they may be discarded at any time if some of the resources used by the agent need to be reclaimed (for example, disk space to hold the link database or memory needed to hold the link map described in section 6.4).

The evidence we have shows that discovering links is a worthwhile proposition for an agent that wishes to increase the probability that links pointing into its document space are resolved correctly. A more expansive study is required to further validate this evidence.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

The problem of maintaining contextual link integrity is an important one. As the use of standards such as XLink and XPointer become more widespread, web users will increasingly encounter documents containing contextually broken links. Preventing this situation gives an information provider a definite advantage in terms of the value of the information being provided. Most previous approaches to maintaining WWW link integrity apply only to preventing existentially broken links. This dissertationdescribes a system we have implemented to maintain contextual link integrity in today's WWW environment.

Our system is open in the sense that specialized editors are not required. Users can continue to use whatever tools they are familiar with to create their documents. The only requirement imposed by the system is that a "check-in/out" process is used by

the users when they wish to being/end editing of a particular document. This process then takes care of all communication with the relevant agents.

We have described algorithms that allow web servers to automatically and efficiently maintain link integrity . These algorithms have been shown to be efficient compared to simplistic brute force algorithms that might be applied. As far as we know, our algorithm for contextually broken link detection is the first algorithm proposed to detect broken links efficiently. We have shown that our algorithm is an optimal algorithm for solving this problem.

Furthermore, we have developed an agent proxy to provide contextual link integrity. The most significant aspect of this proxy architecture is its ability to interact with the current generation of web browsers and servers with little modification. We have argued that the communication protocol used by these agents is correct, and shown that it does not impose a significant processing or communication overhead on the servers and network infrastructure of the current WWW. We have shown that the agent performs correctly in the event of several common failure scenarios.

One of the biggest criticisms of the agent approach is the dependence upon communication between independently managed agents. Indeed, several researchers dismiss the possibility of solving the link integrity problem in this way primarily due to this issue. We have shown that inter-agent communication is unavoidable for agents wishing to guarantee 100% contextual link integrity. We do not fully agree that inter-agent communication is the determining factor in deciding whether our approach is feasible. Microsoft is basing a large part of its corporate strategy for the early 2000's on communication between web services [61], which indicates that at least the corporate world

does not view such communication as prohibitive. Even if most information providers were to agree to inter-agent communication, however, there certainly could be those who would not wish to do so. Thus, allowing agents to operate effectively without inter-agent communication is certainly an issue that needs to be addressed. In chapter 7, we described an alternative approach through which agents could "discover" the existence of links into their document spaces independently, reducing the need for inter-agent communication. Through a study of document modification histories and link access data, we have shown that this discovery technique can effectively provide link agents with the information necessary to maintain contextual link integrity most of the time.

In order for our system to be widely deployed, it must not introduce significant time delays in the document development process. We have extensively researched the possibility of implementing a distributed processing architecture for resource efficient, fast processing of XML documents, an operation that will become increasingly important as the use of XML as a data description language proliferates. We have seen evidence that a distributed architecture would be beneficial, based on the data gained from implementing a multi-threaded document processor. This multi-threaded processor has been shown to have significant performance benefits over a single-threaded document processor, particularly when running in a machine in which there are multiple processors.

8.2 Future Work

Taken as a whole, the algorithms and systems described in this dissertationcan be used by an organization to guarantee contextual link integrity for the documents in its web servers' document spaces. However, there are several opportunities to extend and improve on the work we have already accomplished.

Firstly, and perhaps most importantly, the correctness of the agent communication protocol described in chapter 6 is a crucial component of our system. While we have argued for its correctness, a formal verification of the correctness of these protocols is certainly desirable.

A second area where further research could be beneficial would be to extend the study of document modifications and web access patterns described in chapter 7. The results we have reported are based on data that is limited in scope and size. To further validate these results, a longer study that would hopefully involve multiple web sites should be conducted. Involving multiple sites from different types of domains (e.g. .com and .org) will further confirm the validity of the discovered link approach.

Although we were not able to successfully implement distributed document processing with reasonable performance, the idea still seems to be a good one. Distributing the processing load among multiple machines allows us to more efficiently process large documents. In addition, by aggregating the memory resources of a set of machines, it should be possible to process large documents a single document processor would be incapable of handling.

Due to the relatively high cost of making remote procedure calls, we have found that pre-fetching and caching of portions of the tree at the client is a useful technique. Determining an appropriate pre-fetch and cache replacement strategy is important, since performance of the distributed document processor can be significantly hampered if poor decisions are made for these strategies. In order to decide what the optimal cache replacement algorithm is, it is necessary to have a model of typical application access patterns to the DOM representation of a document. A study of several significant DOM applications could be used to develop a model of how document content is typically accessed, thus leading to more efficient caching.

Finally, as the WWW continues to evolve, it is very likely that the proportion of **dynamic documents** will continue to increase. A dynamic document is a document that is generated at the time it is requested, and is typically not stored in a single file. Our system is currently based on the 'document=file' paradigm, which is still present in many WWW sites. Developing a system that could deal with dynamic documents is certainly a challenge.

Bibliography

Bibliography

[1] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystk-Nielsen, and Arthur Secret. The World-Wide Web. *Communications of the ACM*, 37(8):76–82, August 1994.

[2] Theodor Nelson. *Literary Machines.* Ted Nelson, 1987.

[3] Nigel Woodhead. *Hypertext and Hypermedia: Theory and Applications.* Sigma Press, Winslow, England, 1^{st} edition, 1990.

[4] Vannevar Bush. As we may think. *The Atlantic Monthly*, July 1945.

[5] International Organization for Standardization, Geneva, Switzerland. *Information Technology - Hypermedia/Time-based Structuring Language (HyTime) Second edition, 1997-08-01.* International Standard ISO 10744-1997. Available online at http://www.ornl.gov/sgml/wg8/docs/n1920/html/n1920.html.

[6] Frank Kappe. Hyper-G: A Distributed Information System. In B. Leiner, editor, *Proceedings of INET*, San Francisco, California, 1993. Internet Society.

[7] Hugh C. Davis, Simon Knigh, and Wendy Hall. Light hypermedia link services: A study of third party application integration. In *Proceedings of the ACM Conference on Hypertext, ECHT'94*, pages 41–50. Assocation for Computing Machinery, 1994.

[8] Frank Halasz and Mayer Schwartz. The dexter hypertext reference model. *Communications of the ACM*, 37(2):30–39, February 1994.

[9] Hugh C. Davis. Referential integrity of links in open hypermedia systems. In *Proceedings of Hypertext 98*, pages 207–216, Pittsburgh PA USA, 1998. Assocation for Computing Machinery.

[10] Text Encoding Initiative, [Online] Available http://www.uic.edu/orgs/tei, published 1998.

[11] Steven J. Derose and David Durand. The TEI Hypertext Guidelines. *Computers and the Humanities*, 29(3):181–190, 1995.

[12] S. Hockey. Developing access to electronic texts in the humanities. *Computers in Libraries*, 13(2):41–43, February 1993.

[13] Douglas C. Engelbart. Toward augmenting the human intellect and boosting our collective IQ. *Communications of the ACM*, 38(8):30–33, August 1995.

[14] Tim Berners-Lee. Hypertext markup language - 2.0, [Online] Available http://www.w3.org/MarkUp/html-spec/html-spec.ps, published November 1995.

[15] Dave Raggett. HTML 3.2 Reference Specification, [Online] Available http://www.w3.org/TR/REC-html32, published January 1997.

[16] HTML 4.01 Specification, [Online] Available http://www.w3.org/TR/html4/, published January 1999.

[17] Extensible Markup Language, [Online] Available http://www.w3.org/TR/REC-xml, published February 1998. This is a W3C Recommendation.

[18] International Organization for Standardization, Geneva, Switzerland. *Information Processing, Text and Office Systems, Standard Generalized Markup Language (SGML) = Traitement de l'information, systemes bureautiques, langage standard généralisé de balisage (SGML). First edition, 1986-10-15.* International Standard ISO 8879-1986. Federal information processing standard; FIPS PUB 152.

[19] Steven J. DeRose. *The SGML FAQ Book.* Kluwer Academic Press, Boston, Dordrect, London, 1^{st} edition, 1997.

[20] ISO/IEC 10744:1997. Hypermedia/Time-based Structuring Language (Hytime), [Online] Available http://www.ornl.gov/sgml/wg4/docs/n1920, published 1997.

[21] XML Linking Language (XLink), [Online] Available http://www.w3.org/TR/1998/REC-xlink-19980210, published March 1998. This is a work in progress and subject to change at any time.

[22] XML Linking Language (XLink), [Online] Available http://www.w3.org/TR/xlink, published June 2000. This is a W3C Candidate Recommendation.

[23] XML Pointer Language (XPointer), [Online] Available http://www.w3.org/TR/1998/WD-xptr-19980303, published March 1998. This is a work in progress and subject to change at any time.

[24] XML Pointer Language (XPointer), [Online] Available http://www.w3.org/TR/xptr, published June 2000. This is a W3C Candidate Recommendation.

[25] Niels Olof Bouvin. Unifying strategies for web augmentation. In *Proceedings of the ACM Conference on Hypertext, ECHT'99*, pages 91–100. ACM, ACM, 1999.

[26] Alberto O. Mendelzon and Tova Milo. Formal models of web queries. In *Proceedings of PODS 97*, pages 134–143, 1997.

[27] Frank Halasz and Mayer Schwartz. The dexter hypertext reference model. In J. Moline, D. Benigni, and J. Baronas, editors, *Proceedings of The Hypertext Standardization Workshop*, pages 95–133, Gaithersburg, MD, 1990. National Institute of Standards.

[28] U. Black. *OSI: A Model for Computer Communications Standards.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1991.

[29] Norman Meyrowitz. The missing link: Why we're all doing hypertext wrong. In Edward Barrett, editor, *The Society of Text: Hypertext, Hypermedia, and the Social Construction of Information*, pages 107–114. MIT Press, Cambridge, Massachusetts, 1989.

[30] Bernard J. Haan, Paul Kahn, Victor A. Riley, James H. Coombs, and Norman K. Meyrowitz. Iris hypermedia services. *Communications of the ACM*, i35(1):36–51, January 1992.

[31] Steven J. DeRose and David G. Durand. *Making Hypermedia Work: A User's Guide to HyTime.* Kluwer Academic Publishers, Norwell, Massachusetts 02061 USA, 1994.

[32] Udo Flohr. Hyper-G organizes the web. *Byte Magazine*, 20(11):59–64, November 1995.

[33] Keith Andrews, Frank Kappe, and Hermann Maurer. The Hyper-G network information system. *Journal of Universal Computer Science*, 1(4):206–220, April 1995.

[34] Frank Kappe. A scalable architecture for maintaining referential integrity in distrubted information systems. *Journal of Universal Computer Science*, 1(2):84–104, February 1995.

[35] Leslie Carr, Hugh Davis, David De Roure, Wendy Hall, and Gary Hill. Open information services. In *Proceedings of the Fifth International World Wide Web Conference*, Paris, France, 1996. World Wide Web Consortium.

[36] Jim Pitkow. Web characterization overview, [Online] Available http://www.w3.org/WCA/Reports/1998-01-PDG-answers.htm, published March 1999.

[37] Michael L. Creech. Author-oriented link management. In *Proceedings of the Fifth International World Wide Web Conference*, Paris, France, 1996. World Wide Web Consortium. Available at http://www5conf.inria.fr/fich_html/papers/P11/Overview.html.

[38] PURL home page, [Online] Available http://purl.oclc.org.

[39] P. Mockapetris. Domain Names – Implementation and Specification. RFC 1035, 1987.

[40] S. Ingham, D. Caughey and M. Little. Fixing the "broken link" Problem: The W3Objects Approach. *Computer Networks and ISDN Systems*, 28(7-11):1255, 1996.

[41] Andres S. Noetzel and Stanley M. Selkow. *An Analysis of the General Tree-Editing Problem*, pages 237–252. Addison Wesley, 1983.

[42] Thomas A. Phelps and Robert Wilensky. Robust intra-document locations. In *Proceedings of the Ninth International World Wide Web Conference*, Amsterdam, Netherlands, 2000. World Wide Web Consortium, Foretec Seminars, Inc.

[43] Leslie Carr, David De Roure, Wendy Hall, and Gary Hill. The distributed link service: A tool for publishers, authors and readers. *World Wide Web Journal*, 1(1):647–656, 1995.

[44] Document Object Model DOM Level 1 Specification, [Online] Available http://www.w3.org/TR/REC-DOM-Level-1, published October 1998. This is a W3C Recommendation.

[45] Jon Bosak. Xml version of the bible, [Online] Available http://sunsite.unc.edu/pub/sun-info/standards/xml/eg/rel200.zip, published Oct 1998.

[46] James C. Galvin. *Harmony of the Gospels*, pages 1931–1936. Tyndale House Publishers and Zondervan Publishing Company, Grand Rapids, Michigan, 1991.

[47] David Megginson. SAX 1.0: The Simple API for Xml, [Online] Available http://www.megginson.com/SAX/index.html, published May 1998.

[48] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee, L.Masinter, and P. Leach. HTTP 1.1 Specification. RFC 2616, [Online] Available http://www.w3.org/Protocols/rfc2068/rfc2068, published June 1999.

[49] Sun Microsystems. Java Remote Method Invocation Specification, [Online] Available http://www.javasoft.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html, published October 1998.

[50] Sun Microsystems. Getting Started Using RMI, [Online] Available http://www.javasoft.com/products/jdk/1.2/docs/guide/rmi/getstart.doc.html, published 1999.

[51] D. Kristol and L. Montulli. HTTP state management mechanism. RFC 2109, February 1997.

[52] Junkbusters Corporation. How web servers' cookies threaten your privacy, [Online] Available http://www.junkbusters.com/ht/en/cookies.html, downloaded August 2000.

[53] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–564, July 1978.

[54] D. Mills. Network Time Protocol (version 3) specification. RFC 1305, March 1992.

[55] T. Brisco. DNS Support for Load Balancing. RFC 1794, April 1995.

[56] E. Anderson, D. Patterson, and E. Brewer. The Magicrouter, an Application of Fast Packet Interposing, [Online] Available http://www.cs.berkeley.edu/ẽanders/projects/magicrouter/osdi96-mr-submission.ps, published May 1996.

[57] IBM alphaWorks. XML parser for Java, [Online] Available http://www.alphaworks.ibm.com/tech/xml4j/, published Feb 1998.

[58] Jeffrey C. Mogul Gaurav Banga and Peter Druschel. A scalable and explicit event delivery mechanism. In *Proceedings of the 1999 USENIX Technical Conference*, Monterey, California, 1999. Advanced Computing Systems Association. Available at http://www.usenix.org/events/usenix01/cfp/banga/banga_html/.

[59] Xiangping Chen and Prasant Mohapatra. Lifetime behavior and its impact on web caching. In *International Conference on Computer Communications and Networks*, Estes Park, Colorado, 1999. Institute of Electrical and Electronics Engineers.

[60] Sun Microsystems. Javadoc 1.3, [Online] Available http://www.javasoft.com/j2se/1.3/docs/tooldocs/javadoc/index.html, downloaded August 2000.

[61] Microsoft Corporation. Microsoft.NET: Realizing the next generation internet, [Online] Available http://www.microsoft.com/net/whitepaper.asp, published June 2000.