Title: A Scalable Web-Based Real-Time Information Distribution Service for Industrial Applications

Session: Special Session on Network Based Technologies and Applications

Authors' Names: Seejo Sebastine, Kyoung-Don Kang, Tarek F. Abdelzaher and Sang H. Son

Contact Author: Seejo Sebastine

Address: Department of Computer Science, University of Virginia, Charlottesville, VA 22903

Phone: (804) 982 - 2610, 2294, 2200, 2205

Fax: (804) 982 - 2214

E-mail: {seejo, kk7v, zaher, son}@cs.virginia.edu

Keywords: Content Distribution, Dynamic content, Real-time Monitoring, Scalablity

A Scalable Web-Based Real-Time Information Distribution Service for Industrial Applications *

Seejo Sebastine, Kyoung-Don Kang, Tarek F. Abdelzaher, Sang H. Son
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
{*seejo, kk7v, zaher, son*}@*cs.virginia.edu*

Abstract

In this paper we discuss a web-based content distribution service for wide-area distributed embedded systems to allow remote monitoring of their state using web browsers, servers, and caches. The service allows browsing the real-time state of remote devices, industrial processes, controllers, appliances, agents, and other entities that may publish their state. For a remote monitoring application, it creates the illusion that all monitored entities are available locally in real-time, as opposed to scattered across a wide-area network with unpredictable backbone delays. The main difficulty is to achieve this in a scalable manner without wide-area network support for real-time communication while simultaneously reducing the amount of backbone traffic (which carries process state updates). Another challenge is to acheive this without significant modifications to the existing web infrastructure. To support decision making, we provide timely response and bounded consistency for the monitored data. We have built a prototypical service architecture based on freely available web servers/proxy caches and shown its basic applicability to industrial applications by the experimental results.

*The work reported in this paper was supported in part by the National Science Foundation under grants CCR-0093144 and CCR-0098269.

1 Introduction

We envision a global information utility for distributed industrial applications which require concurrent soft real-time monitoring of devices, controllers, agents, or other entities situated at geographically diverse locations. Examples of such applications may include distributed intelligent vehicle transportation systems, automated metropolitan traffic information systems, process monitoring in industrial factories, distributed financial market transactions, and distributed interactive multi-agent systems.

The utility offers the abstractions of *publishers* and *subscribers*. Publishers are devices or agents whose state must be made globally available. For example, a traffic sensor on a smart freeway may constitute a publisher. In this case, the sensor might make available, measurements of the degree of congestion on the freeway to be accessed by wireless agents located in vehicles, which in turn advise their drivers of the most opportune route to the destination. The subscribers are those who access the information made available by the publishers; in this case, the agents in the vehicles. As is evident in this example, the publishers do not always know who the subscribers are, since the pool of subscribers may vary dynamically.

Due to its pervasiveness, the Internet is the most logical choice of network infrastructure for the envisioned wide-area information dissemination utility. The World Wide Web is the dominant interface to Internet-based services today. Hence, in our architecture, publishers and subscribers communicate using a web-based interface. One method that can be used to publish such information is by hosting it in web pages accessed by the subscribers. Since the published information can change frequently, these web pages will need to be *dynamically generated* as requests for this information arrive from subscribers.

Essentially, such a global information dissemination utility is provided by a *content distribution service* which handles the task of disseminating the published information to all its subscribers. In order for such a content distribution service to be of use in industrial applications it must handle two requirements, timeliness and consistency; and must at the same time be scalable. Consider the distributed intelligent transportation system in the previous example. The information publishers can sense various freeway conditions such as traffic volumes, vehicular speeds, etc. to determine congestion on the freeway. This information can be published on the web using *publisher agents* in a web server to dynamically generate web pages to reflect the time-varying real-world state. Subscribers to the service contact the web server to determine the current traffic condition on the freeway. In a large scale system, lots of subscribers may concurrently contact the web server for up-to-date information. As a result, the web server can be considerably overloaded. The service delay can increase significantly, potentially making the information useless from the subscribers' perspective. For example, consider a subscriber requesting the current traffic status and receiving a delayed response. It would do little help to the subscriber to infer that the vehicle should have exited the freeway, after the exit has already passed! Hence timely response, in spite of the unpredictability of the Internet and possible server overload, is of utmost importance.

Scalability is also an important consideration in such a service. For example, in a distributed freeway control system handling a multitude of freeways, there could be thousands of vehicles on the freeways at the same time, all accessing the traffic information concurrently. In such a scenario, it is imperative that relevant published information be replicated to avoid system overload.

The solution we suggest to satisfy all these requirements is *guarantee-based, QoS-sensitive, on-demand replication and caching* of published information, at proxy caches close to the subscribers. Caching published content at proxy caches near the subscribers can improve response time. Furthermore, the scalability of the service can be improved by reducing the publisher (original content server) load and the Internet backbone traffic. However, caching can introduce consistency problems. The original publisher would update the original content to reflect the real-world status, which can change from time to time such as the traffic condition and stock prices. To address this problem, we provide a *bounded consistency* for dynamic content between the original publisher and the proxy caches. One important hindrance in implementing this service on the current web infrastructure is the *lack* of support for caching dynamically changing data on existing proxy caches.

We propose a novel scalable architecture for on-demand replication of *dynamically changing* web content. Our architecture allows the publisher of the information to specify the guarantees required regarding the consistency of the content. For example, the publisher of stock quotes can require that the information seen by the subscriber should not be more than 3 minutes old. In comparison, the publisher of traffic information can require that it be consistent within a period of 10 minutes. Our architecture aims to provide the best possible QoS to the subscriber, while simultaneously trying to meet the publisher specified guarantees. In this regard, we employ a novel QoS-sensitive approach, which dynamically trades off consistency to improve the latency, considering various factors such as backbone network congestion, potential overload at the caching proxy, etc. This architecture is the main contribution of our information service. This paper describes a prototype of this architecture which has the merit of being compatible with today's web infrastructure and protocols, allowing it to be easily deployed on the current Internet.

The rest of this paper is organized as follows. Section 2 describes the related work in this area. Section 3 describes the *"active caching"* architecture and its various components. Section 4 presents the QoS model used by us to provide guarantees. Section 5 gives the experimental evaluation of our architecture. Section 6 concludes the paper and discusses avenues for future work.

2 Related Work

The issue of scalable on-demand *static* content replication and distribution has been addressed in numerous papers [1, 2, 3, 4, 5]. In comparison, related research is relatively scarce when dealing with *dynamic* web content. However, recently there has been considerable interest in attempting to cache dynamic content to improve the scalability and availability of web-based information services [6, 7, 8, 9, 10, 11] The methods proposed attempt to reduce the workload of dynamic content generating web servers by caching the generated content on a distributed set of reverse-proxies situated just before the web servers [7, 8, 11, 12, 13]. Specifically, Smith et al. [7] proposed a solution for cooperative caching of dynamic web content on distributed *web servers*. Clustered nodes collaborate with each other to cache and maintain result consistency using a Time-To-Live protocol. Iyengar et al. [8, 11] proposed a scheme in which servers export an API that allows individual applications to explicitly cache and invalidate application-specific content. They achieved significant performance improvement in web servers having a high proportion of dynamically generated web content. However, these approaches are limited to *web servers*. As a result, the user perceived latency improvement and backbone traffic reduction can be relatively limited as compared to proxy caching, which caches dynamic content near the clients.

Cao et al. [6] proposed an *Active Cache* which caches dynamic content at a proxy. Their solution is to attach a piece of Java code, called a *cache applet*, with each dynamic document. The cache applet is executed on the proxy whenever a request for a cached document is received. It provides the requested service without contacting the original content server. This approach is very flexible and can be used to maintain consistency in an application-specific manner while at the same time allowing dynamic updates to the existing documents. However, this flexibility is achieved with a high computational cost. It requires starting up a new Java process (VM or compiled code) for every request in order to execute the applet. Also, it is required to create and maintain a pattern-matching network for each cache-applet to keep track of result equivalence.

As regards the issue of consistency management, a lot of work has been done in the case of static web content, as is evident from the widely accepted Time-To-Live (TTL) approach. In the recent past, a considerable amount of work has also been done in managing consistency of dynamically changing information on *web servers* [7, 8, 11, 12, 13]. On a web server, strong consistency is feasible since the updates and the data dependency are available locally, which is not the case in a proxy server. Our approach addresses this issue by trading off consistency in a bounded manner for improved response time (due to proxy caching).

Recently, research interest has been rapidly increasing regarding the applicability of the Internet for remote monitoring and control of manufacturing operations. [14] shows the wide applicability of the Internet for remote monitoring and control. Service response time is considered to be one of the main problems for the acceptance of Internet-based monitoring/control. We handle this problem by caching dynamically generated content such as the process state in industrial plants near the subscribers, e.g., operators, to reduce the response time. In [15], the web is used to enable the monitoring of industrial plants in the LAN or intranet environment. In contrast, our service architecture is designed to deploy in the Internet to enable remote monitoring/control on a global scale.

3 An Active Web Caching Architecture

In this section we present our novel scalable architecture for guarantee- based on-demand replication of *dynamically changing* web content such as traffic data, stock quotes, industrial process state etc. Our architecture is designed to build upon the existing web infrastructure and does not need any components that are not already in use on the web. *Publisher agents (or scripts)* are programs which interface the *publisher* of the information to a web server. A publisher agent queries the state of the publisher (possibly using a proprietary protocol) and converts the obtained state representation into a dynamically generated web page (in HTML format), which can then be served by the web server to all requesting subscribers. The subscribers can then display this information in their browsers, or use it in arbitrary computations. A single web server may interface to multiple publishers, each through its own agent.

In our approach, the *publisher agents* are replicated on-demand onto *"active caches"* as shown in Fig. 1. Current web proxy caches can only replicate static web content while volatile, dynamic content is deemed uncacheable. We augment the current static caching infrastructure with our *active caches*. Active caches replicate dynamically generated content on-demand, hence, augmenting the functionality of the web with the capabilities required to implement our global information utility. We designed our active caches to be compatible with current caching proxies. Only minor modifications are needed in existing proxy configuration files to interface to our novel service.

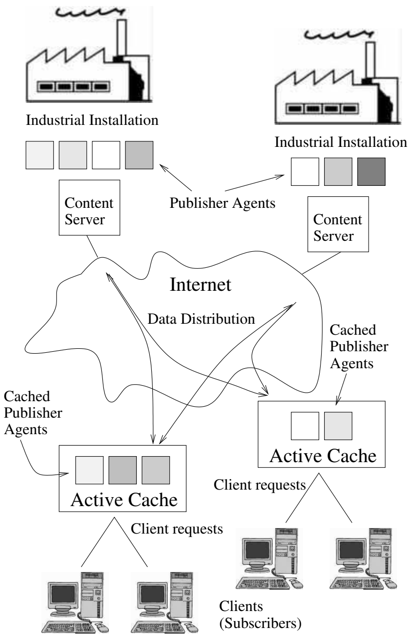

Figure 1. Architectural View

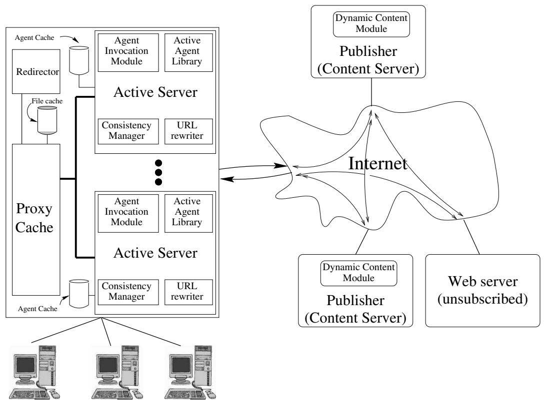

Figure 2. Components of an Active Cache

The active caches take care of the responsibility of replicating agents and published data on the cache, and maintaining data consistency with the publishers. Each active cache can communicate with the other caches, with the client and with the rest of the Internet. Each *active cache* is composed of two primary components; a standard static web *proxy cache* and an *active server*, as shown in Fig. 2. The *proxy cache* handles the job of caching static content and is already a part of the existing web infrastructure, while the *active server* is a new component that handles the job of executing one or more types of related dynamic content, e.g. cgi, fast-cgi, ASP, etc. Hence, each *active cache* can have multiple *active servers* to handle different types of dynamic content.

We have used a modified version of the freely available Squid proxy cache [16] as our *proxy cache* component and the Apache web server [17] as our *active server*. Both Squid and Apache are largely used in the web infrastructure today. Subscribers contact the Squid cache component with content requests. If the requests are for static content, Squid directly contacts the *content server*, retrieves the results, and caches them for future accesses. However, if the requests are for dynamic content, Squid redirects them to the Apache *active server* component, which then invokes the corresponding publisher agent to generate the requested content.[1]. Since both the *proxy* and *active server* components are either running on the same physical machine or on machines on a local network, the overhead involved is far less than going over the Internet backbone to the *content server*. In addition, Squid also temporarily caches the results published by the publishing agent. Hence requests in the immediate future for the same published values are served from the Squid cache (if they are still valid), thereby reducing the subscriber-perceived latency.

The other components of the architecture as shown in Fig. 2 are a *URL rewriting engine*, the *Consistency Manager module* and an *agent cache*. The URL rewriting engine handles the job of fetching publisher agents from the content server, if they are not locally available. The agent cache caches these agents until they are no longer valid. The Consistency manager implements our QoS-sensitive consistency management algorithm as described in Section 4.

The *content server* also requires a few modifications in order to recognize requests from active caches. Upon receiving a request for a publisher agent from an active cache, its *dynamic content module* forwards the agent and its data to the active cache, instead of invoking it locally. When the requested publisher agent is received, the active cache invokes it locally (in the *Agent Invocation Module*) and also caches it in its *agent cache*. In order for the publisher agents to interact with our system we require them to adhere to our *"Active Agent API"*, implemented by the *"Active Agent Library"*. A limitation of our system is that currently it deals only with "read-only" agents and does not allow the agents to modify any server state.

4 QoS-Sensitive Aproach to Dynamic Content Caching

In this section, we formulate a guarantee-based, QoS-sensitive consistency management scheme introduced in Section 1. Ideally, subscribers would prefer to have the most up-to-date information published by the publisher. However, this would entail forwarding every request to the original content server over the Internet. This can incur an unpredictable response time, caused mainly by two bottleneck

[1]The agent may be fetched from the *content server*, if it is not available

factors: unpredictable Internet backbone delays and possible server overload. To ensure that subscribers get a timely (and fairly recent) response, thereby aiding informed decision making, we trade off the consistency of the information for timeliness. Furthermore, we require the publisher to specify the desired consistency bounds to ensure that the subscribers do not receive completely out-of-date information. We formulate the trade-off issues between the timeliness and consistency as a QoS optimization problem, and present an efficient heuristic for QoS management.

4.1 Problem Formulation

We consider our service as one in which the publishers of information establish contracts with us requiring certain guarantees, both in terms of the consistency of information and timeliness of response. In our QoS model, *consistency* is defined as the temporal difference between the original data at the publisher, which represents the real-world state, and the proxy cached copy. We require the publisher to provide us with the minimum (best) and maximum (worst) allowed consistency levels, $Min\ C_i$ and $Max\ C_i$ respectively, e.g. 2 minutes (best) to 10 minutes (worst). For *timeliness*, the response time should not exceed a certain upper bound specified by the publisher, which can estimate the required average response time for its subscribers. Based on the definition of the consistency and timeliness above, we define our rewards (profit) as follows:

$$R_{total} = \sum_{i=1}^{N} \{Timely\ Hits_i\ \times\ QoS_i\} \qquad (1)$$

where N is the total number of cached publisher agents. We define a *timely hit* as one in which we are able to satisfy both the timeliness and consistency guarantees. QoS_i is defined to quantify the current consistency management level:

$$QoS_i = \frac{Max\ C_i - Current\ C_i}{Max\ C_i - Min\ C_i} \qquad (2)$$

Note that a hard timeliness guarantee is not possible, since a few factors such as backbone delay can be beyond our control. We assume that the contractual guarantees with the publisher of the information are established after considering these factors. The advantage of our service is that, we can improve the timeliness of the information service compared to the current de facto standard web infrastructure, in which the subscribers always have to contact the publisher for dynamic content across the Internet backbone.

In the applications we intend to target, we expect the timeliness guarantees are more stringent than the consistency. For example, traffic information should be available in a few seconds, however the data can be a few minutes old. From this observation, we adopt an approach of *periodically polling* the content servers to check the state of the published data in a cost-effective way. Clearly, the *polling period* is dependent on the required consistency of the data. Note that we can not provide hard guarantees on the consistency as well, i.e. we can not state that the information received by the subscribers is never older than $Max\ C_i$ minutes. This is also due to the unpredictable backbone delay between the original content server and active caches. Our QoS model provides zero rewards, if we cannot provide a consistency better than $Max\ C_i$. The main objective of our QoS model is to maximize the rewards and at the same time attempting to satisfy the guarantees. Note that the subscriber received QoS can also increase as the rewards increase. More specifically, response time is considered the most important performance metric and the consistency can be traded off (in a bounded manner) to improve the timeliness. The details of the trade-off techniques are given in Section 4.3.

4.2 Factors Affecting QoS

There are three time-varying factors, which can affect QoS. They are identified below:

- *Internet Backbone delay* is our main source of unpredictability. It is a main factor to determine the *polling period*. Backbone delay consists of the time (S) required to set up (and breakdown) a network connection with the content server, send a HTTP request and receive the reply. It is measured as:

  $$d = S + Bx \qquad (3)$$

  where B is the delay per byte and x the total number of bytes sent/ received.

- *Active cache utilization* mainly depends on CPU utilization needed to dynamically generate web pages by executing the corresponding agents. CPU utilization can increase as more agents are executed in a time period. This can result in CPU utilization of the active cache reaching a certain level at which it can not accept/serve more subscriber requests without violating the required response time.

- *Network bandwidth* is a restricted resource too. To provide better consistency, the content server needs to be polled more frequently. This results in more requests being sent out from the active cache on its network, potentially causing congestion and reducing the available network bandwidth, which in turn results in poorer server response. Hence, care has to be taken to avoid link overload.

Care must also be taken to sensure that active caching does not increase the load on the content server.This can occur if some published data is updated very frequently by the publisher, but rarely accessed by the subscriber. In this case, the original content server load can actually increase by caching due to the excessive polling requests. Without active caching, it can serve the content on demand at possibly lower computational cost. To address this problem, we cache only *hot* agents. An agent is considered *hot*, if its access is more frequent than the polling. Since the access frequency can show a large deviation from one measuring period to another, a moving average is taken to smooth the deviation:

$$SAF_i = a \times SAF_i + (1-a) \times AF_i \tag{4}$$

where $0 \leq a \leq 1$, SAF_i is the smoothed average and AF_i is the current value.

4.3 QoS Management Scheme

In our QoS management scheme we dynamically adjust the *consistency level* considering the factors described in Section 4.2. We describe only the consistency degradation procedure due to the limited space. The upgrade procedure is logically symmetrical to the degradation. A cached agent i would have a polling period P_i based on the currently provided consistency (C_i). In order to provide a consistency of C_i over a network with average delay d,

$$P_i \leq C_i - d \tag{5}$$

Periodic polling can be performed by setting a timer to timeout at constant intervals (t). Thus, P_i is a multiple of t:

$$P_i = \lfloor \left(\frac{C_i - d}{t} \right) \rfloor \times t \tag{6}$$

As C_i is adapted, P_i is also recalculated. At every timeout, the following steps are performed:

First, the current CPU utilization (U) of the active cache is measured. If it is above the threshold (U_{thresh}), then the current consistency C_i of N_u data objects is degraded by a constant K_u (the temporal distance between the original and cached copy could increase):

$$N_u = \lceil \left(\frac{N \times [U - U_{thresh}]}{100 - U_{thresh}} \right) \rceil \tag{7}$$

$$C_{i_{new}} = C_i + K_u \tag{8}$$

where N is the total number of cached publisher agents.

Next, each of the data objects, scheduled to be fetched at that timeout, are fetched from the content server if modified. The backbone network delay d is measured. If the backbone delay has increased compared to the previous timeout, the C_i's will be degraded:

$$C_{i_{new}} = C_i + K_d \times \delta d \tag{9}$$

where K_d is a constant and δd is the variance of the delay.

At each data fetch from the server, the available backbone network bandwidth B is monitored to ensure that the link capacity, specified by the backbone link provider, is not exceeded. The number of outgoing requests must be reduced, if the available bandwidth B falls below a certain threshold B_θ. This can be done by degrading the consistency for N_b data objects by a constant K_b:

$$N_b = \lceil \left(\frac{n_q \times [B_\theta - B]}{B_\theta} \right) \rceil \tag{10}$$

$$C_{i_{new}} = C_i + K_b \tag{11}$$

where n_q is the number of remaining data objects to be fetched at the current timer interrupt.

5 Evaluation

In this section, we present experimental results to show the average latency improvement due to the proposed active caching. We also show the validity of the timeliness-consistency tradeoff by showing that the consistency is maintained according to the QoS specification.

An automatic URL generator (SURGE [18]) is extended to generate requests for dynamic content provided by the publisher agents. Over 100,000 requests for 2000 publisher agents are generated in approximately 10 minutes. The testbed consists of a *content server*, an *active cache* and multiple clients. The clients and the cache are considered to be on a local network. Internet backbone delays are simulated between the content server and active cache. We used 700MHz Pentium IIIs with 256MB RAM running Linux 2.2.12 for both the *content server* and the *active cache* and 700MHz Pentium IIIs with 64MB RAM running Linux 2.2.12 as the *clients*. The clients and active cache are connected by a 100Mbps Ethernet link. Although we use one server, it is modeled as if the agents are being fetched from different servers on the Internet for a realistic experiment. This is done by introducing random delays in agent fetches. We introduce two sets of delay: (25ms to 75 ms) to model the nominal Internet backbone delay for local area and another set of delays (200ms to 250ms) for wide area. Also, we classify data into two classes based on the strictness of the consistency requirements: one with relatively strict consistency requirements (60s, 120s), and another with more flexible consistency requirements (60s, 420s).

To show the performance due to caching, two sets of performance evaluation are performed. In the first set, the client requests are sent directly to the server, and in the other the client requests are directed to the active cache. Fig. 3 compares the average user-perceived latencies. On average, we find about 100-150% reduction in latency. The latency for the 'no-cache' case is the sum of the modeled Internet

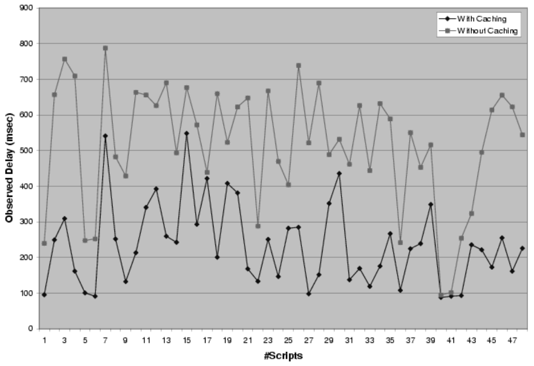

Figure 3. Average Latency Improvement

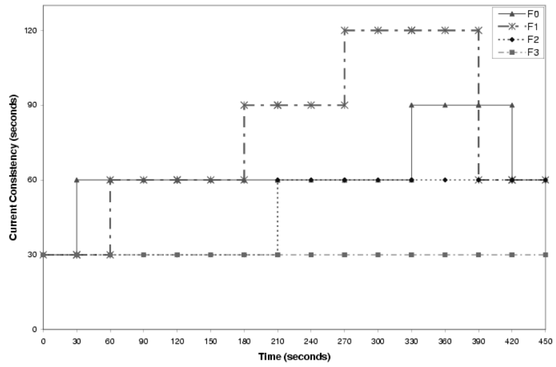

Figure 5. Consistency Tradeoff

backbone delay and the agent execution time. In the actual Internet environment, we could observe a greater difference in latencies as the network is more congested.

Fig. 4 demonstrates the user-perceived latency for a single agent, with and without the cache. As seen in the figure, the latency is limited below a few tens of milliseconds for the most part, when the requests are served from the cache. The initial peak delay occurs on the first access to the agent, when it (and all the data that it uses) need to be fetched from the content server. A few peaks correspond to the points in time when the agents are executed on the cache, however, active caching service shows the latency improvement in every case and the improvement can increase under the Internet congestion as discussed before.

Finally, fig. 5 shows how the consistency is traded off for timeliness as the bottleneck factors (Section 4.2) get worse. The figure shows the current consistency provided to four representative data files. Table 1 describes the consistency requirements and the corresponding update periods on the content server for the data files.

From the figure, we can observe that every file is initially polled at the period P_i=30s to provide $Min\ C_i$ of 60s. As the backbone delay increases, the files are degraded in the increasing order of the update frequency i.e. $F_0 < F_1 < F_2$. F_3 is never degraded since it the content server never updates it. Since $Max\ C_i$ for F_0 is 120s, its consistency is not degraded more than 90s, whereas the consistency for F_1 is degraded up to 120s. From this, we can see that the QoS in terms of consistency is differentiated between the two classes with relatively strict and flexible consistency requirements.

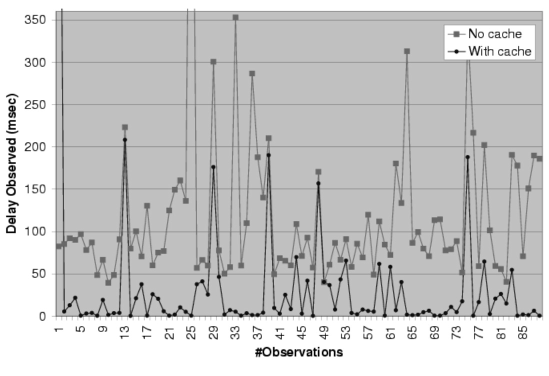

Figure 4. Response Time for a Single Agent

6 Conclusions and Future Work

Our proposed content distribution architecture provides the basis for a global industrial information utility built with existing web infrastructure. We are able to support soft real time monitoring of the information publishers, while at the same time providing bounded freshness guarantees on the information provided to the subscribers. We are able to do this in a scalable fashion by migrating the need for computational power from the servers to proxy caches, while at the same time reducing backbone network traffic. Despite the complexity of the problem, we show that it can be mapped to a QoS optimization problem and provide an efficient heuristic for QoS management resulting in a significant performance improvement.

Currently, we only support read-only publishing agents. Subscribers are not allowed to modify any state on the publisher. We are investigating methods to allow write access in a timely manner, thereby enabling control operations in addition to monitoring. Security/authentication are also important research issues to be addressed in the future.

Table 1. Data Characteristics

Name	$Min\ C_i$	$Max\ C_i$	Update Period
F_0	60	120	15 seconds
F_1	60	420	60 seconds
F_2	60	120	240 seconds
F_3	60	420	∞ (never updated)

References

[1] P. Cao and S. Irani, "Cost-aware www proxy caching algorithms," in *Proc. of the USENIX Symp. on Internet Technologies and Systems*, Dec. 1997, pp. 193–206.

[2] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell, "A hierarchical internet object cache," Tech. Report 95-611, Computer Science Dept., Univ. of Southern California, Los Angeles, California, Mar. 1995.

[3] J. Dilley, M. Arlitt, and S. Perret, "Enhancement and validation of the squid cache replacement policy," in *4th International Web Caching Workshop*, San Diego, California, Mar 1999.

[4] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: A scalable wide-area web caching sharing protocol," in *SIGCOMM*, 1998.

[5] R. Caceres, F. Douglis, A. Feldman, and M. Rabinovich, "Web proxy caching: The devil is in the details," in *Proc. of the Workshop on Internet Server Performance*, Madison, WI, Jun 1998.

[6] P. Cao, J. Zhang, and K. Beach, "Active cache: Caching dynamic contents on the web," in *Proc. of Middleware '98*, 1998, pp. 373–388.

[7] V. Holmedahl, B. Smith, and T. Yang, "Co-operative caching of dynamic content on a distributed web server," in *Proc. oof Seventh IEEE International Symp. on High Performance Distributed Computing*, 1998, pp. 243–250.

[8] A. Iyengar and J. Challenger, "Improving web server performance by caching dynamic data," in *Proc. of USENIX Symp. on Internet Technologies and Systems*, Dec 1997, pp. 49–60.

[9] F. Douglis, A. Haro, and M. Rabinovich, "Hpp: Html macro-preprocessing to support dynamic document caching," in *Proc. of USENIX Symp. on Internet Technologies and Systems*, Dec 1997, pp. 83–94.

[10] B. Smith, A. Acharya, T. Yang, and H. Zhu, "Exploiting result equivalence in caching dynamic web content," in *Proc. of Second USENIX Symp. on Internet Technologies and Systems*, Oct 1999, pp. 209–220.

[11] J. Challenger, P. Dantzig, and A. Iyengar, "A scalable system for consistently caching dynamic web data," in *Proc. of IEEE INFOCOM'99*, Mar. 1999.

[12] H. Zhu and T. Yang, "Class-based cache management for dynamic web contents," in *IEEE INFOCOM*, 2001.

[13] K. Rajamani and A. Cox, "A simple and effective caching scheme for dynamic content," Tech. Report TR 00-371, Computer Sc. Dept. at Rice Univ., 2000.

[14] A. C. Weaver, J. Luo, and X. Zhang, "Monitoring and Control Using the Internet and Java," in *IECON99*, 1999.

[15] M. Furuya, T. Fukomoto, and T. Sekozawa, "WWW-Browser-Based Monitoring System for Industrial Plants," in *IECON99*, 1999, vol. 3, pp. 1146–1151.

[16] Squid Web Proxy Cache, "Squid-2.3," http://www.-squid-cache.org/.

[17] The Apache Software Foundation, "Apache-1.3.9," http://www.apache.org/.

[18] P. Barford and M.E. Crovella, "Generating representative web workloads for network and server performance evaluation," in *Performance '98/ACM SIGMETRICS '98*, 1998.