Autonomous Cooperating Web Crawlers

by

Gregory Louis McLearn

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2002

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

A web crawler provides an automated way to discover web events – creation, deletion, or updates of web pages. Competition among web crawlers results in redundant crawling, wasted resources, and less-than-timely discovery of such events. This thesis presents a cooperative sharing crawler algorithm and sharing protocol. Without resorting to altruistic practices, competing (yet cooperative) web crawlers can mutually share discovered web events with one another to maintain a more accurate representation of the web than is currently achieved by traditional polling crawlers.

The choice to share or merge is entirely up to an individual crawler: sharing is the act of allowing a crawler M to access another crawler's web-event data (call this crawler S), and merging occurs when crawler M requests web-event data from crawler S. Crawlers can choose to share with competing crawlers if it can help reduce contention between peers for resources associated with the act of crawling. Crawlers can choose to merge from competing peers if it helps them to maintain a more accurate representation of the web at less cost than directly polling web pages. Crawlers can control how often they choose to merge through the use of a parameter ρ, which dictates the percentage of time spent either polling or merging with a peer. Depending on certain conditions, pathological behaviour can arise if polling or merging is the only form of data collection.

Simulations of communities of simple cooperating web crawlers successfully show that a combination of polling and merging ($0 < \rho < 1$) can allow an individual member of the cooperating community a higher degree of accuracy in their representation of the web as compared to a traditional polling crawler. Furthermore, if web crawlers are allowed to evaluate their own performance, they can dynamically switch between periods of polling and merging to still perform better than traditional crawlers. The mutual performance gain increases as more crawlers are added to the community.

Acknowledgements

I would like to thank everyone who has given their assistance and support during the completion of this thesis.

Special thanks must go to my supervisor Gordon V. Cormack. His insights and criticisms proved to be invaluable, and his patience was infinite. Thanks for everything, Gord.

I would also like to thank my readers, Charlie L.A. Clarke and William B. Cowan for taking the time to read this work and for supplying helpful comments.

I must also thank Karin Hung – for she is truly an inspiring person and capable of great things.

Thanks to the members of the Programming Languages Group, who provided not only intellectual moments, but also many needed distractions.

This work has been made possible through generous funding by the National Sciences and Engineering Research Council (NSERC) of Canada.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

This thesis examines mechanisms whereby a set of autonomous web crawlers can share information to their mutual benefit. The Hyptertext Transfer Protocol (HTTP) – the protocol that drives the web – does not have the ability to inform interested parties of *web-events* – the creation, deletion, or updates of web objects (pages, images, etc.). The most common method of addressing this problem is to use a polling web crawler – a software program designed to traverse the web in search of web events. Web crawlers consume a great deal of network bandwidth and do not discover web-events in a timely manner; the most powerful web crawlers can take weeks or months to discover a particular web-event[28].

There are a large number of crawlers currently active on the web. They compete for bandwidth, but by and large, do not share their discoveries. This thesis describes a general protocol to allow competing web crawlers to cooperatively share knowledge of web events. The choice to share or merge is entirely up to an individual crawler: *sharing* is the act of allowing one crawler (call it crawler M) to access another crawler's web-event data (call this crawler S). *Merging* occurs when crawler M requests web-event data from crawler S. Crawlers can choose to share with competing crawlers if it can help reduce contention between peers for resources associated with the act of crawling. Crawlers can choose to merge from competing peers if it helps them to maintain a more accurate representation of the web at less cost than directly polling web objects. It is hypothesized that crawlers using combinations of polling and exchanges of web-event data can mutually achieve more accurate representations of the web than a strictly-polling crawler.

1.1 Problems with Web Crawlers Today

Several major problems affect non-cooperative web crawlers on the web. The first problem is that web crawlers do not maintain a high-degree of freshness. The second is that multiple crawlers can redundantly crawl the same regions of the web. The third is that with the proliferation of web crawlers comes increased contention for shared network resources.

1.1.1 The Freshness Problem

Assume that at time t_i there exists a set of web objects W such that the *state* of all of W can be captured as a set M_i. At time t_j $(j \geq i)$ some subset of W may have changed due to internal or external forces (publishers, database queries, etc.). Another snapshot of the web objects in W at time t_j results in a representation of the state of each object (M_j). A web-event describes changes that occur to any specific object $w \in W$ between time t_i and t_j or any new objects added between t_i and t_j. The set of objects that have changed between t_i and t_j define the set of web-events $S_{i,j}$. The fraction of web objects that have not changed between t_i and

Server	**avg % hits due to web crawlers per day**	**avg % bytes due to web crawlers per day**
`goedel`	6.5%	13.2%
`mef07`	10.5%	5.5%

Table 1.1: Traffic associated with crawlers at two busy web servers at the University of Waterloo.

t_j represents the *freshness* of the set W for the interval $t_j - t_i$:

$$freshness = \frac{|W| - |S_{i,j}|}{|W|}$$

Crawlers that try to keep a fresh set of web pages must schedule revisitations to existing pages during the course of crawling. If a web crawler requests a web page that has not changed between t_i and t_j, then the web crawler has wasted their resources, as well as the resources of the web server. This problem is magnified when hundreds of independent, competing web crawlers visit a web server over a period of time.

1.1.2 Bandwidth and Overlap Problems

Table 1.1 show the activity of web crawlers as they request information from two busy web servers at the University of Waterloo: `mef07.uwaterloo.ca`[1] and `goedel.uwaterloo.ca`[2]. All activity was recorded over a three-month period from May 17, 2002 to July 13, 2002. Web crawlers were identified as any IP attempting

[1] `mef07.uwaterloo.ca` is the main web server for the various math and computing science departments and graduate students.

[2] `goedel.uwaterloo.ca` is the main web server for all undergraduate math and computing science department information as well as undergraduate students.

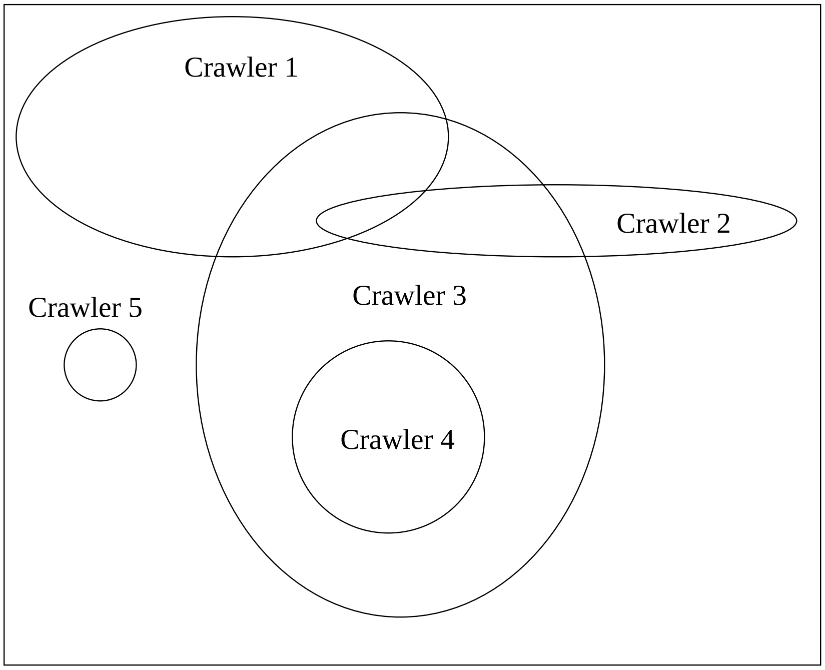


Figure 1.1: Overlap of five crawlers in the web. Some crawlers can crawl large regions and others only crawl small regions. Some crawlers overlap completely and have a complete duplication of effort such as the case of crawler four (4) within crawler three (3). All areas not contained within an ellipse is considered uncrawled data.

to download `/robots.txt`[3]. The table shows that even though web crawlers constitute about 0.5% of the overall number of clients[4], they account for a significant portion of the clients causing data traffic to-and-from the web servers.

Competition among crawlers can lead to detrimental behaviour at a more fun-

[3] Any web crawler that aims to be a good net citizen should follow the Robots Exclusion Standard [27]. A single request for the `/robots.txt` file within the three-month period should have been observed, even in the face of a web crawling using a long-persisting `/robots.txt` file cache. Spurious requests for `/robots.txt` from other sources may cause a slight skew in the results. Note that any IPs within the University of Waterloo subnet were excluded; we are only interested in web crawlers operating external to the University.

[4] 1446 unique IPs corresponding to web crawlers were identified versus the over 250000 unique IPs corresponding to non-web crawlers.

damental level. Studies in 1997 indicate a small region (1.4%) of crawling overlap common to major web crawlers at the time. Further analysis showed that pairwise crawling overlap regions between four major search engines of the time ranged from 0.24% to 4.08%[3]. Of course, since this study, the results have very likely been significantly altered.

Over the years, more and more crawlers have been released on the web. Many web crawlers now make use of distributed or parallel technology in efforts to increase a search engine's web coverage. Web crawlers have had negative impacts on Internet resources in the past[26], and so while brute-force crawling techniques may help a crawler gain a small competitive edge, the impact on shared resources could be more problematic.

Figure 1.2 represents a simulation which graphs the freshness for a varying number of crawlers (N), all of which attempt to crawl a simulated web of one million web objects. The cache freshness starts from 1.0 (perfect) and decreases over time (the simulation assumes each crawler starts with a fresh snapshot of the web). The simulation is based on the software described in Chapter 5. A multi-processor contention model is used to simulate bandwidth being divided among several web crawlers. This contention model is described in section 5.1.

Figure 1.2 illustrates how freshness degrades as more crawlers are added to the system (as N increases) in the face of competition for shared resources (bandwidth).

Increasing the freshness for web crawlers should not depend on using more crawlers; instead, crawlers should examine the fundamental way in which they regard competing peers.

Chapter two will examine web crawlers and how they can communicate with one another. Chapter three will develop the protocol used for crawlers to be able to

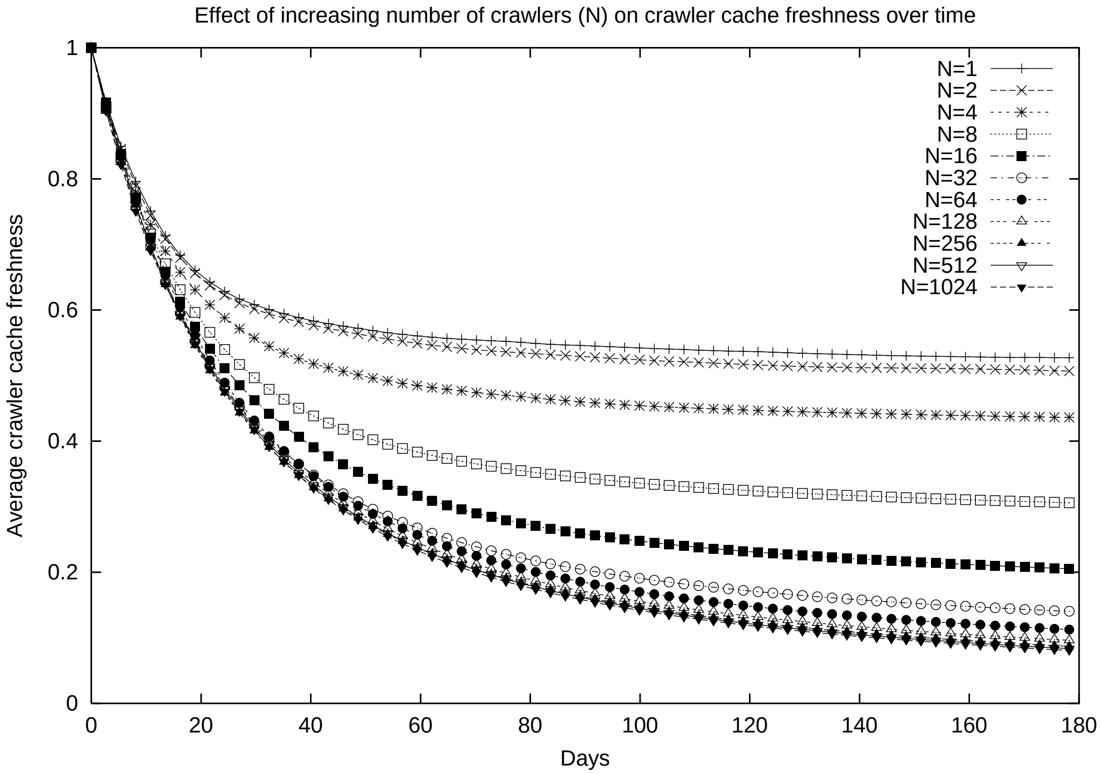


Figure 1.2: Simulation in which a number of crawlers compete for bandwidth as a limited resource.

share information back and forth. A simple implementation of a crawler using the protocol to augment polling is presented in chapter four along with a brief intuitive analysis of how inter-crawler interaction can work. Chapter five describes a simulation architecture used to provide reinforcement of the the analysis of the previous chapter. Chapter six describes experiments using a small sample of sharing policies in which self-interested sharing behaviour is show-cased. Finally, conclusions are drawn and future research directions are discussed in chapter seven.

Chapter 2

Background Information

This chapter presents the literature from three related aspects. First, we examine the evolution of the web crawler from inception to its current incarnation. Second, we examine the communicating paradigms in which web crawlers can share web-events with one another. Finally, we discuss various implementations as to how such web-events could be disseminated efficiently and effectively among cooperating web crawlers.

2.1 Introduction to Web Crawlers

Since its inception in the early 1990's, the World-Wide Web has undergone explosive, exponential growth. Consumers increasingly find themselves unable to browse the ever-changing, distributed hyperlink structure of the web. Furthermore, they are subjected to information overload – literally, information is *too abundant*. Centralized web search indexes have become the panacea of this problem, and web crawlers are generally the enabling technology. With the availability of the technology, web crawlers are also rapidly becoming more popular with individuals for specific information-finding tasks.

A web crawler is an automatic web object retrieval system that exploits the web's dense link structure. It has two primary goals:

1. To seek out new web objects, and
2. To observe changes in previously-discovered web objects (web-event detection).

The basic web crawler algorithm has not changed since the World Wide Web Wanderer (the first reported web crawler) was designed in 1993[29]. Almost all crawlers follow some variant of the basic web-traversal algorithm shown in algorithm 2.1. (Web crawlers typically contain much more functionality than outlined in algorithm 2.1, though such functionality merely serves to satisfy the primary goals).

Algorithm 2.1 Basic web crawler traversal algorithm.

```
Require: $p_0$ is a valid web URL hyperlink
Require: $Q$ is a queue of valid hyperlinks
Require: $P$ is a set of web pages
Require: $H$ is a set of hyperlinks
$Q \leftarrow P_0$                                {insert $P_0$ into the queue $Q$}
while $|Q| \neq \emptyset$ do
    $p \leftarrow Q$                              {get head of queue $Q$}
    retrieve web page $p$
    $P \leftarrow P \cup p$
    extract URL hyperlinks contained in $p$ into $H$
    for all $h \in H, h \notin Q$ do
        $Q \leftarrow h$
    end for
end while
```

Crawlers must continue to deal with issues of scalability as the World-Wide Web expands. How does one efficiently and effectively crawl the current set of almost

2.5 *billion* publically indexable web pages[1] if crawlers are limited by crawling speed and difficulty in predicting web-events?

The speed at which a crawler can traverse the web is limited by a number of factors, including the bandwidth of the crawler and the latency of the network. Modern, heavily multi-threaded crawlers can currently crawl at rates up to 100 web pages per second.

Predicting when a web object is going to change helps to limit the amount of useless polling[5] done by a crawler to determine if it has been updated since the last visit (see primary goal #2). The fewer resources wasted by a crawler doing useless polls, the more that can be delegated to the task of locating new information. Unfortunately, even with numerous studies into how web pages change, prediction is still a relatively difficult task[6, 7, 10, 12, 13, 16, 22].

In the end, crawlers are going to be relying upon communicating with others – be it instances of themselves (in the parallel sense), or with crawlers outside of their controlling domain (ie. a competing corporation). It is the lack of organization between crawlers in the latter sense that this work is based. We are interested in autonomously cooperative sharing web crawlers – crawlers that can make decisions on their own, and communicate with others when the need arises. The next section assesses the different communication paradigms within web crawler communities.

[1] The publically-indexable web consists of all web objects not hidden behind `/robots.txt` protection[27], authentication mechanisms, forms, databases, etc. As of September 2002, the Google search service (http://www.google.com) had indexed about 2.5 billion web documents, 390 million images, and 700 million Usenet messages. These numbers continue to climb. Of course, these numbers represent a lower bound on the size of the entire web (indexed and non-indexed).

2.2 Taxonomy of Crawler Communication

Web crawlers have increasingly become more complex in their design and organization to address the need of crawling the web in a timely manner. Depending on their organization, crawling systems may devote time and energy communicating with one another to help coordinate their behaviour to reduce overlapping crawling regions.

Ho[24] describes a taxonomy of web crawling communication patterns in the context of information-gathering agents[2]:

- non-interactive
- purely coordinated
- coordinated with collaboration
- purely collaborative
- self-interested

Each communication paradigm can be illustrated by figure 2.1. Communication can occur vertically from a central authority to web crawlers, horizontally among peer web crawlers, in both directions, or not at all. (The choice of communication vertically is independent of the choice to communicate horizontally.) Intermediate brokers can exist as part of the hierarchy to aide in scalability and flexibility of information dissemination. They work on behalf of the central authority, and in turn control the actual crawlers. The Harvest system[4] uses a series of brokers similar to those in figure 2.1 to provide an efficient and flexible caching solution.

[2] An agent is a process that performs a task on behalf of a user, usually in collaboration with one or more other agents to perform a collective task and/or reach a mutual goal.

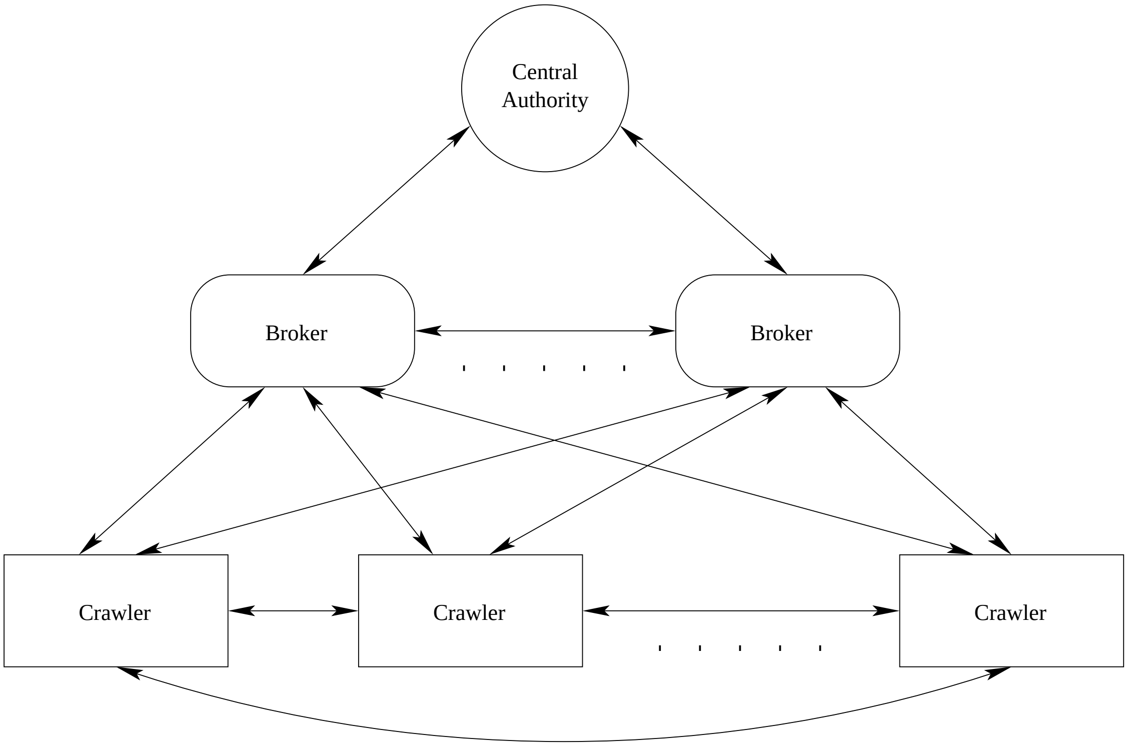


Figure 2.1: Communication interactions within systems of web crawlers.

2.2.1 Non-interactive

Non-interactive web crawlers have limited functional scope and require no communication with other web crawlers. Many personal web crawlers (eg. web-site cloning tools such as the offline browsing mode built into Microsoft Internet Explorer) and the work discussed in the SPHINX project[30] fall into this category. Although they affect the network to some degree, their independent nature implies they would not benefit from adapting to a network of communicating web crawlers.

2.2.2 Coordination

Coordinated crawlers usually manifest themselves as an explicit hierarchy and use vertical communication as shown in figure 2.1. Coordinated web crawlers usually receive their crawling instructions from a central authority. In the simplest case, a central authority hands out groups of URLs to each crawler drone under its control. Each crawler performs one iteration of the basic crawler algorithm described in algorithm 2.1 using this group of URLs. After crawling the initial set, the drone will return the set of crawled web pages (P) and newly extracted hyperlinks (H) to the central authority. The authority collects all extracted URLs to distribute to the drones as directed by its policies. The central authority can ensure that web crawlers working in this fashion can traverse efficiently, with little or no overlap in web coverage. The Public Robot Server Manager (PRSM)[39] is an example of a central authority presiding over a set of vertically-communicating web crawler drones.

2.2.3 Collaboration

Collaboration among crawlers is horizontal communication among peers at the same level of authority. Such crawlers make decisions affecting future crawling based on information received from peers. Collaborating crawlers working within a coordinated system are exemplified by the multi-agent paradigms described in DIAMS[9] and *Amalthaea*[31] which use networks of agents, all interested in achieving the same mutual goal. Purely collaborating systems of web crawlers may exist as part of a distributed system without a controlling entity. The X4 crawler described by Chung[14] uses a set of web crawlers which collaborate in order to allow specific crawlers to crawl particular topics.

2.2.4 Self-interest

A self-interested web crawler is an independently communicating web crawler that has the ability to autonomously self-evaluate its own performance and act accordingly (modify its level of communication with others) to try to improve. A self-interested crawler is interested in performing at an optimal level: decisions that it makes regarding its performance may be based solely on selfish reasoning. Of course, decisions that affect how it communicates with other web crawlers could affect how those web crawlers communicate with it. Hence, any selfish decisions made by an autonomously self-interested crawler may have to take into account the effects of its decisions on the entire system of crawlers.

Ho[24] uses a biological energy model inspired from artificial life theory. The crawlers make crawl meta-data (similar to web-events) available to other remote crawlers. It is up to the local crawler to make *local* (selfish) decisions to either retrieve or ignore the information depending on the cost of retrieval. Ho's crawlers monitor their health (their *energy*) using a combination of a *potential* function and a *cost* function. The potential function determines how costly communication with a remote crawler could be. The cost function determines how beneficial it is to regularly (or semi-regularly) merge crawl data from the remote crawler. The local crawler's energy is updated based on, among other things, how much information is gained or lost by communicating with other crawlers.

Coordination, collaboration and self-interested communication strategies can all be viewed as cooperative types of communication among crawlers. However, coordination and collaboration imply a sense of imposed architectural limits as to how such cooperation can proceed. Self-interested cooperative communication implies the decision-making process may not be fixed by architecture, and that communication patterns could change at any time.

Game Theory and Autonomous Crawlers

The decision-making process performed by autonomous crawlers can be examined from a game-theoretic perspective. The classic game-theory problem "The Prisoner's Dilemma" can be applied to the decision-making process used by crawlers capable of exchanging information[3] by assuming that mutual cooperation can yield the greatest benefit.

Nash equilibria are especially useful for dealing with cooperative crawlers that mutually exchange information with one another[19]. Mutual cooperation can affect the entire system of crawlers. If a set of strategies are being used to exchange information, a Nash Equilibrium occurs when no single crawler can change their

[3] A version of the Prisoner's Dilemma can be read at http://william-king.www.drexel.edu/top/eco/game/dilemma.html

cooperation strategy to increase their personal benefit if all other crawlers in the system do not change their strategies. Multiple equilibria exist when multiple combinations of strategies can result in such stalemates.

2.3 Implementing Web-event Dissemination Systems

The various communication paradigms have several different methods in which they could exchange web-events. Web-event detection and delivery implementations basically break down into those that are based on notification and those that are based on polling. Active notification and polling each have their advantages and disadvantages. Notification systems can deliver every event to an interested party. Unfortunately, high-frequency event data could overwhelm an the listener if too many notifications are delivered. A polling solution can choose the rate at which to poll the data. This runs the risk of missing events in the event that the poll rate is less than the event rate, or can result in many useless requests if events occur less frequently than the source is polled. Active notification systems do not waste such resources.

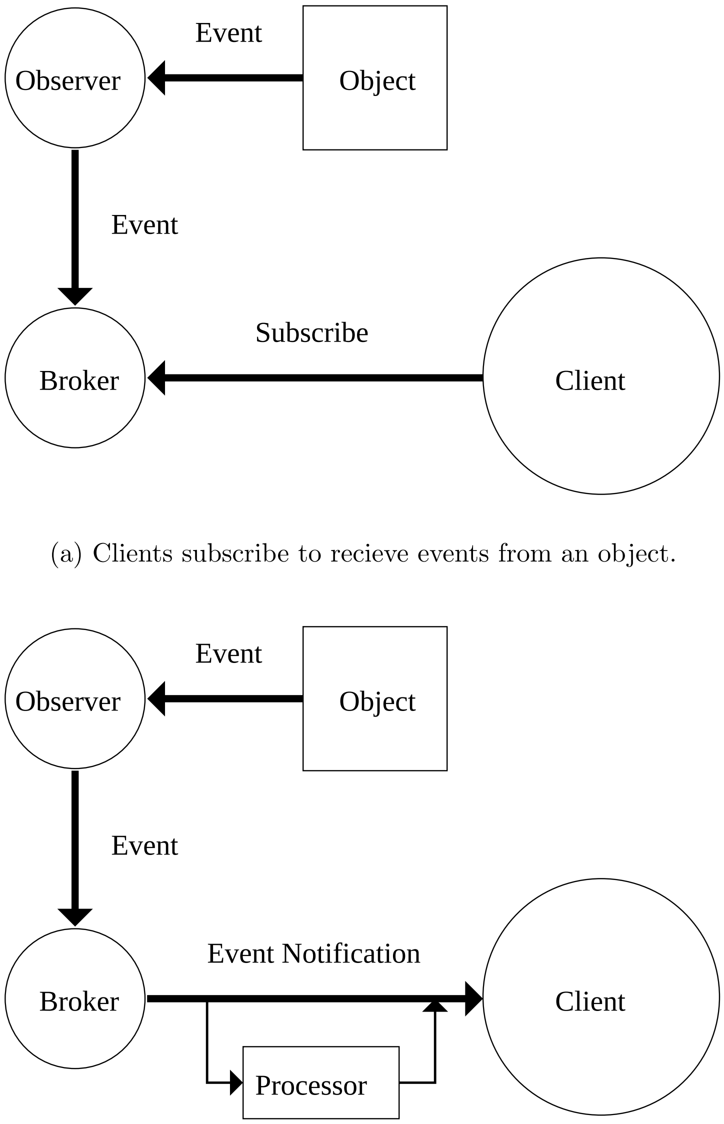
2.3.1 A General Event Notification Architecture

Logically, an event notification system of any sort can be broken down into the constituent parts illustrated in figure 2.2(a). The object is the source of all events, which must be observed in order to be processed. If they are not observed, then they are lost. The act of generating an event is free; it is the act of observing and delivering the event which incur a cost[35]. In figure 2.2(b), the observer is clearly the primary sink, although both the broker and the client are also event sinks. The broker is a component that may allow preprocessing of events before they are delivered to the client such as aggregation (union of events), filtering (intersection/difference) and temporal ordering.

Before an event can be delivered to the client, the broker must know of its existence. The client subscribes to the broker as a recipient of events from the broker's source observer as shown in figure 2.2(a).

Once subscription succeeds, events pass from the source object to the client, optionally passing through a series of processors as seen in figure 2.2(b). As stated above, these processors can exist at the broker-side, or they can exist at the client-side or both. The advantage to broker-side preprocessors is that the number of events being delivered is throttled back. This is useful if the source generates more events than a client can handle (although the broker must still be able to keep up). However, implementing broker-side preprocessing can be expensive in terms of processor time and management.

It should be noted that any of the components in figure 2.2 may be loosely-coupled (shown) or tightly-coupled. A tightly-coupled system may combine any of the source object, observer and broker into single entity. Additionally, the broker component itself may be a complex system as used by Hinze and Faenses[23]. The SIENA system[8] extends the generic architecture by allowing the intermediary broker to be part of a larger distributed mesh. This architecture is also apparent in the proxy configurations discussed in[40].



(a) Clients subscribe to recieve events from an object.

(b) Events are propagated from the object to the client, optionally being filtered.

Figure 2.2: Basic subscription-based event notification architecture

2.3.2 Information Delivery Methods

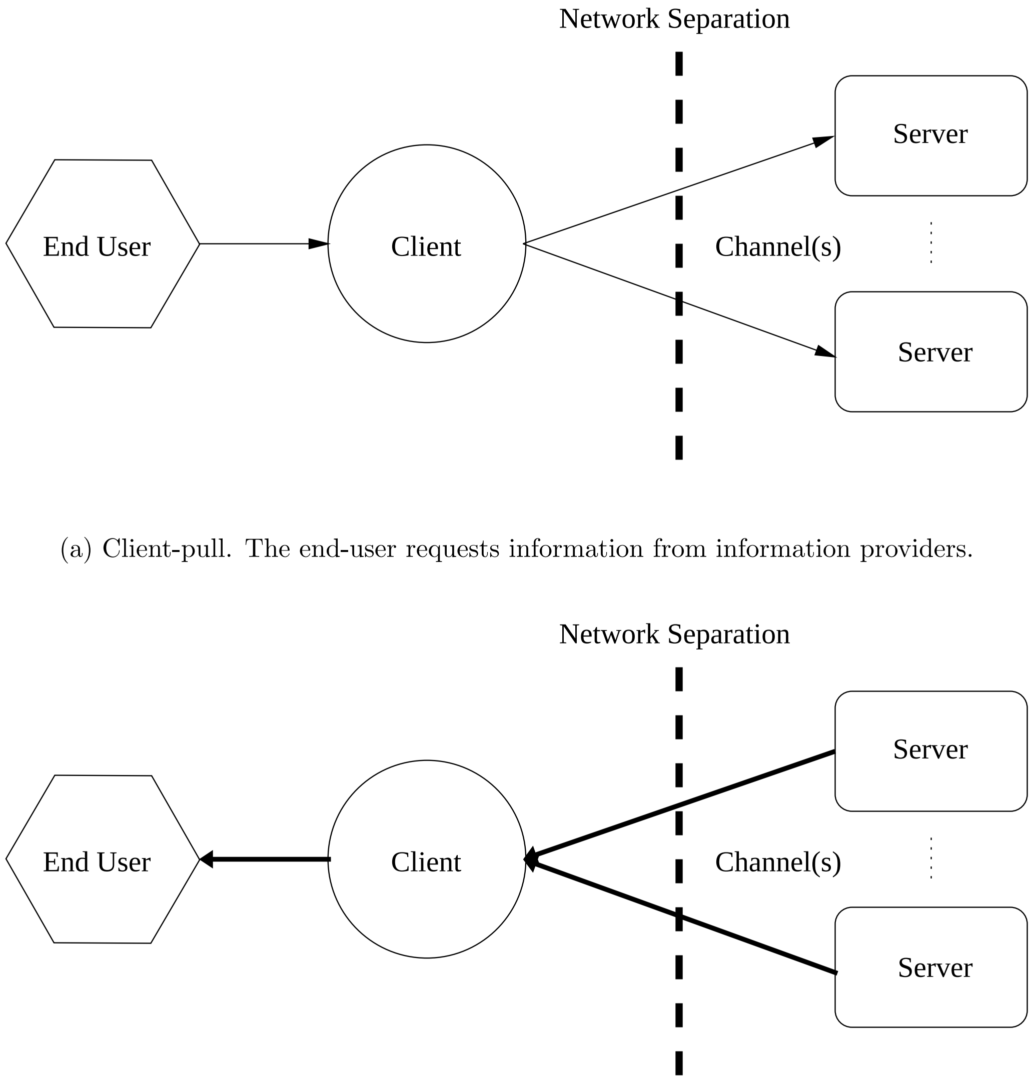
As implied by the generic architecture, there are two distinct forms of communication. These are illustrated in figures 2.3(a) and 2.3(b).

The dominant model of communication on the web is client-pull. Server-push applied to the web emerged in the mid 1990's as a potential panacea to web information overload which was readily becoming a problem[37]. Server-push was to be the active notification system missing on the web. Unfortunately, bandwidth overload from transmitting large amounts of information to thousands of users as well as managing user accounts on the server limited the scalability of push[21, 37]. Even so, other protocols on the Internet continue to make use of push architectures such as Usenet[25]. In an attempt to curb the bandwidth issues, the ideas of automatic periodic client-polling, and a hybrid (push/pull) model of information delivery were adopted (figure 2.4). In the hybrid model, small information packets are sent to the client from the server (eg. URL x has been updated). The client then uses the traditional pull method to retrieve the much larger content-body.

2.4 Distributing Web Events

In order to determine the state of any object on the web, one must poll using HTTP. Implementing a web crawler would be much easier if web objects notified their changes to interested entities[4]. Over the years, a number of initiatives have attempted to incorporate active notification into the context of web objects. Some or

[4] Unfortunately, this active web would suffer crippling scalability issues. Imagine if one million web objects suddenly changed at the same time. This is not too difficult to imagine, considering that there are at about 2.5 billion web *pages*. The network would suffer the same scalability issues as the pure-push transmission model.



(a) Client-pull. The end-user requests information from information providers.

(b) Server-push. The information provider sends information asynchronously to the end-user.

Figure 2.3: Two forms of data transfer.

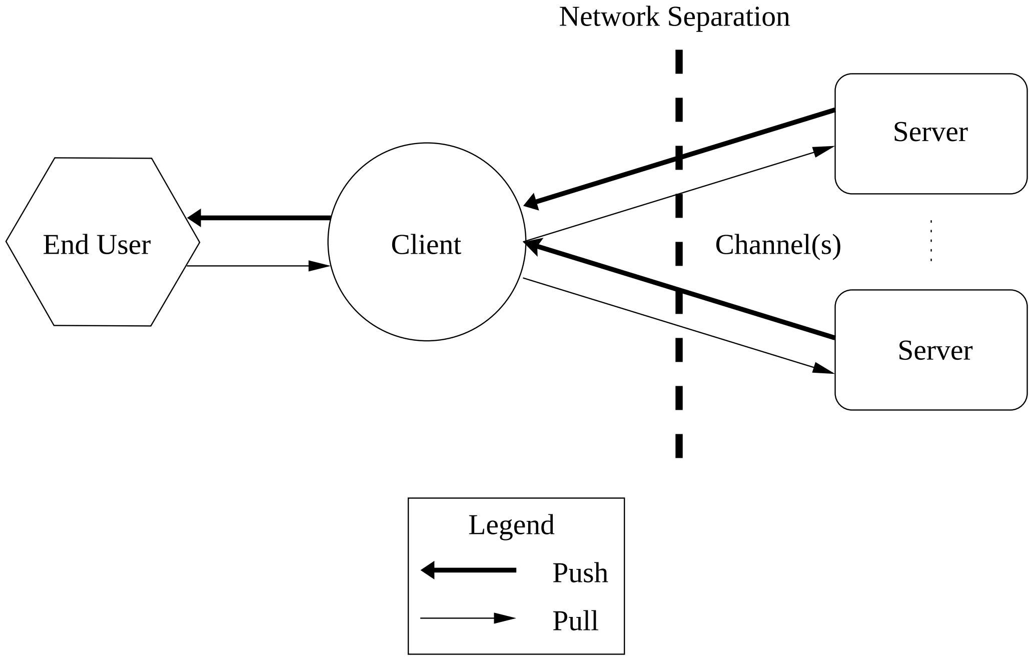


Figure 2.4: A hybrid model combining the best of client-pull and server push.

all of these are interesting ideas to incorporate into crawlers that share information about the ever-changing web.

2.4.1 Changes to HTTP

Hypertext Transfer Protocol [2, 17], the protocol used to drive the web, has been studied in an attempt to add notification features. Particularly of interest are the efforts of the General Event Notification Architecture Base (GENA-Base)[15] and the Event Notification Protocol[34]. In the former, any object with a valid URL can be the source for an event. New HTTP extensions[5] are proposed:

- Subscribe: a client wishes to be informed of events generated by the object

[5] A generalized framework for setting up HTTP extensions can be found in RFC2774 [33].

at the URL.

- Unsubscribe: a client no longer wishes to receive events generated by a URL.
- Poll: a client wishes to check for outstanding events associated with a URL.
- Notify: a client or object at a particular URL wishes to inform another client or object at another URL of an event.

The Event Notification Protocol proposal[34] uses XML coupled with new extensions to HTTP to aide in event notification as applied to web-based distributed authoring and versioning (WebDAV[18, 36]). Interested clients can subscribe to a notification server to receive events about changes to objects at a particular URL.

2.4.2 Crawler-centric Web-Event Dissemination

Aliweb (Archie-like indexing of the web) was a web crawler developed in late 1993 to gather documents[26]. Aliweb did not automatically traverse the web. Instead, it required web servers to register with it, and provide a text file containing meta-information about each and every web document the server wanted to make public. This text file would be periodically retrieved by Aliweb with a frequency set by the web server administrator.

Brandman *et al* re-examines per-web server update indices[5]. Their work allows for search engines to discover changes without retrieving full indices. Crawlers receive web object meta-data (URL, last-modified-date, file size, file checksum, etc.) which can be used to decide if their local copy is stale. This idea is carried still further by Gupta and Campbell, by allowing the local web server to measure the popularity of its own object repository[20]. Web crawlers can then use the popularity and frequency of change data measured at the web server to schedule efficient crawls. Furthermore, the data is pushed to subscribed search engines rather than remaining passive on the local web server.

Ho's own protocol for web-event dissemination among a network of crawler peers describes the role of web object meta-information as a succinct representation of a web event[24].

Chapter 3

Protocol

Ambiguity in communications – indeed, in humans as well as in computer systems – can lead to problems or utter failure. The Internet, and hence, the web, is driven by protocols in order to achieve successful data transmission. It is not enough to say that web crawlers will communicate web-event data with one another. An algorithmic procedure is necessary to ensure that all crawlers interested in cooperation know *how* to share and merge with peers.

The protocol used to allow web crawlers to communicate web-event data with one another must be both simple and efficient. A protocol that uses non-standard technology or that is difficult to implement will be rejected. It is desirable to reuse and exploit existing frameworks and technologies to ensure easy incorporation into existing systems. Web crawlers are already a deeply-rooted software paradigm on the web: crawlers augmented to make use of the web-event data cooperation protocol must remain backwards-compatible. The end-goal is to provide a protocol that focuses primarily on effective and efficient web-event dissemination with the least possible impact on current architectures.

Traditionally, crawlers act as clients and have little or no server-based responsibilities built into them. In order to be able to respond to peer requests for web object meta-data, web crawlers must use web-server capabilities. All communication between web crawler peers uses existing HTTP standards and all information transmitted to and from peers is in easily-parseable human-readable text.

3.1 Web-Events Describe the Ever-changing Web

Web-events are succinct representations of changes to a web object. The smallest amount of information required to convey a web-event is the last-modified-date of a specific URL. Previous knowledge of the last-modified-date of a URL enables a client to determine if indeed a change has occurred[1]. Additional information, such as the classification of a web-event as a CREATE, UPDATE, or DELETE event can be added to allow filtering by event type.

Storing web-events is far cheaper than storing the full text of a web object. They can be encoded in about 200 bytes and compressed to about 25% of that size.

Web-events must follow the state transitions shown in figure 3.1. Exactly one CREATE web-event and one DELETE web-event are generated and observed. A variable number of UPDATE events can be generated (U_g) and observed (U_{ob}). The difference between the set of generated web-events for a particular web object and those events observed by a web crawler manifests itself as the number of missed events ($|U_g| - |U_{ob}| \geq 0$).

Logically, when a web object is published to the web, a CREATE web-event

[1] This assumes that the last-modified-date of a specific URL can be determined and that it increases monotonically with each web event.

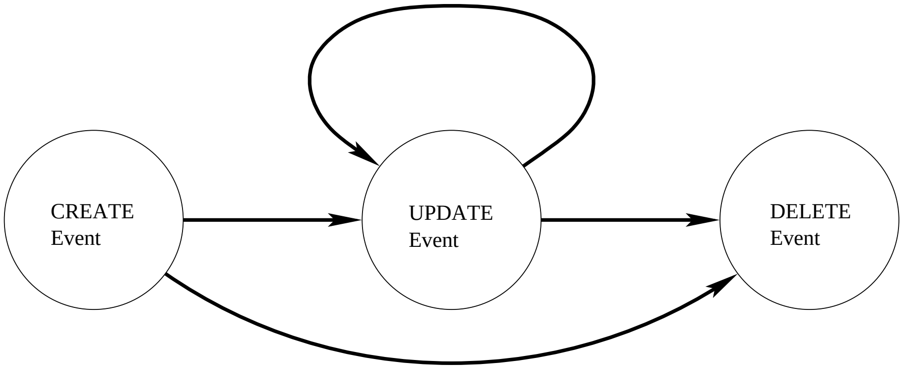


Figure 3.1: The NFA state machine of web-event generation for any web object's lifecycle.

occurs. When an already published object is modified in any way, then an UPDATE event is generated. Finally, when the object is removed permanently from the web, a DELETE web-event is generated. These events are captured in the state of the object; though one must observe a state *change* to accurately detect an event. Such changes could easily be missed by a polling web crawler if the event rate is greater than the poll rate of the crawler. As a result, a crawler may develop inconsistencies between the actual state of the web and its perceived state of the web.

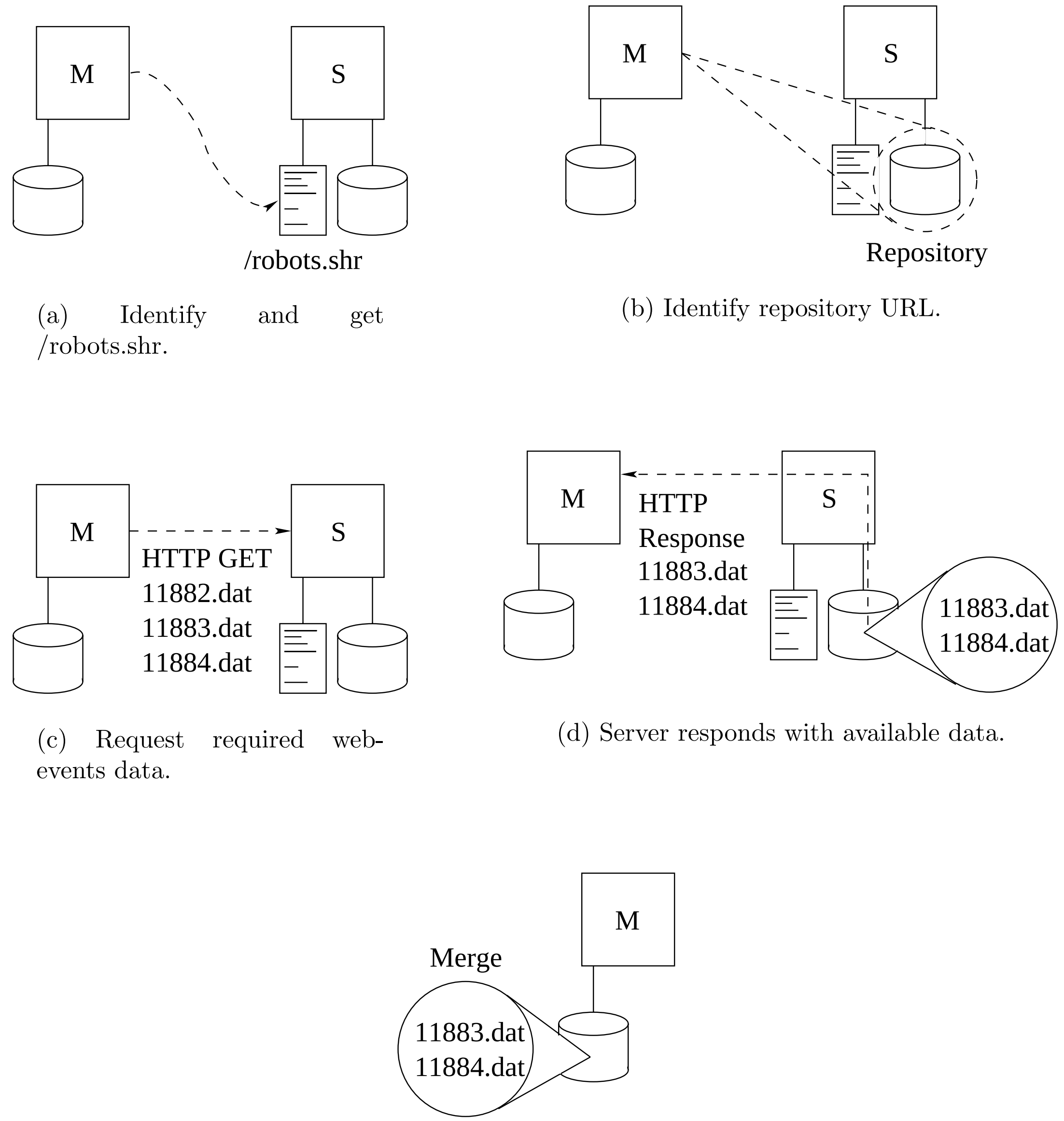
Presented in the remainder of this chapter is the specification for a two-part protocol: a web-event sharing crawler and a web-event merging crawler. Both parts of the protocol hinges on the *share-control file*. This protocol does not dictate use of the shared data; it is not even necessary for a crawler to share their own data. Both the sharing and the merging specifications are independent operations, though their specifications are intrinsically intertwined.

3.2 Information Sharing and Merging

A pair of web crawlers adhering to the data-sharing-protocol is shown in figure 3.2 performing a typical request-response transaction. Figure 3.2(a) requires the merging crawler (M) to request the *share-control file* from the sharing crawler (S). The share-control file contains information that identifies the web crawler, as well as specific details required by crawler M to successfully merge web-event data from crawler S. In figure 3.2(b), web crawler M identifies the *share-repository*: this repository is a key aspect about crawler S describing where web-event data is stored. Figure 3.2(c) has crawler M computing which web-event data files must be obtained from crawler S and retrieving them (in this case, three files called `11882.dat`, `11883.dat`, and `11884.dat` are requested). This computation is based upon the current date, and the date of the previous merge with crawler S, if any. All file transfers are based on HTTP GET operations. Figure 3.2(d) has web crawler S's web server transferring two of the three requested files (`11883.dat` and `11884.dat`) to crawler M. Figure 3.2(e) shows web crawler M merging the contents of these files to update it's view of the web.

3.2.1 Share-control File

The share-control file is the enabling component of the protocol. This file is labeled `robots.shr` and is accessible from the root of a web crawler's web server (eg. `http://www.foo.com/robots.shr`). The content of this file provides information about the serving web crawler, as well as how the web crawler shares its information. No meta-data information is stored in this file. This file is engineered to be easy to parse: it uses the standard `field:value` syntax used by both HTTP[2] and the Robots Exclusion Standard[27]. The available fields are described below.



(a) Identify and get /robots.shr.

(b) Identify repository URL.

(c) Request required web-events data.

(d) Server responds with available data.

(e) Merge web-event data with local repository.

Figure 3.2: A typical, step-by-step request-response scenario between two data-sharing-protocol enabled web crawlers (M and S).

contact The contact information for this crawler is optional. This should be an email address of the human operator of this crawler. This field should have the same value as used by the crawler for its HTTP FROM field-header (as defined in section 10.8 of RFC 1945[2]).

crawler The identification string for this crawler is required. It is composed of the host/IP for locating this crawler, the port on which the web server serving the `/robots.shr` file listens and a crawler moniker. The moniker is the same string as used by this crawler for its HTTP USER-AGENT field-header value (as defined in section 10.15 of RFC 1945[2]).

repository The repository field is required. It is a URL path used to locate where web-event data files are stored.

special A crawler can optionally advertise a set of URLs for which it is especially proficient at crawling. Local web servers are prime specialty candidates, as well as any web servers attached on a high-speed, low-latency network. Peers can choose not to crawl URLs that fall within this crawler's special set, and instead choose to merge their shared data. There can be more than one *special* field in the `/robots.shr` file, and their field values are cumulative.

version The required version string indicates that this crawler uses a specific protocol version. All other peers must use the specific version if possible.

An example `/robots.shr` is shown in figure 3.3.

```
version: 1.0
crawler: plg2.math.uwaterloo.ca:33433 WaterlooCrawler/1.0Beta/PLG
contact: glmclear@uwaterloo.ca
special: http://129.97.224.77/
special: http://plg2.math.uwaterloo.ca/
special: http://www.math.uwaterloo.ca/
repository: http://plg2.math.uwaterloo.ca/share-dir/
```

Figure 3.3: A sample `/robots.shr` file. This example illustrates the use of each of the fields.

3.2.2 Web-Event Data Files

Each web-event data file consists of a series of records separated by blank lines. All records consist of a series of lines laid out in easily-parseable `field:value` pairs. A record represents meta-data about a specific URL, which presumably, has been identified by a crawler as being affected by a web-event. The meta-data records compose the actual data content transferred between crawlers S and M in figure 3.2(d). No web object content is ever transferred between sharing/merging crawlers; it is the responsibility of crawler M to later retrieve the content from the server on which the web object identified by a web-event resides.

In an attempt to reduce the ratio of descriptive information to actual information, the field names are purposely truncated or abbreviated.

cid The crawler identification is required. The format for the *cid* is the same as the crawler field value defined in the `/robots.shr` file above. It is used to identify the crawler that first identified this web-event.

size The size of the web object in bytes is required.

lmd The last-modified-date of the object is required. It is a sequence of digits representing the number of seconds since Jan. 1/1970 GMT when the object was last updated.

lpd The last-polled-date of the object is required. It is a sequence of digits representing the number of seconds since Jan. 1/1970 GMT when the object was last polled by the crawler identified in the *cid* field (could be itself or a peer).

stat The web-event status flag is required. The flag is one character denoting the classification of the web-event corresponding to this record. The valid values are `C`, `U`, `D` for CREATE, UPDATE, and DELETE, respectively.

ttl The time-to-live is the number of seconds after the last-polled-date (LPD) when this entry is no longer valid. This field is optional, and may be used if the crawler associated with this entry is unsure whether it will be visiting the web object again, or if it is relatively sure that no changes will occur to the web object during that time. Merging crawlers can use this value to effectively schedule revisitation of either the URL or future merging operations to the crawler sharing this data.

url The specific URL associated with the recorded web-event meta-data. This field is required.

An example web-event data file containing web-events is shown in figure 3.4. The *cid* field is used to identify the web crawler that reported the original web event data. This is useful if a merging web crawler wishes to use the field to identify which peer crawlers would have the most up-to-date meta-information about a URL.

```
url:  http://plg.uwaterloo.ca/plg.html
size:  9824
lmd:  993226666
lpd:  1008242754
cid:  plg2.math.uwaterloo.ca:33433 WaterlooCrawler/1.0Beta/PLG
stat:  U

url:  http://www.google.com/index.html
size:  2332
lmd:  1011415503
lpd:  1011415503
cid:  129.97.224.77:7777 SharingRobot/1.0
ttl:  1209600
stat:  C
```

Figure 3.4: A sample web-event data file.

3.2.3 Share-repository

All web-event data files are stored in the share-repository. Merging web crawlers expect to be able to request a specific file from the sharing web crawler using standard HTTP requests. Physically storing the web-event meta-data is implementation-dependent, as long as the external interface is perceived as an accessible file.

All web-event data files are referenced using a specific naming convention to which both crawlers M and S must adhere. The file name prefix is a sequence of digits followed by the suffix `.dat`. The sequence of digits prefixing the file extension is the number of *days* since Jan. 1/1970 GMT (call this day D). All web-event records contained in this file must have been discovered (ie. have a last-polled-date occurring) between $00 : 00 : 00.\overline{0}$ GMT and $23 : 59 : 59.\overline{9}$ GMT on day D. This addressing method is similar to that proposed by Brandman, *et al.* [5].

Using figure 3.2 as a reference and the share-repository location information stated in figure 3.3, the HTTP request header from crawler M to crawler S (located at `http://plg2.math.uwaterloo.ca:33433` according to the URL in figure 3.3) would look like:

```
GET /share-dir/11883.dat HTTP/1.0<CR><LF>
<CR><LF>
```

(Crawler M wants to get the web-event data from crawler S for the date of July 15, 2002: $11883\,days * 86400\,sec/day$, is the number of seconds since Jan. 1/1970, which can be converted to a date.)

Crawler S would proceed to serve the requested file if it existed. Note that in, figure 3.2(d), file `11882.dat` did not get returned to crawler M from crawler S. Assuming no errors, this could imply one of two thing: (1) crawler S did not have any knowledge of web-events occurring on day 11882, or (2) file `11882.dat` was removed due to space considerations. Crawler M has no choice but to accept that no data may be available for that day.

3.3 Specific Protocol Issues

3.3.1 Event Resolution Rules

When multiple, independent crawlers share data with one another, it is possible for crawlers to have varying views of the web. Some may have high-quality snapshots of a portion of the web, and others may have lower-quality approximations of a portion of the web. Crawlers that overlap may have recorded different events depending on a number of factors.

If a web crawler M merges web-event meta-data from crawler S and has no prior record of the URL associated with the web-event meta-data, then crawler M simply mirrors the web-event and modifies the *cid* field for this meta-data to crawler S.

On the other hand, if crawler M has prior knowledge of the URL associated with the web-event meta-data, then crawler M must resolve the events according to a set of logical rules[2].

When trying to determine recency, the last-modified-date of an event is used.

- In the normal course of sharing, all DELETE events occur after UPDATE events; all UPDATE events occur after CREATE events. This sequence must be adhered to. Any deviation indicates an error, and the crawler should attempt to investigate by scheduling a network poll of the URL directly.

- If a URL has two different CREATE events according to crawler M and crawler S, then the least recent event is assumed to be the most accurate creation date. The least-recent CREATE event can be *discarded*, and the most recent CREATE event is assumed to reflect an *update*. It should therefore be recorded and reclassified as an UPDATE event.

- If a URL has two different UPDATE events between crawler M and crawler S, then the most recent is recorded and all others can be discarded.

- If a URL has two different DELETE events between crawler M and crawler S, then the least recent event is assume to be the most accurate deletion date.

- If a URL recorded in crawler M has an UPDATE event and crawler S has a *more recent* CREATE event, then S's CREATE event is assumed to be an invalid detection of an *update* to the URL. The CREATE event should be merged from M, but reclassified as an UPDATE event.

[2] These rules are based on the assumption that all data shared by crawlers is genuine. See section 7.2.2 for security concerns regarding misleading or incorrect meta-data.

Chapter 4

Theory

Current-day web crawlers can be easily modified to take advantage of cooperative sharing behaviour. It is important to be able to quantitatively validate cooperative sharing as compared to current-day behaviour. We use the concept of freshness, previously introduced in the Introduction, to measure a crawler's performance. The chapter completes with a brief analysis of simple interaction between cooperating web crawlers is done from an intuitive perspective.

4.1 A Simple Sharing Crawler

A simple web crawler capable of traversing the web as well as communicating shared data to and from peers is shown in algorithm 4.1. This, in turn, is based on the basic design of a web crawler (algorithm 2.1).

Our simple cooperating web crawler uses a value known as ρ to enable it to switch between polling a web object and merging web-event data from peers. The value ρ can take on any real value in $[0 \ldots 1]$. In the extreme cases, when $\rho = 0$, the crawler will only merge web-event data and when $\rho = 1$, the crawler will only poll web objects. Any value $0 < \rho < 1$ represents the ability to do both operations.

The condition in lines 2 and 3 of algorithm 4.1 implies that over time, ρ represents a percentage of time devoted to polling; conversely, $1 - \rho$ represents the percentage of time devoted to merging.

In addition to ρ, a crawler can choose whether to make public any of its web-event data by enabling the *sharing* variable (lines 7 and 14 of algorithm 4.1).

Algorithm 4.1 A simple sharing/merging web crawler.

```
Require: $W$ is a set of web objects representing the entire web
Require: $C$ is a set of all web crawlers on the web including this one
Require: $E_c$ is a cache (a set) of web-events to be made public
Require: $E'$ is a set of web-events independent of $E_c$
Require: $Q_w$ is a queue initialized as a random permutation of $W$
Require: $Q_c$ is a queue initialized as a random permutation of $C$
 1: for ever do
 2:   $P \leftarrow$ uniform-random-value
 3:   if $P \leq \rho$ then
 4:     $w \leftarrow \text{head}(Q_w)$              {$w$ is a specific web object}
 5:     $e \leftarrow \text{visit}(w)$               {get a web-event $e$ about $w$}
 6:     reinsert($Q_w$, $w$)                         {$w$ gets put back into rotation}
 7:     if sharing then
 8:       $E_c \leftarrow E_c \cup e$                {store the web-event to be merged by others}
 9:     end if
10:   else
11:     $c \leftarrow \text{head}(Q_c)$              {$c$ is a specific web crawler other than this one}
12:     $E' \leftarrow \text{visit}(c)$              {$E'$ is a set of events merged from $c$}
13:     reinsert($Q_c$, $c$)                         {$c$ gets put back into rotation}
14:     if sharing then
15:       $E_c \leftarrow E_c \cup E'$               {store the web-events to be merged by others}
16:     end if
17:   end if
18: end for
```

Given ρ and *sharing*, a crawler implementing a variant of algorithm 4.1 could operate in any of six potential capacities:

1. Current-day behaviour: poll the web without regard for the behaviour of peers ($\rho = 1$, $sharing = false$)
2. Altruistic behaviour: poll the web and share all information without ever merging from peers ($\rho = 1$, $sharing = true$)
3. Parasitic behaviour: always merge from peers (never poll the web directly) and never share with peers ($\rho = 0$, $sharing = false$)
4. Mirroring behaviour: always merge from peers (never poll the web directly) but share everything ($\rho = 0$, $sharing = true$)
5. Non-sharing hybrid: poll the web sometimes and merge from peers sometimes, but never share any information ($0 < \rho < 1$, $sharing = false$)
6. Sharing hybrid: poll the web sometimes and merge from peers sometimes, sharing any information ($0 < \rho < 1$, $sharing = true$)

It is hypothesized that web crawlers that are cooperative (ie. if $0 < \rho < 1$ or *sharing* is enabled) can mutually benefit from sharing web-event data more so than crawlers operating in a traditional current-day role. Such benefit can be quantified by measuring a web crawler's cache freshness.

4.2 Freshness as a Metric

Crawler freshness is the primary measure of the performance. As can be seen from algorithm 4.1, a crawler maintains a cache (E_c) which is a set of web-event data records for associated web objects that have either been polled directly by the crawler or merged from a peer. An individual entry in the cache is denoted as e. A transformation function ($\mathcal{T}$) is used to convert a specific web object w into the associated web-event data record e when polled by a crawler c at time t: $e \in E_c$, $e = \mathcal{T}_t(w)$. *Freshness* of a single cache entry is a binary measure, determining, if at time t, the cache entry e is synchronized (state-equivalent) with its associated web object w [11]:

$$F(e;\, t) = \begin{cases} 1 & if\, e \equiv \mathcal{T}_t(w) \\ 0 & else \end{cases} \tag{4.1}$$

State-equivalence between two web-event data records means the relationship $\mathcal{T}_i(w) \equiv \mathcal{T}_j(w)$ holds for times $i < j$. Note that the nature of web-events implies that if $\mathcal{T}_i(w) \equiv \mathcal{T}_k(w)$ for some time $i < k < j$ such that a web-event occurs on object w at time j, then $\mathcal{T}_i(w) \neq \mathcal{T}_j(w)$. This is because web objects only maintain the current state; once a web-event occurs, any previous state information is lost.

Note that we ignore that the definition given by equation 4.1 assumes the ability to perform instantaneous comparisons between a cached web-event data record and the current state of the associated web object. Such instantaneous comparisons are usually not possible when dealing with the web.

The definition of freshness for a specific crawler's cache (E_c) at time t is an average of the freshness for the individual web-event records stored in E_c at that time:

$$F(E_c;\, t) = \frac{1}{|E_c|} \sum_{e \in E_c} F(e;\, t)$$

Of course, $F(E_c;\, t)$ is only useful as a measure of the instantaneous freshness of the cache E_c. We wish to observe the freshness of a crawler's cache as it *changes over time*. The freshness of E_c over time is intuitively a time-based average:

$$\overline{F}(E_c;\, t) = \begin{matrix} lim \\ t \to \infty \end{matrix} \; \frac{1}{t} \int_0^t F(E_c;\, t)\, dt$$

4.3 Algorithm Analysis

Algorithm 4.1 can intuitively be reasoned to show that there exists a point whereby a combination of polling and merging can yield more web-event data than simple polling.

Presumably, a crawler can schedule when to merge with another crawler. However, if it merges too soon, not enough data will be amassed by the peer to make the merging operation beneficial. If it waits too long, then the crawler runs the risk of doing too much polling and rendering the peer's potentially-shared web-event data records redundant. The strategy used by a crawler to wait "just the right amount of time" is a complex process. Values for ρ cannot be easily analyzed by mathematics. Several complexities in the potential interactions between peers quickly make a more in-depth analysis intractable. Simulations developed in the next chapter are used to support the intuitive evidence that combinations of polling and merging can produce better freshness among mutual cooperative crawling systems than if individual crawlers decided to crawl on their own.

4.3.1 Preamble to the Analysis

We will assume that the web (W) is sufficiently large such that it can be close to infinite in size. Over time, web-events will occur randomly to web objects in the web (W). The dynamics of the physical web have been studied and shown that web-events can be modeled after a Poisson process[10, 11, 38]:

$$Prob(x\,events) = \frac{(\alpha(t_j - t_i))^x \, e^{-\alpha(t_j - t_i)}}{x!}$$

where α is the mean number of web-events that occur during a unit time, and $t_j - t_i$ is the time-interval being examined.

For the purposes of this analysis, we will assume that web-events occur with a rate such that each poll will yield one web-event. This does not imply that web-events are not missed – merely that at least one web-event will occur to a web object w between subsequent polls to that web object by a *specific* web crawler. It will also assume that web crawlers will not poll the same object at the same time. Together these assumptions imply that only one crawler can ever have the most-recent web-event for a specific web object.

Crawlers using algorithm 4.1 will select specific web objects from their crawling set Q_w, and specific peers from Q_c in a uniform random fashion.

We will assume that sharing and merging take the same time as a single network poll. This is a gross assumption, since, of course, transferring large amounts of data over a network can incur more time than sending a single byte.

Finally, we will assume that a crawler has a vested interest in knowing about the entire web.

4.3.2 Intuition of Interaction

If the number of crawlers in the system were, in fact, *zero*, then the web would never get crawled and vital events would be lost. Of course, this means that at least one crawler is required ($C_1 \in C$). The single crawler must crawl the entire web ($Q_w \equiv W$). Since there are no peers with which to merge, C_1 will implicitly have a value of $\rho = 1$. This implies that crawler C_1 will perform $|W|$ network polls, for a total number of $|W|$ polling operations. At most one web-event per poll can be gleaned, resulting in an information gain to network operation ratio of $\frac{|W|}{|W|} = 1$.

This crawler will not necessarily have the most up-to-date cache (E_c) because polling the entire web is a slow process. Many web events may be missed.

When another cooperating crawler (C_2) is added to C, each crawler can poll between 1 and $|W|$ web objects and then share (a single operation) between 1 and $|W|$ web-event records with their peer. The longer a crawler waits to merge, the more web events can be transferred (until a point). The length of time between merges is controlled by a crawler's value of ρ. Mutually, the best case occurs when each crawler crawls about $\frac{|W|}{2}$ objects and then performs a merge operation with its peer. This results in a total of $|W| + 2$ operations over both crawlers for an information gain to network operation ratio of $\frac{|W|}{\frac{|W|}{2}+1} > 1$ per crawler. This will not happen if ρ is either 0 or 1 (since either extreme results in no information being shared and results in $|W|$ operations per crawler).

If we add yet another web crawler (C_3) to the process, then each crawler is capable of polling between 1 and $|W|$ web objects and then sharing between 1 and $|W|$ web-events with another peer. Again, a crawler's value of ρ dictates how long to wait before merging with another crawler. If every crawler decides to do the same amount of work (ie. each polls $\frac{|W|}{3}$ web objects), then a single crawler can merge from both peers to receive $\frac{2|W|}{3}$ web-event data records in only two operations. The total number of operations over all 3 crawlers is $3\left(\frac{|W|}{3}+2\right)$ and the information gain to network operation ratio of $\frac{|W|}{\frac{|W|}{3}+2} > 1$ per crawler.

Each crawler does the same amount of work, and gets three times the data for it's effort. If we attempt to generalize this, we may assume that an optimal crawling distribution is for all crawlers in C to poll:

$$|Q_w| = \frac{|W|}{|C|} \tag{4.2}$$

web objects before attempting merging operations which can yield the remaining web-event data records.

Note that as membership in C increases, the size of an individual crawler's Q_w decreases, until the point where there are $|W|$ crawlers and each crawler $C_i \in C$ crawls only one object. At this point, *each* crawler is performing one poll operation followed by $|W|-1$ merges of all other crawlers to get their single web-event record (for a total operation count of $|W|^2$ over all $|C| = |W|$ crawlers). Per crawler, the information gain to network operation ratio of $\frac{|W|}{|W|} = 1$, which is no better than the case when $|C| = 1$.

Both $|C| = 1$ and $|C| = |W|$ illustrate the limiting scenarios in equal-work cooperative web-crawling behaviour. Since the behaviour for $|C| = 2$ exhibits an information gain greater than either of the limiting scenarios, there must be an optimal number of web crawlers.

4.3.3 Deriving an optimal number of crawlers

An individual crawler's information gain compared to the number of operations required to receive knowledge about all of W can be expressed as a function of the size of the community of crawlers in equation 4.3.

$$f(|C|) = \frac{|W|}{|Q_w| + |C| - 1} \tag{4.3}$$

In other words, a single crawler in a community of $|C|$ crawlers must poll $|Q_w|$ web objects, and then merge from the remaining $|C|-1$ peers to get all $|W|$ objects. Discovering the local extrema of $f(|C|)$ is as simple as solving for, and minimizing $f^{'}(|C|)$ (making sure to substitute appropriately for $|Q_w|$):

$$f^{'}(|C|) = \frac{-|W| \cdot \left(1 - \frac{|W|}{|C|^2}\right)}{\left(\frac{|W|}{|C|} + |C| - 1\right)^2} \tag{4.4}$$

The minimization of $f^{'}(|C|)$ yields $|C| = \pm\sqrt{|W|}$; however, only the positive solution makes sense. This represents the local/global minimum. For a web of size 10^9 in a non-contention environment, the least number of equal-work, cooperating web crawlers needed to achieve optimal freshness is $|C| \approx 31623$, each of which crawls $|C|$ different web objects. If there is no contention for communication resources, then this means that for any size of crawler community in which $|C| > \sqrt{|W|}$, the remaining crawlers ($\sqrt{|W|} - |C|$) should simply merge from all other crawlers rather than poll. In this way, all crawlers will achieve a maximal freshness. Determining which crawlers should poll and which should merge remains a problem.

4.3.4 Number of Operations as Related to Freshness

The above analysis deals with the number of operations that could occur when cooperating crawlers interact. Operation-counting does not translate directly into a measure of freshness. However, it is a good indicator, especially when one takes into account the fact that merges are not, in fact, a free operation. A merging operation can be considered *cheaper* than a network poll, because it is possible to get more than one web-event data record per merge, as compared to being able to get at **most** one record per network poll.

Chapter 5

Simulation Software

A web crawling simulator was developed to quickly collect data while being in full control over crawling parameters. The web crawler can be configured to crawl a virtual web using a variety of strategies. The simulator is able to omnisciently measure various metrics of any web crawler and the virtual web at any point in time. Our primary interest is in how the freshness of a crawler's cache can change over time, especially when interacting with peers to share or merge data. Our secondary interest is in how crawlers can mutually affect the freshness of their own cache and the cache of others as they compete for resources.

5.1 Simulation Architecture

The simulation uses a freely available simulator package known as YACSIM[1]. Yet Another CSIM [C-Simulator] is an event- and process-oriented simulator imple-

[1] Available for download from http://www.crpc.rice.edu/softlib/rppt.html

mented as a set of library calls for the C programming language. The GNU Scientific Library[2] provides the necessary random-number generator and random-number distribution functions.

Three objects are represented within the simulation environment: web objects, web crawlers, and the virtual web (network). The network is simply a passive, observable entity, whereas web objects and web crawlers are active participants. The crawler is the most sophisticated of all of the simulated objects and accounts for nearly all of the work.

The architecture of the simulation is shown in figure 5.1. All of the simulated objects have resources associated with them. Simulation resources use a multi-processor-sharing contention model based on a uni-processor model supplied by YACSIM: if there are k simulation objects requiring service from a multi-processor-sharing resource that has s servicing slots, then each object will obtain an amount of service proportional to $\frac{s}{k}$. If $s > k$, then all k objects will receive full use of a processor (ie. no contention).

5.2 Web Objects

A web object is a uniquely-identified entity within the simulated web. It possesses the basic properties of file size (in bytes) and a last-modified-date.

Web objects can be polled and retrieved in the simulation, but both operations have a cost associated with them. The cost is realized by the time to transfer the data (which is directly related to the *bandwidth* associated with the requester and the object).

[2] Available for download from http://www.gnu.org/software/gsl/gsl.html

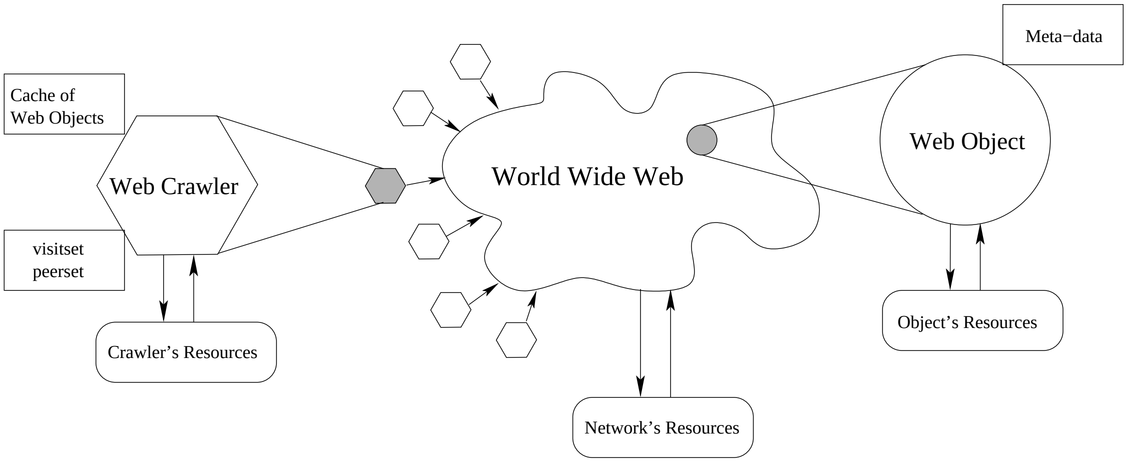


Figure 5.1: Simulation architecture.

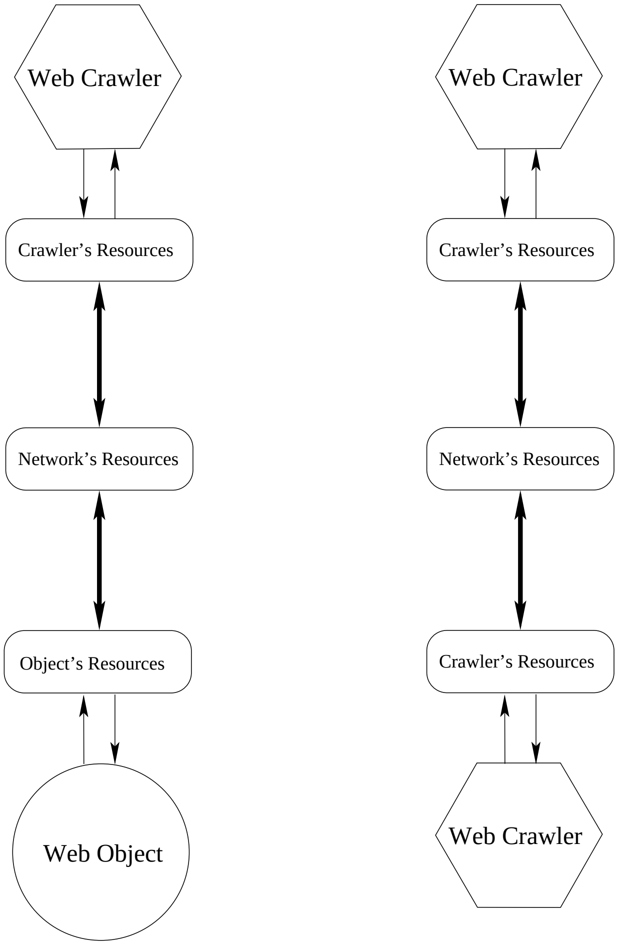
All web objects evolve over time by periodically generating web-events via a YACSIM event process. The standard lifecycle for any web object follows the simple state-machine illustrated in figure 3.1.

All events are exponentially-distributed. CREATE web-events are generated by instantiating new web objects at a rate λ_C. UPDATE web-events are generated at a rate λ_U. Web objects may delete themselves with a Poisson distribution probability using an event rate of λ_D. If $\lambda_C = \lambda_D$, then the size of the virtual web remains approximately constant over time.

5.3 Web Crawlers

Web crawlers have a number of important structures and algorithms associated with them. Crawlers essentially consist of the following pieces:

- A queue of web objects to poll (*visitset*)
- A queue of web crawler peers to contact (*peerset*)



(a) Crawlers communicate with web objects via the shared network resource.

(b) Crawlers communicate with other peers via the shared network resource as well.

Figure 5.2: Simulation communication and contention model between crawlers and objects and crawlers and crawlers.

- A cache to store web object meta-data polled by the crawler or obtained from a peer
- ρ - a variable used to determine how often this crawler should poll and merge; it is used in algorithm 4.1

Web objects that have been known to be deleted are removed from the *visitset*. The decision to poll the network or merge records from another crawler is determined by crawler's value of ρ.

It should be noted that crawlers are assumed not to suffer access errors as a result of communicating with either web objects or other web crawlers.

5.4 The Simulated Web

The web consists of nothing more than a set of web objects and web crawlers. The notion of a web server, prevalent in the actual web, has been removed in this simulation. As illustrated in figure 5.2, the network is a shared resource used by all web crawlers and web objects. Directly influencing the cost of the network access is the network bandwidth.

5.5 Simulation Variables

Each of the variables alluded to in the previous sections are described in more detail. In each case, the set of legal values and/or selection method and criteria is described.

The bandwidth associated with the link from a web object to the network cloud as shown in figure 5.2(a) is described using a uniformly chosen random variable B_{ob}. It can range from 5 KBps to 200 KBps. Note that it is possible for an object to be unreachable by a crawler if the object's bandwidth is set to 0.

The bandwidth associated with the link from a web crawler to the network cloud as shown in figure 5.2(b) is described using a uniformly chosen random variable B_{cr}. B_{cr} is chosen from 5 KBps to 200 KBps. A crawler can be considered unreachable if its bandwidth is set to 0.

The network bandwidth is fixed at 1.5 Mbps.

5.6 Web Crawler Algorithms

Algorithm 5.1 follows quite closely to algorithm 4.1. Each network access to a web object costs some time, which is clearly charged against the crawler regardless of the outcome of the polling operation (updated or not). When one crawler attempts to merge from a target crawler, both crawlers will incur the communications cost, since they are both involved.

Line 7 of algorithm 5.1 shows the crawler's cache being updated in the event a poll yields a new web-event.

Lines 14 to 15 represent the local crawler attempting to determine the necessary information from the remote crawler's `/robots.shr` file. Only if this succeeds can a merge occur.

Lines 17 to 19 show how a single merging operation could potentially yields several web-events being added to the crawler's cache.

Algorithm 5.1 `Crawler :: algorithm()`

```
Require: $visitset$ is a structure containing the set of all URLs that will be visited.
Require: $peerset$ is a structure containing the set of all peers that will be contacted.
loop
  $rr \leftarrow$ random()
  if $rr \leq \rho$ then
    $URL \leftarrow visitset$.pop()          {get the head URL from the queue}
    $obj \leftarrow$ poll($URL$)
    if $obj \neq \emptyset$ then
      $cache \leftarrow cache \cup obj$
    end if
    if $obj$.event.type $\neq$ DELETE then
      $visitset$.push($URL$)
    end if
  else
    $peer \leftarrow peerset$.pop()          {get the next crawler to merge from}
    validate-peer($peer$)
    if peer-is-valid then
      $eventset \leftarrow$ merge-from($peer$)
      for $x \in eventset$ do
        $cache \leftarrow cache \cup x$
      end for
    end if
  end if
end loop
```

Algorithm 5.2 `Crawler :: poll()` from line 5 of algorithm 5.1.

```
Require: $URL$ is a URL for an object that may or may not exist on the physical web.
1: $obj \leftarrow$ get-network-state($URL$)
2: $cobj \leftarrow cache \cap obj$ {Determine if we already have previous knowledge about $URL$ or not}
3: if $cobj = \emptyset$ then
4:    if $obj$.event.type $\neq$ DELETE then
5:       return $obj$                          {detected a CREATE web-event to object}
6:    end if
7: else
8:    if $obj$.event.type $\neq$ DELETE then
9:       if $obj$.last-modified-date $>$ $cobj$.last-modified-date then
10:         return $obj$                       {detected an UPDATE web-event to object}
11:      end if
12:   else
13:      return $obj$                          {detected a DELETE web-event to object}
14:   end if
15: end if
```

The polling algorithm described in algorithm 5.2 has three branches of interest. If a web object has never been seen by a crawler before (line 3), then the crawler assumes that the web object has just been created.

If the web object is already known to the polling crawler, then it can use the difference in information retrieved from this poll and the previous record to determine if a change has occurred. Note that if the crawler detects that an object has been deleted, then it will remove it from the polling queue[3].

The merging process outlined in algorithm 5.3 shows the two-stage merging process. In order for a crawler to merge data from a target crawler, it must first check

[3] The idea that something has been permanently deleted is somewhat of a controversy, since traces of the content may be cached or preserved. The Internet Archive's Wayback Machine (http://www.archive.com) is an example of object preservation.

Algorithm 5.3 `Crawler :: merge-from()` from line 16 in algorithm 5.1.

```
Require: $peer$ is a cooperating web crawler sharing its web-event cache.
$updates \leftarrow \emptyset$                                   {initialize the return set to be empty}
if $peer$ is sharing data then
    $lastvisit \leftarrow peervisitset \cap peer$                {get last time we visited peer}
    if $lastvisit = \emptyset$ then
        $lastvisit \leftarrow$ current-time
        $peervisitset \leftarrow peervisitset \cup lastvisit$    {add new peer to our set}
    end if
    for $x \in peer$.cache s.t. $x$.last-modified-date $\geq lastvisit$ do
        $mycopy \leftarrow cache \cap x$
        if $mycopy \neq \emptyset$ then
            if $mycopy$.last-modified-date $\neq x$.last-modified-date then
                $updates \leftarrow updates \cup x$              {found an updated object on the target}
            end if
        else
            $updates \leftarrow updates \cup x$                  {found a new object on the target}
        end if
    end for
end if
return $updates$
```

to find out when it last visited the crawler. If the crawler has not updated their sharing data file since the last visit, then there is no need to continue. Otherwise, the local merging crawler will requests all available web-events from the remote sharing crawler since the last time the remote crawler was contacted.

Chapter 6

Experiments and Results

The experiments described in this chapter show the effects of various policies that can be adopted by cooperating web crawlers. Specifically, these experiments are designed to illustrate the ways in which ρ can affect the freshness of an individual crawler within a system of $N-1$ peers. The results of these exploratory experiments can be used to dictate the direction of more dynamic strategies in which an individual crawler can monitor it's performance and crawling environment and adjust ρ accordingly. We examine changes in the size of the crawling space as well as changes in the disposition of a crawler (parasitic, mirror, hybrid, etc. as outlined in section 4.1).

6.1 Experimental Setup

All simulations were performed operated under a common set of conditions. The size of the simulated web was set to an initial size of 1×10^6 objects, with a creation growth rate mean of 3476 web objects per day[1]. Conversely, objects were deleted from the simulated web with an identical mean, thus resulting in an approximately static web. The simulation ran over a simulated 180-day (1.5552×10^7 seconds) span.

The network bandwidth was set at 1.5 Mbps, and all crawlers operated with an individual bandwidth of 100 Kbps. In the absence of contention, web crawlers could crawl approximately one object per second. The contention model – the processor-sharing model as described in section 5.1 – was dampened by using a constant $s > 1$. This helps avoid resource contention crippling the performance of a system of web crawlers.

Crawlers started with an entirely fresh cache for the portion of the web they were set to crawl (ie. freshness for a crawler at time 0 was 1.0). Unless otherwise dictated by a crawler's policy, web objects were randomly ordered and selected for crawling; peer crawlers were also randomly ordered and selected for merging.

All crawlers independently operate as a single-thread: no parallelism is expressed or implied by algorithm 4.1. Each poll performed by a crawler was a simulated HTTP HEAD operation. In all cases, the web server associated with an object was assumed to transmit the last-modified-date information for each web object requested. A merging crawler was assumed to operate under the pretense that it retrieved web-events beginning with the most recent. Furthermore, the sharing crawler did not perform any filtering based on a peer's merge-request, and so all pertinent web-events were transmitted to the merging crawler. Note that a crawler that was in the process of merging could not poll, but a crawler in the act of sharing could continue to poll with a penalty applied to their personal network bandwidth.

[1] A constant growth rate of 0.3476% per day for the web is assumed from the growth data described in a study by Cyveillance [32].

6.2 Experiments and Analysis

6.2.1 Establishing a Baseline

Under current world conditions, a set of web crawlers will crawl the entire web, entirely ignoring peers. Using our simulation model, this corresponds to each crawler attempting to crawl the entire web, having a $\rho = 1.0$, and not allowing any peer to merge. The number of crawlers crawling the simulated web was varied at 2^N, $0 \leq N \leq 10$. As the baseline simulation ran, the system of N crawlers stabilized in their cache freshness value. That is, at the end of the simulation, the crawler's cache freshness is indicative of the value that the crawler is capable of achieving. The equilibrium state achieved by all N crawlers in the system can be averaged to show the average crawler cache freshness.

The simulation was run with network contention enabled and disabled allowing for the two curves shown in figure 6.1. The only affecting factor should be contention for network resources among crawlers. An increase of crawlers in the system should elicit a decrease in the freshness of an average crawler's cache.

Although the entire system of N crawlers starts with a cache freshness of 1.0 at time 0, when the simulation ends, the equilibrium state for a system of N crawlers in a non-contending environment ends with an average freshness of about 20% regardless of the size of N. This freshness value arises from various parameters controlling the speed of crawling and the frequency of web-events. Increasing the speed of crawling or decreasing the frequency of web-events would result in an increased average crawler cache freshness. Conversely, decreasing the speed of crawling or increasing the frequency of web-events would result in a decreased average crawler cache freshness.

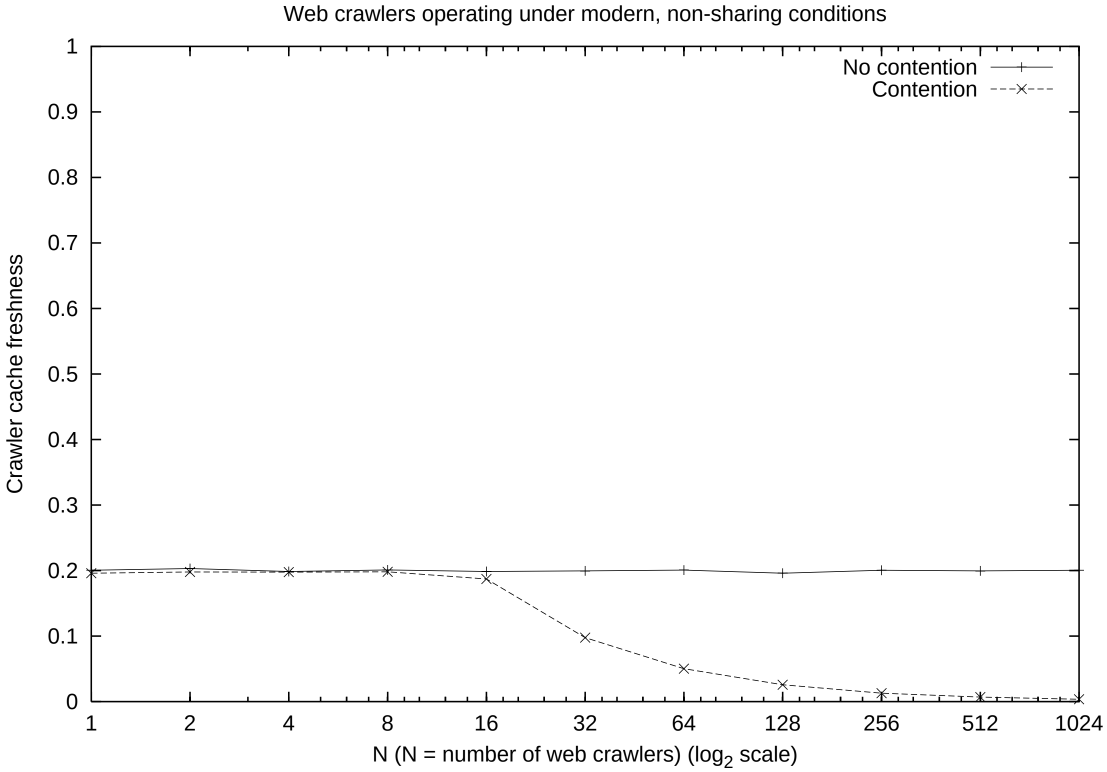


Figure 6.1: Crawlers running under current world conditions (oblivious to peers; no crawler is sharing).

With contention enabled, figure 6.1 shows that, as expected, when N is large enough, resource contention among all crawlers in the system adversely affects the average crawler cache freshness. The rate of decrease in this curve is based on the $\frac{s}{k}$ processor-sharing contention model employed by the simulator. Contention is not immediately noticeable because $s \geq k$ when N is small.

6.2.2 The All-or-Nothing Approach

In order to enable sharing and merging, crawlers must enable their ability to share data and use algorithm 4.1 with $0 < \rho < 1$. If all N crawlers in the system use the same fixed value of ρ, then the crawling system evolves in an interesting way.

Small values of N

Figure 6.2 shows a system of two (2) web crawlers using algorithm 4.1 such that all web crawlers operate with the same value of ρ. A curve denoting the behaviour of the system with and without contention is shown. Each curve represents the average behaviour for all crawlers in the system. The optimal cache freshness is indicated by a mark on each curve. It is at these marked points that a combination of merging and polling yields the best freshness. It would be expected that values of ρ close to 0 and 1 would show a decline in the freshness compared to values of $0 < \rho < 1$. For $N = 2$, the optimum should occur for a value of $\rho \approx 1$ since there is only one other crawler from which to receive web-events. In order to make a merge worthwhile, the crawler should wait until enough information has been amassed by its peer.

The optimum freshness with two sharing crawlers shows a distinct improvement in the average cache freshness: the highest point of each curve in figure 6.2 is above the baseline average cache freshness of figure 6.1.

Two interesting aspects are present in this graph: the relative closeness between the contention and non-contention curves, and the behaviour at the extreme ends of the curves. The relative closeness between the two curves is apparent because the processor-sharing contention model ($\frac{s}{k}$ for k jobs) uses a constant $s > 1$. Contention will be practically non-existent for such a small value of N.

When $\rho = 0$, the freshness is approximately 0 as well. This is to be expected. When all crawlers are trying to merge from one another and no polling is done, the freshness will degrade completely. When ρ increases, a larger percentage of time

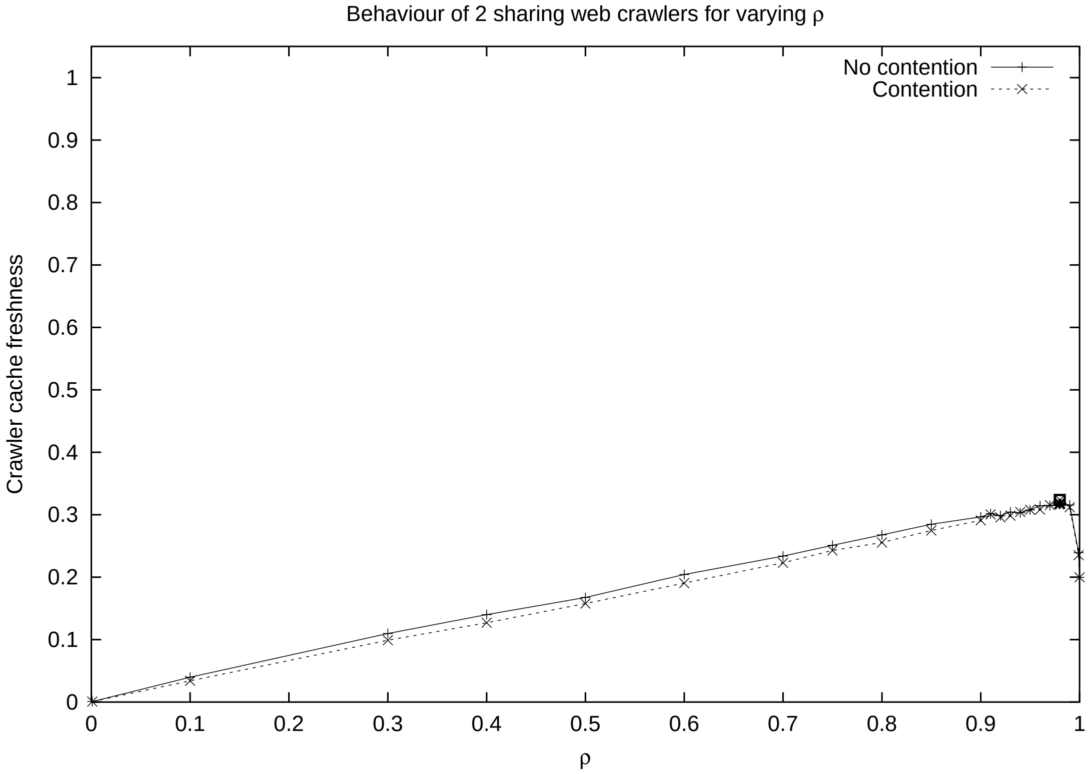


Figure 6.2: Two crawlers share and merge with one another with various values of ρ.

is spent polling – doing real work – and the freshness for each crawler begins to increase. Sharing only becomes a useful operation for medium-high values of ρ. At this point, sharing enables each crawler in the system to achieve a freshness better than that achieved by the baseline in figure 6.1. The optimum point is achieved close to $\rho = 1.0$ as expected. The sudden decline as ρ approaches 1.0 after the optimal values in each curve means that the cache freshness in each respective crawler simulation is actually hindered for values of ρ greater than the optimum. This is expected: the amount of work to retrieve web objects by merging with other crawlers is a bit less than the work required to retrieve them directly since multiple data-records can be merged for the same cost as a single poll.

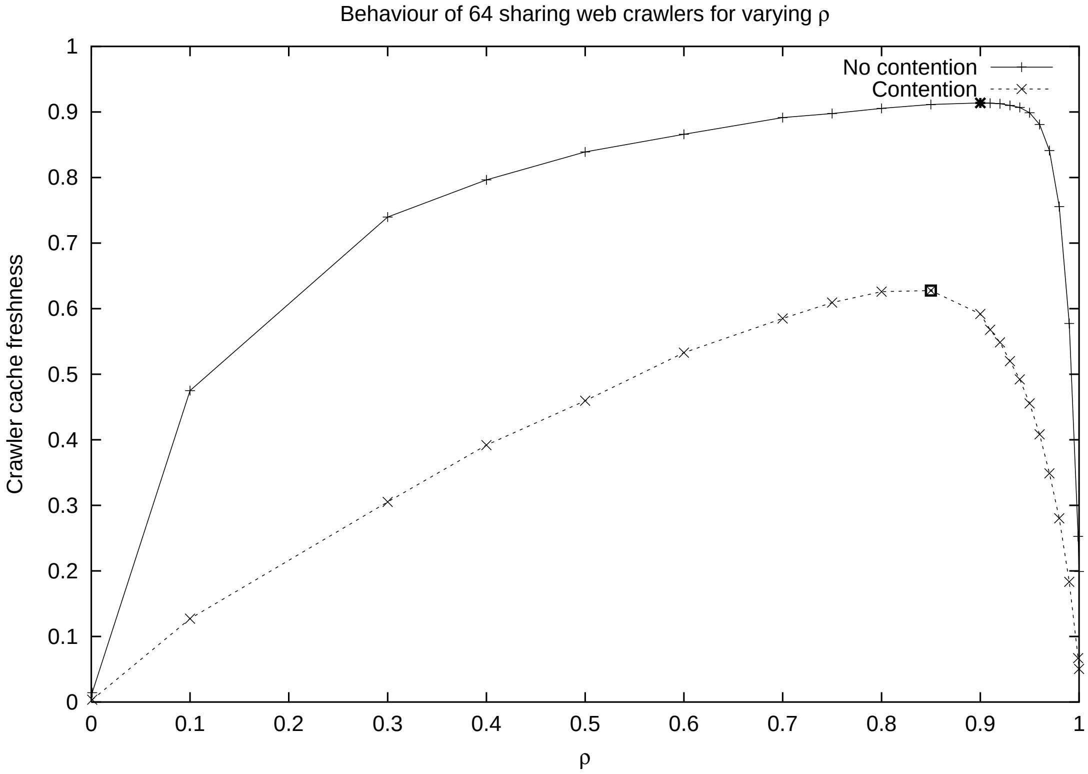


Figure 6.3: 64 crawlers share and merge with one another for various values of ρ.

Larger Values of N

If we increase the number of crawlers in the system without changing any other parameter, then we can see how such an increase in N has a profound effect on the average crawler cache freshness. Figure 6.3 shows a system of 64 web crawlers. All use the same value of ρ for their entire running time. The maximum average crawler cache freshness for the non-contention and contention models is indicated in each. With such a large set of web crawlers, merging should show an increased performance relative to the baseline performance of 0.2. However, contention for resources will also be apparent.

The most obvious feature of figure 6.3 is the large performance gap between the contention and non-contention simulations. The non-contention curve illustrates the behaviour if resource contention were not a factor.

The optimum crawler cache freshness is shown at about $\rho = 0.85$ for the contention curve. This value of ρ is less than the value of ρ for the optimum when $N = 2$ in figure 6.2. However, the optimal cache freshness is about twice that when $N = 2$, and three times greater than the baseline.

Figure 6.4 shows a system with $N = 256$. Like figure 6.3, we expect to see a significant performance gap due to resource contention.

The performance gap shown in figure 6.4 is actually more pronounced than in figure 6.3. Freshness in the non-contention curve is almost perfect because there are so many crawlers in the system from which to gather information from. The curve then falls sharply when all the crawlers are polling more than 90% of the time, which causes resource contention to access the web objects. The non-contention curve also sports a plateau between $\rho = 0.3$ and $\rho = 0.9$ which illustrates that it is only slightly more advantageous to poll more than half the time when N is quite

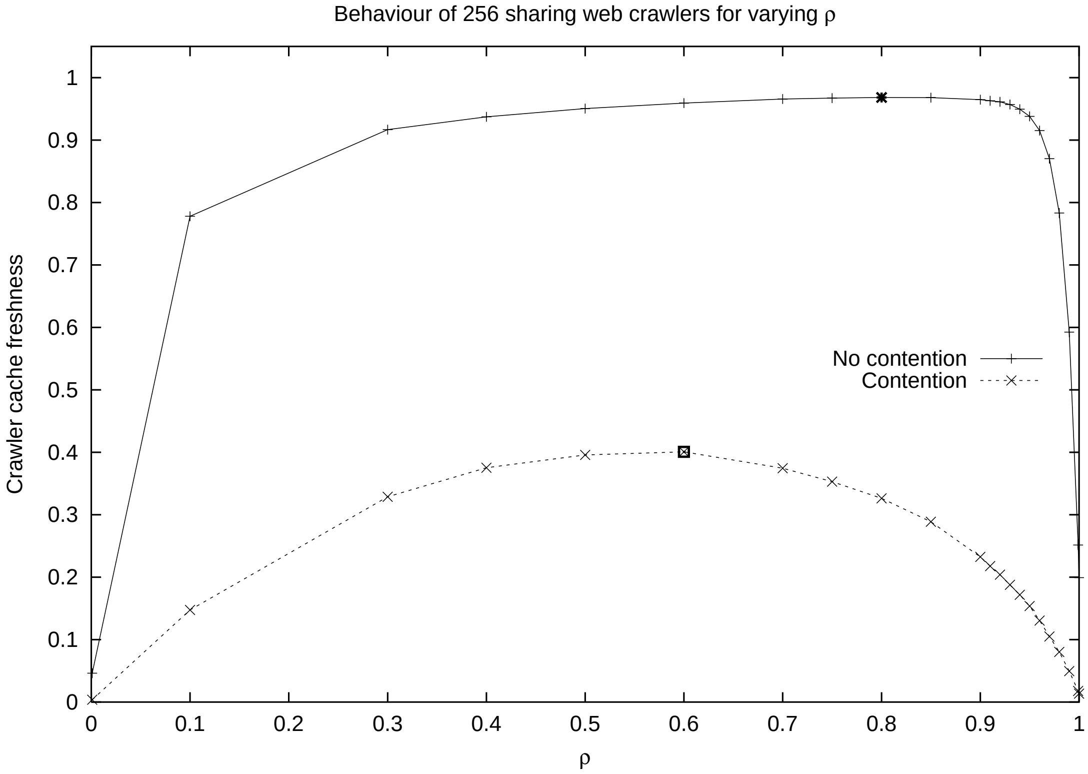


Figure 6.4: 256 crawlers share and merge with one another for various values of ρ..

large.

Figures 6.2, 6.3, and 6.4 examine the role of varying ρ for a set of N crawlers. Each of figures 6.5 and 6.6 shows the associated crawler cache freshness for $\rho = 0.001$, $\rho = 0.30$, $\rho = 0.85$ and $\rho = 1.0$. The value of ρ is fixed for the entire set of crawlers over the duration of a simulation. The figures start at $N = 2$ because a single crawler does not exhibit interesting features.

Figure 6.5 graphs the behaviour when there is no contention. Each simulation shows that freshness tends to increase regardless of the number of crawlers in the system. However, as N increases, smaller values of ρ cause the growth toward a freshness of 1 to slow significantly.

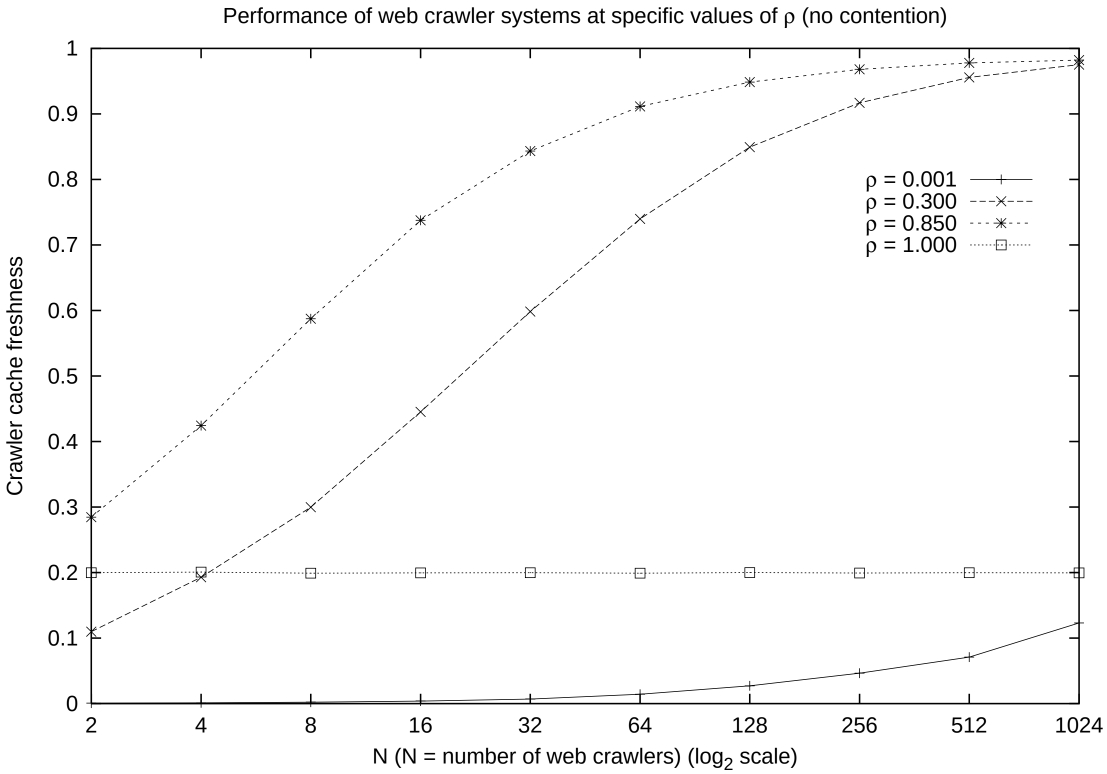


Figure 6.5: Various crawler systems for increasing values of ρ over a set of N non-contending web crawlers.

Figure 6.6 graphs the behaviour when there is contention for resources. This figure shows that the majority of systems observe much better freshness when $\rho = 0.85$ then for any other value of ρ except for when N is very large. When N is very large ($N \geq 2^8$), our simulations show that a low value of ρ (indicating more merging than polling) may achieve better freshness (as shown by the cross-over point between the curves for $\rho = 0.85$ and $\rho = 0.30$).

Finally, figure 6.6 shows that $\rho = 0.30$ has a distinct plateau for $2^4 \leq N \leq 2^8$. This indicates that $\rho = 0.30$ offers similar potential over a wide range of values of N.

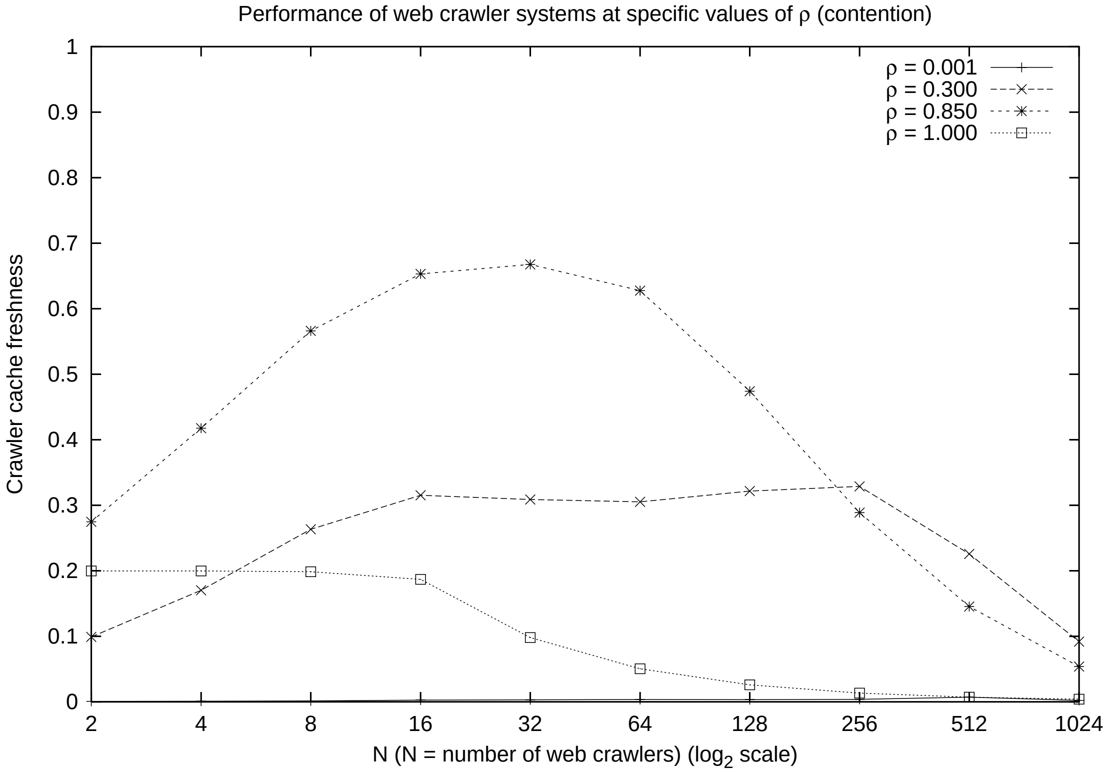


Figure 6.6: Various crawler systems for increasing values of ρ over a set of N contending web crawlers.

6.2.3 Exercising Varying Values of ρ

If all crawlers within a cooperating crawling system are independently initialized with a random value of ρ, then a number of interesting effects can be witnessed. An example system consisting of 64 crawlers is shown in figure 6.7. The data was obtained by running several simulation trials in which each of the $N = 64$ crawlers was assigned a valid uniformly-generated random value for ρ. The mean uniform-distribution is also displayed in figure 6.7 as an example of a "perfect" distribution of values for ρ.

The law of averages states that all crawlers should assume a value of ρ such that each successive pair of values is approximately equidistant from one another. In fact, the perfect equidistant distribution crawling system is overlaid in figure 6.7 to show that this is occurring. Working with a system of $N = 64$ crawlers makes some assumptions about the uniformity of the resulting data that will be discussed below.

Figure 6.7 shows the interesting characteristic that in a 64-crawler system, the freshness is at its best for crawlers that exhibit a ρ between 0 and 0.60, and it is at its worst for a crawler that exhibits a ρ close to 1. However, as figure 6.3 shows, if all crawlers in an $N = 64$ system take on a value of ρ close to 0, the average freshness for a crawler is worse than that shown here. Overall, this graph aims to illustrate the freshness that could be expected if a random value for ρ were assigned to a crawler within a system of $N = 64$ crawlers, each of which is also assigned a random value for ρ.

Curves can be generated for various N similar to that shown in figure 6.7. It is the shape of the curve that is important to know such that we can discern the size and breadth of the plateau section (if any) and how steep the curve increases or decreases (figure 6.7 shows a steep decrease). We apply a least-squares line fit to the data points for each of the curves generated for various N.

Figure 6.8 shows the result of applying a least-squares regression curve to each of the crawling systems for $N \geq 1$. The slope (m) of the regression line $y = mx + b$ is plotted. The magnitude of the slope indicates how the type of value for ρ that should be selected in order to attempt to maximize the freshness of the system. Slopes that are positive indicate that values closer to 1.0 should be selected, whereas negative slopes indicate that values closer to 0 should be selected. Relative differences between the magnitudes are indicative of how close to an extremum ($\rho = 0$ or $\rho = 1$) from which to select: the higher the magnitude, the closer to the extremum, the particular ρ should be.

One interesting feature of figure 6.8 is that when $N = 512$, the magnitude of the slope is less than the magnitude for $N = 256$. Indeed, although not shown, it is expected that as $N \rightarrow \infty$, the magnitude would approach 0 to indicate that when a system is too large, there is no bias toward more polling or more merging. Any value of ρ would achieve the same level of freshness – which is to say, a freshness of about zero.

This analysis fits with previous findings showing that for smaller values of N, a value of ρ favoring polling should be selected – indeed, for $N = 2$ a value of ρ close to 1 should be selected which is suggested by the magnitude and direction of the slope in figure 6.8.

6.2.4 Mirrors and Parasites

If you recall from section 4.1, a crawler can operate in one of six modes. When a crawler acts as a mirror, it uses a value of $\rho = 0$, but in turns shares everything

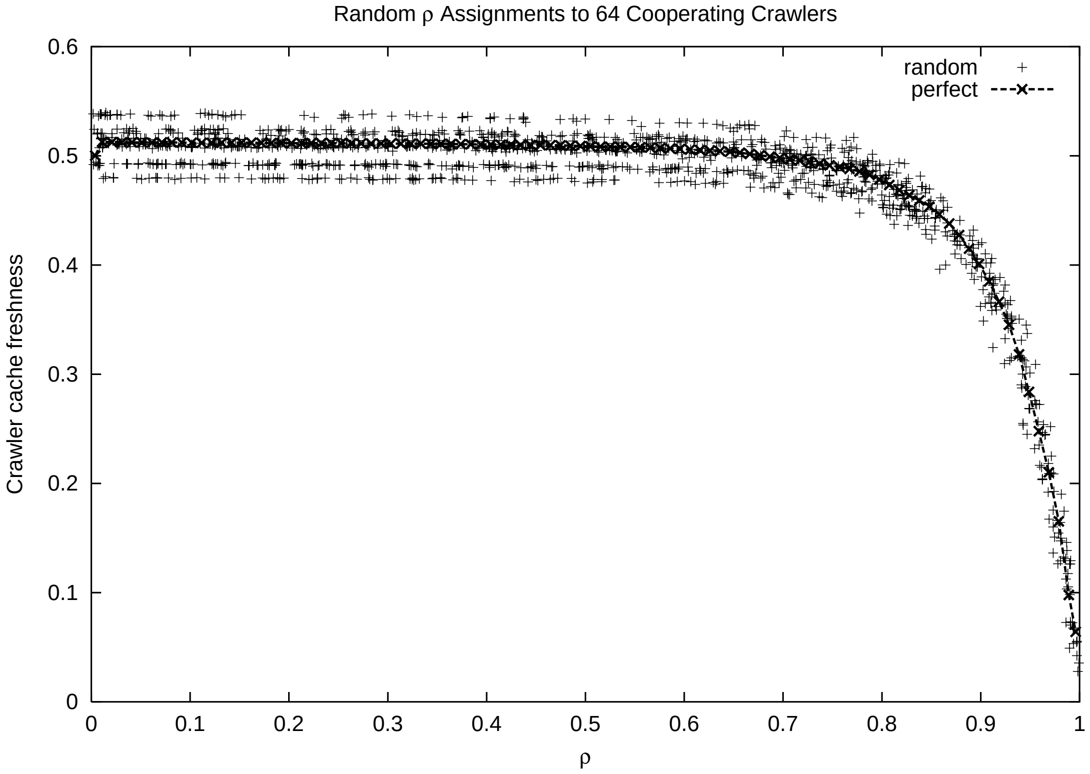


Figure 6.7: System of 64 web crawlers. Each has an independent, randomly assigned value of ρ. The equidistant distribution curve is overlaid as a comparison.

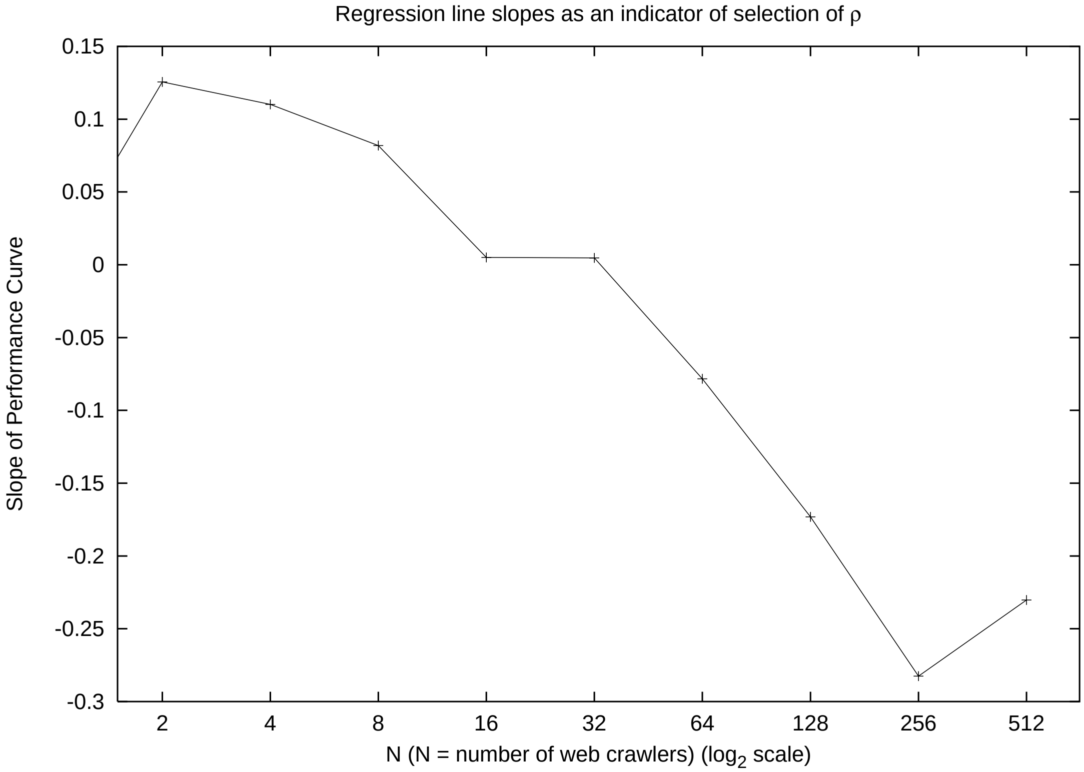


Figure 6.8: Slopes of regression lines for systems of crawlers in which a random ρ was assigned. The slope (m) is for a line $y = mx + b$ which fits to curves similar to figure 6.7.

that it merges with any peer. It relies on others to do network polling. A parasite is similar, but it does not share anything that it merges from others.

A system involving a total of N web crawlers was set up. Of those, $N - 1$ crawlers are regular cooperating web crawlers which use the same, fixed value of ρ. The last crawler acts as a global mirror which merges from all of the $N - 1$ crawlers and makes all of the shared data available to any other peer.

Figure 6.9 shows the results of comparing a mixed crawler environment for which $N = 32$ (ie. there are 31 cooperating crawlers and 1 mirror) and a completely cooperating environment (measured at $\rho = 0.85$). Figure 6.10 shows the same except with a single parasite instead of a mirror.

Both figures are extremely similar. Theoretically, the cooperating crawlers influenced by a parasite should be impacted more than those cooperating crawlers influenced by a mirror. However, with only one non-cooperative crawler in each experiment, this difference cannot be seen.

The other notable feature of each figure is that the crawler operating with a value of $\rho = 0$ *consistently* maintains a cache freshness as good as, or better than its cooperating peers. When all of the cooperating crawlers have a $\rho \simeq 1$, the non-cooperative crawler's cache freshness is at its best. This illustrates one of the properties of being a single non-cooperative crawler in a cooperative environment: a non-cooperative crawler's cache freshness can never be *worse* than the average cooperating peer.

Incentive to Crawl

Neither a mirror or a parasite poll. Hence, the above crawling systems have the equivalent polling power of a system of $N = 31$ crawlers (each with $\rho > 0$). As

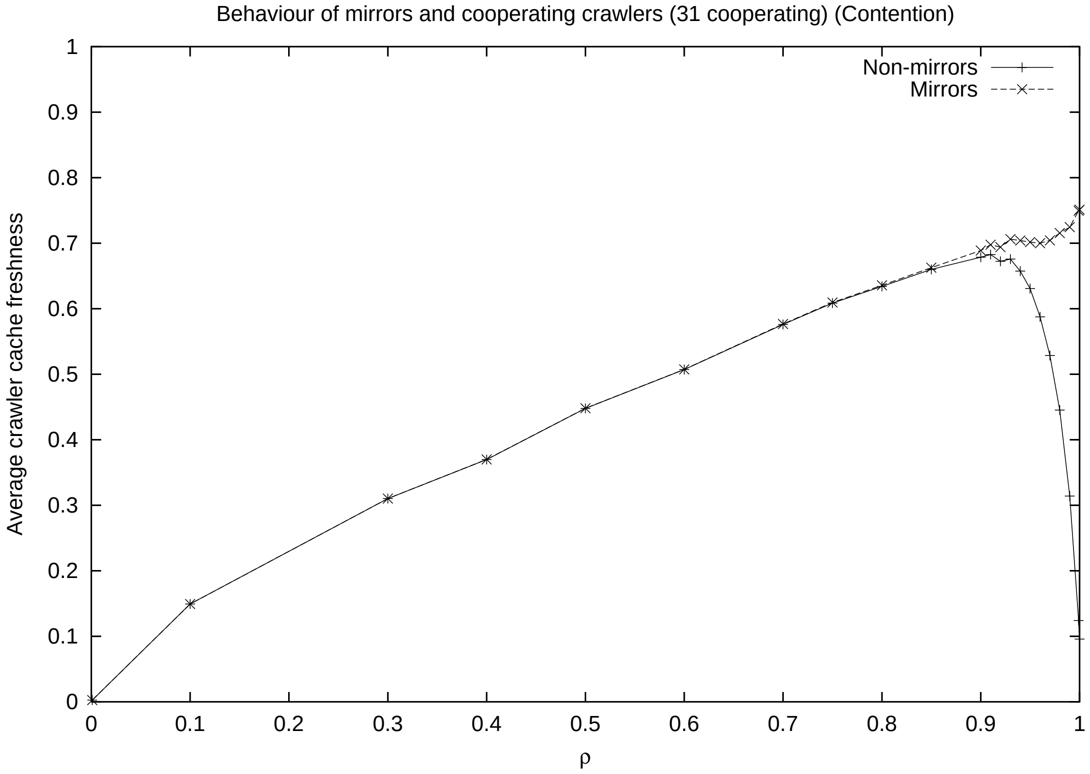


Figure 6.9: An example system of crawlers with a single global mirror. The mirror is consistently better than any of the cooperating crawlers for any ρ.

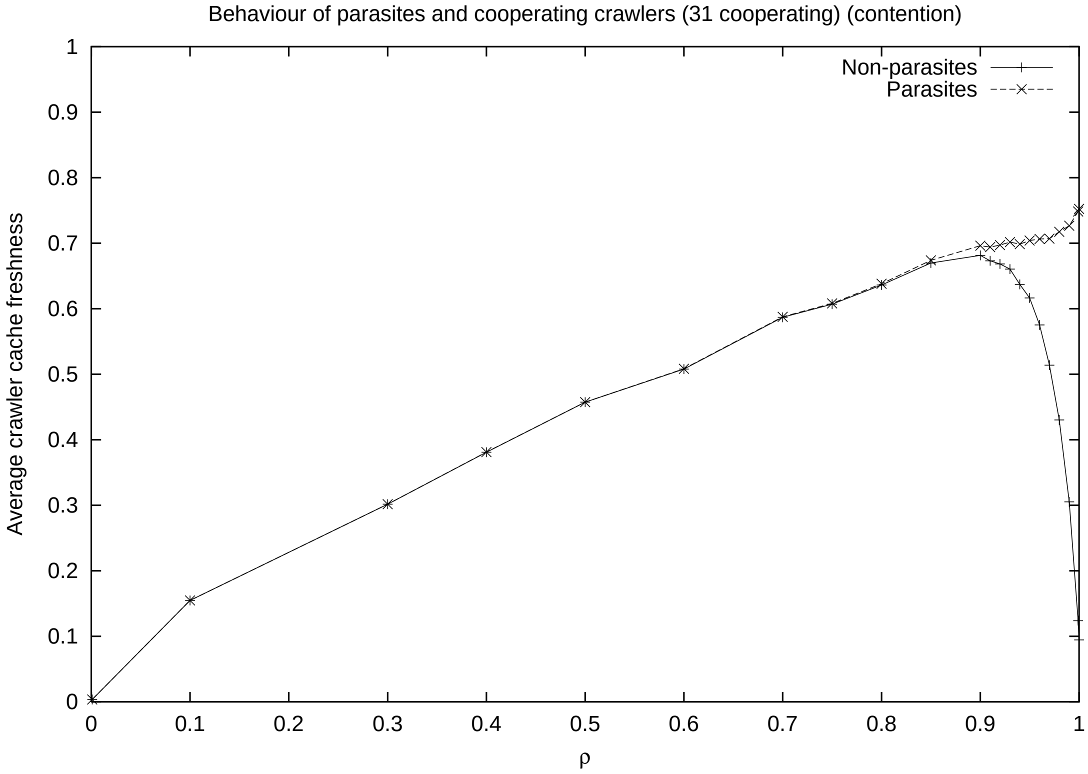


Figure 6.10: An example system of crawlers with a single parasite. The parasite is consistently better than any of the cooperating crawlers for any ρ.

shown in figure 6.6, increasing N (which increases the collective polling power) can increase freshness of the average crawler up to a point. Therefore, a non-cooperating crawler could achieve a greater degree of freshness if it contributed to the collective polling power.

Two simulations were set up such that $N - 1$ cooperating crawlers running with a fixed $\rho = 0.85$ were influenced by a single non-cooperating crawler (mirror and parasite, respectively). It is expected that the performance of a single non-cooperating crawler is inferior as compared to a pure cooperating crawler environment for small values of N. Similarly, it is expected that the performance of a single non-cooperating crawler is superior when N is large enough.

Figure 6.11 plots the results of the experiment in which a single mirror is used. The freshness of the mirror is compared to the freshness achieved by an average cooperating peer being influenced by the single mirror. Similarly, figure 6.12 plots the results when a single parasite is used. Accompanying each of the results is the curve shown in figure 6.6 to show how a completely cooperating environment of N crawlers would perform.

Again, both figures are extremely similar. The cooperating crawlers influenced by a parasite should be impacted more than those cooperating crawlers influenced by a mirror. With only one non-cooperative crawler in each experiment, this difference cannot be seen.

The expected behaviour is observed in both figures by the existence of a crossover point between the performance curve of a purely-cooperative system and a system containing a single non-cooperative crawler. For smaller N, the non-cooperating crawler does not contribute to the collective polling power which impacts negatively on the freshness of the entire system. For larger N, enough crawlers

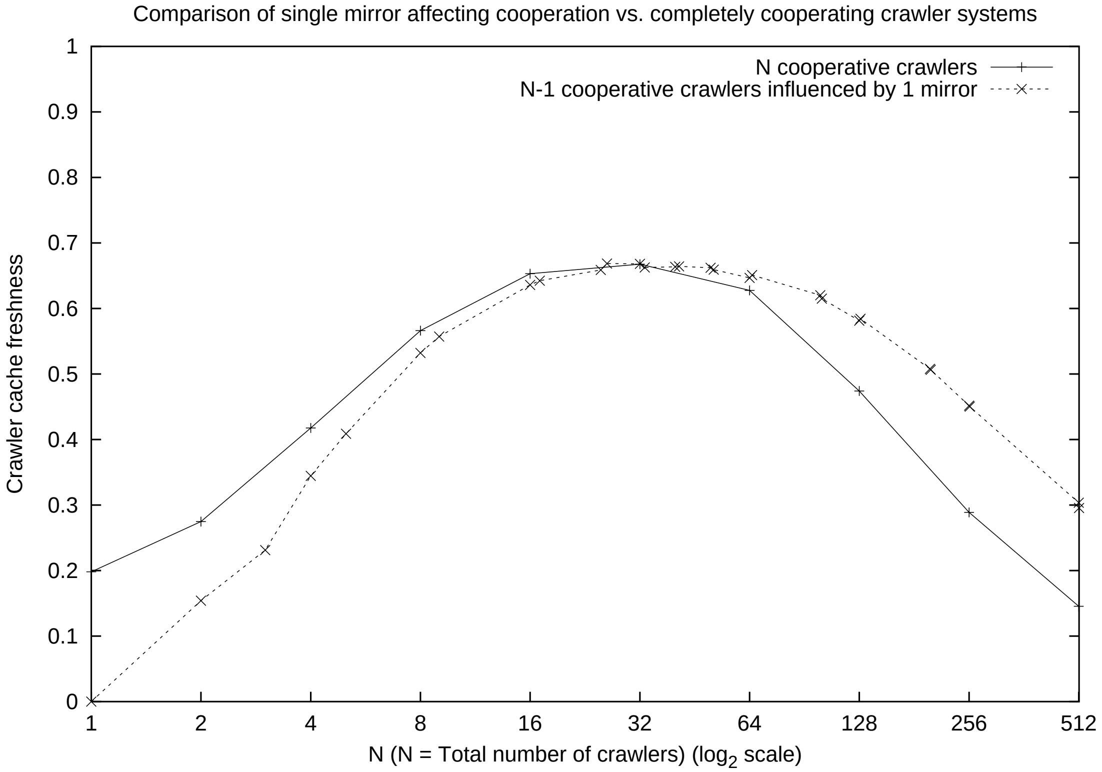


Figure 6.11: One global mirror operates within various-sized crawling systems.

exist to supply polling power. The cross-over point illustrates an over-abundance of cooperating crawlers.

6.3 Dynamic Strategies

Figure 6.7 hints that in the absence of any other knowledge about the crawling system, it may be useful for a single crawler to implement a value of $\rho = 0$ in order to receive the largest number of web-events. Unfortunately, if all crawlers decide on this strategy, then the freshness for an average crawler is worse-off than if all crawlers had all settled on a particular non-zero value for ρ.

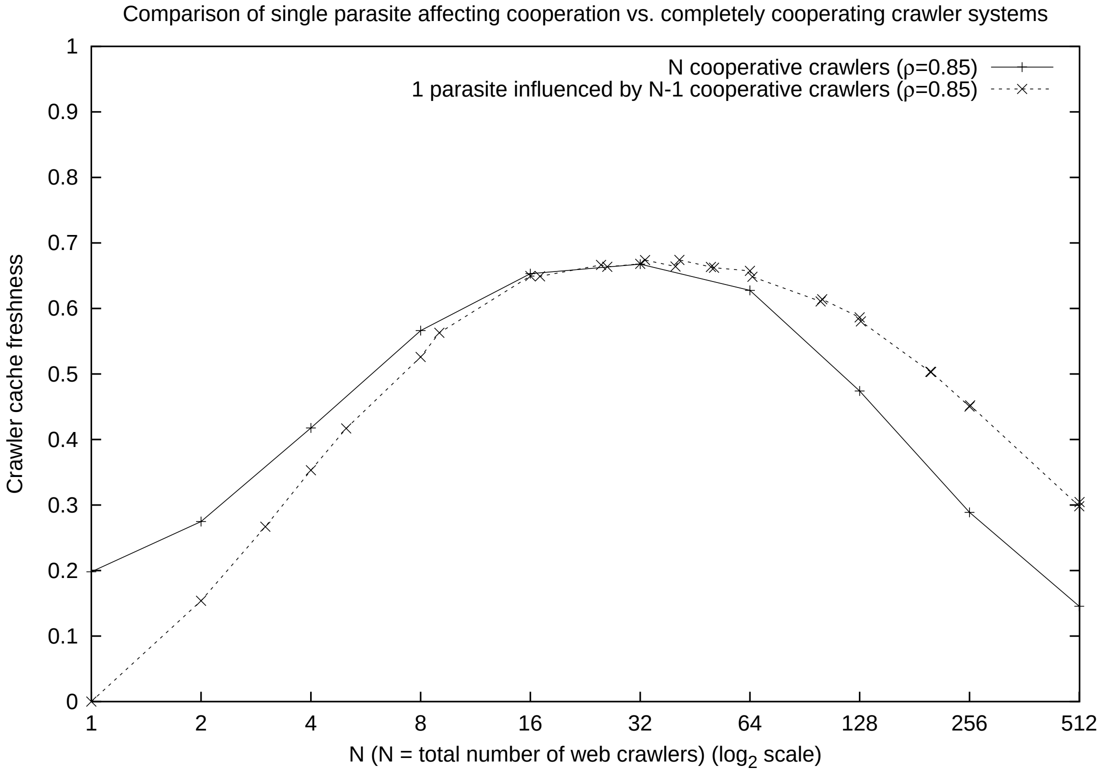


Figure 6.12: One parasite operates within various-sized crawling systems.

6.3.1 Bang-bang Dynamic Systems

A bang-bang system is a simple method used in control systems to adjust parameters dynamically in response to environmental conditions. We will allow crawlers to dynamically change their value of ρ depending on how effective their current value of ρ appears to be at maintaining their freshness. A crawler cannot measure their own cache freshness directly; they can only measure the freshness of independent web objects. To determine how effective it is doing, a crawler can use the following heuristics to approximately measure it's own performance:

- When a crawler polls, the number of times that it polls versus the number of web-events received for polling indicates how effective polling is. This ratio (call it p_r) can never be greater than 1 because a crawler can only receive 0 or 1 web-events per poll.
- When a crawler merges, the number of times that it merges versus the number of web-events received for merging indicates how effective merging is. This ratio (call it m_r) can be any number ≥ 0 since a merge can result in several web-events being transmitted per merge attempt.

The value of ρ should be high if $p_r > m_r$ and low if the converse condition occurred. The value of ρ must never be 1.0 or 0.0 since this would effectively ensure that m_r and p_r (respectively) could never change to challenge the other. Also note that if it takes a long time to perform a merge operation, then the next merge operation will result in a large number of web-events being ready. Transmitting all of these web-events takes time and resources. By the time they have been received by the merging crawler, more web-events could have been detected. This can result in a vicious circle in which m_r is high, but the overall freshness is still low.

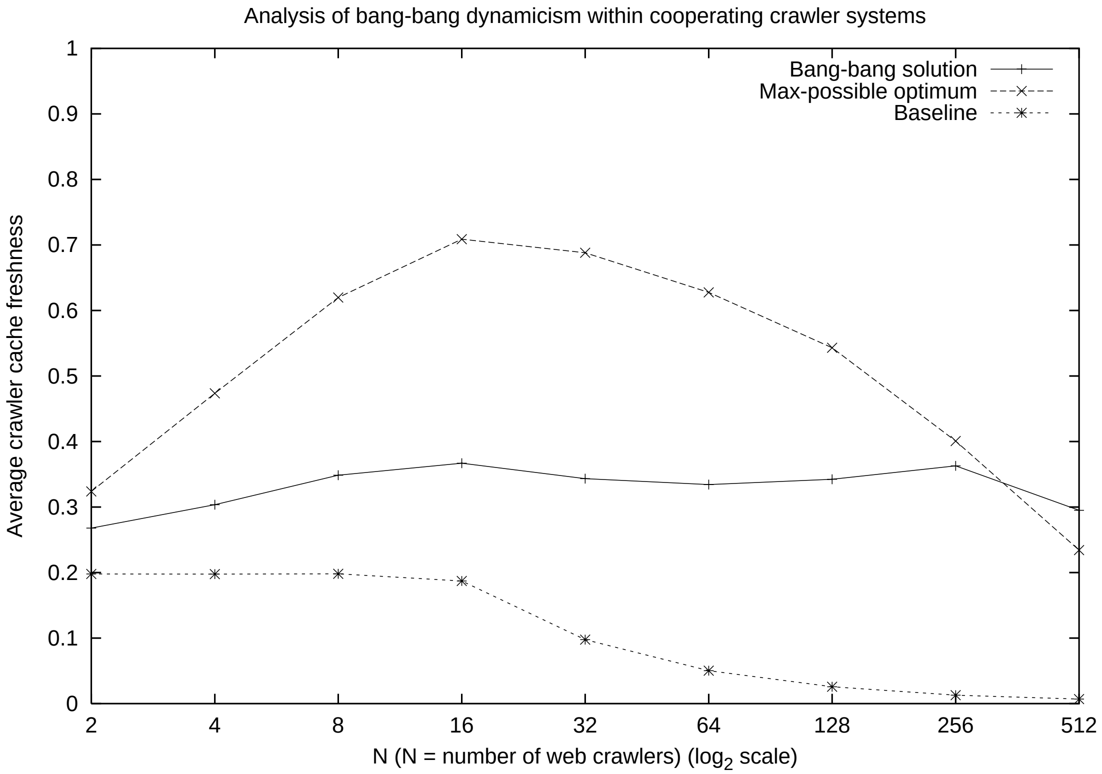


Figure 6.13: Simulations for various selected values of N, in which crawlers use the bang-bang model to adjust ρ. This model is compared to the baseline as well as the optimal freshness seen when using the fixed-ρ strategy.

In order to reduce the effects that past decisions made using this scheme have on the present, a dampener factor is applied to p_r and m_r. We only look at the previous 30% when making a decision about the present. No parameter studies were performed to determine how changing the dampener factor affected the freshness.

Figure 6.13 shows a series of simulations run for selected values of N. Each crawler in the system is using the bang-bang model for dynamically adjusting ρ. In this case, low is $\rho = 0.10$ and high is $\rho = 0.90$. A binary change from one to the other is performed: no other values of ρ is permissible.

Figure 6.13 shows the results of running a series of simulations for a specific set of N crawlers using the bang-bang dynamic strategy. In this figure, two other lines are present: the line labelled "max-possible optimum" represents the system of crawlers in which ρ has been set to a fixed constant for all crawlers in the N-crawler system (equivalent to the optimum observed in, for example, figure 6.3). The plain line is the baseline from figure 6.1.

It is interesting to note that this dynamic strategy produces a crawler cache freshness that is better than the baseline (figure 6.1) for all values of N. However, the difference between the freshness produced using this scheme is quite less than the maximum possible value achievable when ρ is fixed for small-to-mid values of N. Only for large values of N is the dynamic system better than the maximum possible freshness achievable when ρ is fixed. Note that this does not imply that a fixed-ρ strategy is the optimal strategy.

Equally interesting is that the bars are at an approximately constant height. This means that the bang-bang solution was successful in modifying ρ to account for the size of N.

Table 6.1 shows the percentage of time that an average crawler in a system of N crawlers spent in the low-ρ position ($\rho = 0.10$), and how much time it spent in the high-ρ position ($\rho = 0.90$).

It is clear in table 6.1 that the system is adjusting the amount of time spent in the high and low modes as a function of the number of crawlers in the system. These results support the previous findings that when N is small, a high ρ should be used, and when ρ is medium-large, a lower ρ should be employed. The interesting result is when N is very large: we see that ρ-high has started to increase for $N \geq 256$. It is expected that when $N \rightarrow \infty$ it is no longer an issue to determine which value

N	ρ-low	ρ-high
2	0.362	0.638
4	0.572	0.428
8	0.707	0.293
16	0.780	0.220
32	0.805	0.195
64	0.808	0.192
128	0.816	0.184
256	0.782	0.219
512	0.628	0.373

Table 6.1: Percentage of time an average crawler in the bang-bang dynamic strategy spends in ρ-low mode ($\rho = 0.10$) and ρ-high mode ($\rho = 0.90$).

of ρ is more beneficial than another. Because of the sheer number of crawlers, any value of ρ will result in the same freshness performance – namely that freshness will approach zero. It is expected that if we continue to increase N, table 6.1 would show ρ-low=ρ-high=0.5.

Chapter 7

Conclusions and Future Work

7.1 Conclusions about Cooperative Behaviour

With the size of the web increasing at a dizzying rate, web crawlers are being more and more challenged to discover and maintain web objects, especially on behalf of web search engines. Currently, web search engines relying on web crawlers to keep their indices up-to-date are falling behind. Certain techniques can ensure that the most popular material is kept up-to-date, but that is limiting.

This thesis presented a general protocol to allow competing web crawlers to share information among peers to their mutual benefit. Crawlers can choose to merge shared data from competing peers if it helps them to maintain a fresher database at a lesser cost than directly polling web objects on the web. However, merging can become as prohibitive if not carefully controlled. Hence, it is shown in theory and simulation that web crawlers must strike a balance between polling and merging to obtain a degree of freshness which can exceed current-day behaviour.

A number of simulations were devised to show the behaviour of combining polling and merging within a system of crawlers. All crawlers were based on a simple algorithm in which the key lay in the ability to switch between polling and merging via a parameter ρ.

Simulations in which the value of ρ was fixed at a particular value for all crawlers within a system show that an optimum can be reached which always results in an average crawler cache freshness better than the baseline. The value of ρ that can be used to obtain such performance changes as a function of N. When N is small, ρ should favour more polling than merging; when N increases ρ can decrease toward 0. When N is too large – that is, when there is an overabundance of crawlers in the system, any value of ρ seems to work just as well as any other.

Changing the behaviour of single cooperating web crawlers to function in non-cooperative, merge-only modes serve to illustrate the property of incentive-to-crawl. A non-cooperating web crawler could conceivably perform better if it was acting in a mutually cooperative manner for small values of N. Larger values of N showed that a non-cooperative crawler would gain no benefit from actively polling the web.

Finally, a simulation was built using crawlers that could dynamically change their individual values of ρ. The results indicate that a very simple bang-bang technique can produce better freshness results for a crawler using the strategy as compared to the baseline model, though other non-dynamic strategies are superior. However, the ability to change ρ means that no pre-defined value need exist, which can reduce the complexity in attempting to search ρ based on the size of a community of crawlers. Furthermore, if the number of crawlers can change (which is a realistic scenario), a specific value of ρ at some time t_i may be unsuitable for a community at time t_j.

7.2 Application to the Real World

7.2.1 Implementation Issues

The protocol outlined by Ho[24] uses a communication interface module that can be plugged into a web crawler. Two flat files (`webrobots.shr` and `webrobots.dat`) provide all of the information necessary to share information among crawlers (the former is the list of web crawlers known to the local crawler, and the latter is the data to be shared). A crawler supporting the protocol generates and maintains these files. Visiting robots use HTTP natively to retrieve the contents of the files. Unfortunately, a number of issues go unaddressed.

Computationally, flat files are expensive to manage when existing data needs to be updated rather than appended. Ho recognizes this, and indicates that although the `webrobots.shr` file will remain fairly small (since there are only a small number of robots on the web), the `webrobots.dat` file will grow to an impractical size. Two alternatives to using flat files are to use an indexed file scheme, or a more sophisticated database engine.

Storing information to be shared in a database is the most flexible solution of all, but is potentially the most computationally expensive solution. Downloading needless data can be reduced to nil through the use of query languages and filtering techniques.

Server Locating

Locating a web crawler that implements the cooperation protocol presents a circular problem. How does one locate a web crawler on a distributed network that, in itself, can not function without published hyperlinks contained within HTML web pages?

Essentially, the problem is reduced to an accepted means to publishing references to web crawlers in some well-known form (URLs?) in well-known locations (specific web servers? specific communication ports? specific web pages?).

One example of detecting web crawlers requires assistance by web servers. Web crawlers can be detected by their activities on web servers. The action of downloading `robots.txt` usually indicates the presence of a web crawler. The IP and name could be saved by the web server and stored in a file called `robots.dat`, which could be scanned and merged by any crawling robot to build up a database of potential cooperating crawlers.

7.2.2 Security Concerns

The act of updating the `/robots.shr` file or any shared web-event data file causes a web-event to be generated for that specific file. This can easily generate an infinite loop if two (or more) web crawlers are polling each other's shared web-event data files. To avoid such aberrant behaviour, it is desirable to either limit the number of times each URL can generate an event in a given period or simply avoid recording events pertaining to shared data files. Protection of the share-repository via `/robots.txt` is the prudent thing to do.

The other concern is that up to this point, all data has been assumed to be correct. Unfortunately, malicious web clients adhering to the protocol may practice *cache poisoning*. Cache poisoning can occur when malformed, misleading, or incorrect web-event data is shared by crawler S and added to the collection of data possessed by crawler M. Crawler M could, in turn, distribute this poisoned data to other peers. Decisions to merge or crawl based on previously merged data (which could be tainted) should be avoided. Trust-networks[1] could be used to provide web crawler community feedback regarding the quality of data shared by specific peers.

7.3 Future Work

Three major avenues for future expansion and exploration can be identified.

7.3.1 More dynamic systems

One of the more interesting, but unanswered questions raised by this work deals with how web crawlers can dynamically alter their strategies to increase their own personal cache freshness. As alluded to in earlier chapters, autonomous behaviour does not necessarily imply non-cooperative behaviour. Indeed, this is shown in some of the results of experiments in the previous chapter. Ho[24] showed that self-interested web crawlers can mutually benefit from web-event data sharing; implementing Ho's biological fitness model is an avenue left to be explored.

Additional dynamic strategies could be examined with respect to how they can exploit the relationships and trends discovered by the analysis presented in Chapter 6. Some interesting examples include:

- A crawler could modify its own ρ depending on the number of crawlers perceived in the system.
- A share/merge ratio system could be employed to enforce cooperation. Crawlers violating the ratio system would be forced to poll (ie. they would have to adjust their ρ to favour polling), rather than be allowed to merge.

- Similar to the bang-bang model, crawlers could adjust their ρ depending on the number of *important* web-events detected. Although importance is a qualitative judgment, one could assume that CREATE web-events are slightly more interesting and hence more important than UPDATE events. DELETE events would not be very interesting since no further events could come from a deleted web object.

- Crawlers could advertise their value of ρ to peers. A crawler would use the set of advertisements to adjust it's own value of ρ in an attempt to optimize. Very little work was done on the behaviour of crawling systems with varied values of ρ. Figure 6.7 represents the behaviour when ρ is randomly-distributed among all crawlers in the system.

7.3.2 Real-world Study

It would be useful to actually run a real-world study. A series of web crawlers implementing the web-event data sharing algorithm 4.1 could validate the trends seen in the simulations.

7.3.3 Ubiquitous Sources of Web-event Data

It would be interesting to use sources other than web crawlers to collect web-events. Any web client could be a candidate: this includes web crawlers (by design), web caches, proxies, and even web users.

A subset of web-users providing web-event data could provide magnitudes more timely event-detection, since there are many more web users than web crawlers. It should be noted that collection of data from users poses potential security hazards and ethical treatments which are beyond the scope of this work.

Providing web-event dissemination services with other sources (web servers, proxy servers, etc.) could be realized through development of an Apache[1] web server module. This could be used to provide the services described by Brandman *et al* [5]and Gupta *et al* [20].

[1] http://www.apache.org/

Bibliography

[1] Alfarez Abdul-Rahman and Stephen Hailes. A Distributed Trust Model. In *Proceedings of the 1997 New Security Paradigms Workshop*, pages 48–60, September 1997.

[2] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC 1945: Hypertext Transfer Protocol - HTTP/1.0, May 1996. Available at `http://www.faqs.org/rfcs/rfc1945.html`.

[3] Krishna Bharat and Andrei Broder. A Technique for Measuring the Relative Size and Overlap of Public Web Search Engines. In *Proceedings of the 7th International World Wide Web Conference*, pages 379–388, Brisbane, Australia, April 1998.

[4] C. Mic Bowman, Peter Danzig, Darren Hardy, Udi Manber, Michael Schwartz, and Duane Wessels. Harvest: A Scalable, Customizable Discovery and Access System. Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado, Boulder, March 1995.

[5] O. Brandman, J. Cho, H. Garcia-Molina, and N. Shivakumar. Crawler-Friendly Web Servers. In *Workshop on Performance and Architecture of Web Servers (PAWS)*, June 2000.

[6] Brian E. Brewington and George Cybenko. How dynamic is the Web? *WWW9/Computer Networks*, 33(1-6):257–276, 2000.

[7] Sergey Brin and Lawrence Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.

[8] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design of a Scalable Event Notification Service: Interface and Architecture. Technical Report CU-CS-863-98, University of Colorado at Boulder, Colorado, USA, September 1998. Available at `http://www.cs.colorado.edu/∼carzanig/papers/CU-CS-863-98.ps.gz`.

[9] James R. Chen, Nathalie Mathe, and Shawn Wolfe. Collaborative Information Agents on the World Wide Web. In *ACM DL*, pages 279–280, 1998.

[10] Junghoo Cho and Hector Garcia-Molina. Estimating Frequency of Change. Technical Report ID-135, Standford University, Stanford, CA USA, November 2000. Available at `http://www-db.stanford.edu/pub/papers/cho-freq.ps`.

[11] Junghoo Cho and Hector Garcia-Molina. Synchronizing a Database to Improve Freshness. In *Proceedings of 2000 ACM SIGMOD International Conference on Management of Data*, pages 117–128, Dallas, Texas, USA, May 2000.

[12] Junghoo Cho and Hector Garcia-Molina. The Evolution of the Web and Implications for an Incremental Crawler. In *Proceedings of 26th International Conference on Very Large Databases (VLDB)*, pages 200–209, September 2000.

[13] Junghoo Cho, Hector García-Molina, and Lawrence Page. Efficient Crawling

Through URL Ordering. *Computer Networks and ISDN Systems*, 30(1-7):161–172, 1998.

[14] Chaisen Chung. Topic-Oriented Collaborative Web Crawling. Master's thesis, University of Waterloo, Waterloo, Ontatio, Canada, 2002.

[15] J. Cohen and S. Aggarval. General Event Notification Architecture Base, July 1998. Available at `http://www.alternic.org/drafts/drafts-c-d/draft-cohen-gena-p-base-01.txt`.

[16] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey C. Mogul. Rate of Change and Other Metrics: A Live Study of the World Wide Web. In *USENIX Symposium on Internet Technologies and Systems*, December 1997.

[17] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol - HTTP/1.1, June 1999. Available at `http://www.faqs.org/rfcs/rfc2616.html`.

[18] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. RFC 2518: HTTP Extensions for Distributed Authoring - WEBDAV, February 1999. Available at `http://www.faqs.org/rfcs/rfc2518.html`.

[19] Amy R. Greenwald and Jeffrey O. Kephart. Shopbots and Pricebots. In *Agent Mediated Electronic Commerce (IJCAI Workshop)*, pages 1–23, 1999.

[20] Vijay Gupta and Roy H. Campbell. Internet Search Engine Freshness by Web Server Help. In *Symposium on Applications and the Internet*, pages 113–119, 2001.

[21] Manfred Hauswirth and Mehdi Jazayeri. A Component and Communication Model for Push Systems. In *Proceedings of the Seventh European Engineering Conference held jointly with the Seventh ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 20–38, Toulouse, France, September 1999.

[22] Allan Heydon and Marc Najork. Mercator: A Scalable, Extensible Web Crawler. *World Wide Web*, 2(4):219–229, 1999.

[23] Annika Hinze and Daniel Faensen. A Unified Model of Internet Scale Alerting Services. In *Proceedings of the International Computer Science Conference (ICSC)*, pages 284–293, 1999.

[24] Kinson Ho. WatE$\wedge$er: An Effective and Efficient Web Notification Protocol. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, 1999.

[25] Brian Kantor and Phil Lapsley. RFC 977: Network News Transfer Protocol, February 1986. Available at `http://www.faqs.org/rfcs/rfc977.html`.

[26] M. Koster. Aliweb - Archie-Like Indexing in the Web. In *Proceedings of the First International World Wide Web Conference*, pages 175–182, Amsterdam, March 1994.

[27] Martijn Koster. Robots Exclusion Standard. Available at `http://www.robotstxt.org/`.

[28] Steve Lawrence and C. Lee Giles. Accessibilty of Information on the Web. *Nature*, 400:107–109, July 1999.

[29] Michael L. Mauldin. Lycos: Design choices in an Internet search service. *IEEE Expert*, (January-February):8–11, 1997.

[30] Robert Miller and Krishna Bharat. SPHINX: A Framework for Creating Personal, Site-Specific Web Crawlers. In *Proceedings of the Seventh International World Wide Web Conference*, Brisbane, Australia, April 1998.

[31] A. Moukas. Amalthaea: Information Discovery and Filtering using a Multiagent Evolving Ecosystem. London, 1996.

[32] Brian H. Murray and Alvin Moore. Sizing the Internet. White paper, Cyveillance, July 2000. Available at http://www.cyveillance.com/.

[33] R. Nielsen, P. Leach, and S. Lawrence. RFC 2774: An HTTP Extension Framework, February 2000. Available at `http://www.faqs.org/rfcs/rfc2774.html`.

[34] Surendra Reddy and Mark Fisher. Event Notification Protocol - ENP. WEBDAV Working Group Internet Draft, June 1998. Available at `http://alternic.net/drafts/drafts-r-s/draft-reddy-enp-protocol-00.html`.

[35] David S. Rosenblum and Alexander L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *Proceedings of the Sixth European Software Engineering Conference/ACM SIGSOFT Fifth Symposium on the Foundations of Software Engineering*, pages 344–360, Zurich, Switzerland, September 1997.

[36] J. Slein, F. Vitali, E. Whitehead, and D. Durand. RFC 2291: Requirements for a Distributed Authoring and Versioning Protocol for the World Wide Web, February 1998. Available at `http://www.faqs.org/rfcs/rfc2291.html`.

[37] Aarno Lehtola Tuula Käpylä, Isto Niemi. Towards an Accessible Web by Applying PUSH Technology. In C. Stephanidis and A. Waern, editors, *4th*

ERCIM Workshop on "User Interfaces for All", Stockholm, Sweden, October 1998.

[38] J.L. Wolf, M.S. Squillante, P.S. Yu, J. Sethuraman, and L. Ozsen. Optimal Crawling Strategies for Web Search Engines. In *World-Wide Web 2002*, Honolulu, Hawaii, USA, May 2002.

[39] Hayato Yamana, Kent Tamura, Hiroyuki Kawano, Satoshi Kamei, Masanori Harada, Hideki Nishimura, Isao Asai, Hiroyuki Kusumoto, Yoichi Shinoda, and Yoichi Muraoka. Experiments of Collecting WWW Information Using Distributed WWW Robots. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 379 – 380, Melbourne, Australia, August 1998.

[40] Haobo Yu, Deborah Estrin, and Ramesh Govindan. A Hierarchical Proxy Architecture for Internet-scale Event Services. In *IEEE Eighth International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, Palo Alto, CA, June 1999. IEEE.