Basis Token Consistency: Supporting Strong Web Cache Consistency

Adam D. Bradley and Azer Bestavros
Computer Science Department, Boston University
111 Cummington Street
Boston, MA 02215

***Abstract*- With web caching and cache-related services like CDNs and edge services playing an increasingly significant role in the modern Internet, the problem of the weak consistency and coherence provisions in current web protocols is drawing increasing attention. Toward this end, we differentiate definitions of consistency and coherence for web-like caching environments, and then present a novel web protocol we call "Basis Token Consistency" (BTC). This protocol allows compliant caches to guarantee strong consistency of content retrieved from supporting servers. We then compare the performance of BTC with the traditional TTL (Time To Live) algorithm under a range of synthetic workloads in order to illustrate its qualitative performance properties.**

I. INTRODUCTION

For many years it has been asserted that one of the keys to a more efficient and performant web is effective reuse of content stored away from origin servers. This has taken a number of forms: Basic caching, varieties of prefetching, and more recently, Content Distribution Networks (CDNs). What has become increasingly clear in recent years is that the traditional target of research, poor eviction and replacement algorithms, is not in fact a serious obstacle to "good" use of a caching infrastructure [1].

In the current web, many cache eviction events and uncacahable resources are driven by two server application goals: First, providing clients with a *recent* view of the state of the application (*i.e.*, information that is not too old); Second, providing clients with a *consistent* view of the application's state as it changes (*i.e.*, the client's perception of changes to server state should be non-decreasing in time). The current web protocol, HTTP/1.1 [2], addresses the first goal by way of an expiry mechanism and the second only in a few very tightly constrained ways; unfortunately, the latter mechanisms are not general enough for needs of non-trivial dynamic or interactive web applications.

In this paper we propose Basis Token Consistency (BTC), a backwards-compatible and transparently interoperable extension to the HTTP protocol which enables caches to maintain a completely consistent view of the server without requiring out-of-band communications or per-client server state. We then present simulations which compare the performance and behaviors of BTC with those of the expiry-based weak consistency model.

II. CONSISTENCY AND COHERENCE WITHIN A WEB-LIKE FRAMEWORK

Much of the current body of web cache consistency literature focuses upon a model of consistency drawn from distributed filesystem work; namely, consistency between a single object at the origin server and a cached copy of that same object on the network. While this model has its uses, the web is a much more complicated system than a distributed filesystem; particularly, the relationships among multiple objects provided by a single server are akin to views of a distributed database. As such, for the web we propose definitions of consistency and coherence more in keeping with those used in distributed database research.

A. Consistency

For our purposes, cache *consistency* refers to a property of the entities served by a single logical cache, such that no response served from the cache will reflect an older state of the server than that reflected by previously served responses. Another way of stating this is that a consistent cache provides a *non-decreasing view* of data the server uses to construct its output; informally, once you have seen the result of some event having happened, you should never see anything which contradicts that. This is the definition used in [3].

This definition is a special case of *view consistency* [4], in which a cache may provide different responses to different clients in order to optimize some application goal (such as maximizing client cache utilization), just so long as each client sees an internally consistent (non-decreasing) response stream. Our definition is a special case in that the consistency of the aggregated response stream implies that any subset of that stream will also be consistent.

Notice that this definition is completely independent of *recency*, and of "consistency" between two different caches' copies of the same entity. We define these as *coherence* properties.

B. Coherence

We define a *cache coherence protocol* for the web as a means for making updates to entities propagate through the caching network such that all clients interested in entities affected by those updates eventually see their results; the word "eventually" is given meaning by the details of the coherence protocol itself.

There are two coherence models used in the current web. The first is "immediate coherence" in which caches are forbidden from returning a response other than that which would be returned were the origin server contacted; this guarantees semantic transparency, and as a side-effect also guarantees a consistent view of the server's state.[1] While the current web can only provide this level of coherence by pre-expiring all entities (forcing all caches to re-validate with the origin server on every request), a number of proposed coherence extensions

This research was supported in part by NSF (awards ANI-9986397 and ANI-0095988) and U.S. Department of Education (GAANN Fellowship).

[1] *I.e.*, any given cache's copy of an entity is only usable if it is "consistent" with the server's copy, hence the widespread use of "consistency" to mean "immediate coherence."

use server-originated invalidation methods [5], [6], [7], [8], [9] to proactively notify caches when content is modified. Unfortunately, these messages must generally be sent either via an out-of-band channel (not part of regular HTTP transactions, which poses difficulties in the presence of non-implementing intermediaries or of asymmetric-reachability networks) or a mixed channel (invalidations are attached to response which they may be unrelated to, which raises problems when intermediary proxies do not understand the protocol).

The second model is "bounded staleness"; this is accomplished by expiry mechanisms in the current HTTP protocol which limit how old a cached response can become before it must be validated with the server, guaranteeing that no single cached entity will ever be more than some known timespan out-of-date.

Several proposed mechanisms combine aspects of the above two techniques with a *lease* mechanism to provide a bounded-in-time relaxation of immediate coherence over a finite timeframe without requiring caches to periodically validate their contents. This model is known as Δ-consistency [8], [9], [10].

Coherence is not addressed further in this paper; we believe that a reasonable expiry policy or any of the invalidation-driven models can act as an excellent complement to our proposed consistency mechanism.

III. Basis Token Consistency

We have devised a caching extension to HTTP we call "Basis Token Consistency" (BTC) with the following properties: (a) Strong point-to-point consistency is supported without relying upon intermediary cooperation. (b) No per-client state is required at the server or proxy caches. (c) Invalidations are naturally aggregated in semantically meaningful ways. (d) Invalidation is driven by web applications, not heuristics. (e) The necessary data is transmitted only in related responses, hence out-of-band and mixed-channel messages are not required.

A. Conceptual Overview of BTC

The BTC protocol and algorithms are based upon the concept of a logical vector clock [11], [12]. Each server maintains a logical vector clock, where each element represents a data source ("origin datum") used by the server's application logic; each response is annotated with a list of the elements used to construct it (`cctokens`) and their current logical clock values (`ccgenerations`). Whenever an origin datum is updated, its clock value is incremented; it is therefore trivial to determine if two responses could have co-existed in time or if one necessarily obsoletes the other by comparing the generation numbers of elements appearing in both responses.

This information is provided by the server using the `Cache-Consistent` entity header; a simplified[2] grammar is presented in Figure 1. Each origin datum is represented by an opaque string (`cctoken`), and its clock value is represented in hexadecimal (`ccgeneration`). For example:

`Cache-Consistent:db1row;4e9, db2row@bu.edu;7a`

Caches implementing BTC index their entries on the opaque token strings. Whenever a new entity arrives, each of its tokens' generation numbers are checked against the cache's "current" generation numbers for the same tokens. If they match, no further action is taken. If the newly seen generation number is greater, all entities affiliated with the older generation of that token are marked as invalid while the "current" generation number is updated to the new value. If the newly seen generation number is less than the current value, then the response itself is stale and inconsistent (perhaps produced by an inconsistent cache upstream), so the request should be repeated using the end-to-end reload mechanism.

[2] The complete grammar can be found in [13]; it includes several productions corresponding to an extension not presented in this paper.

```
CacheConsistent =
      ``Cache-Consistent'' ``:''
      #cctokengeneration
cctokengeneration =
      cctoken
      ``;'' ccgeneration
cctoken = cctokenid [ cctokenscope ]
cctokenscope = ``@'' host
cctokenid = token
ccgeneration = 1*HEX
```

Fig. 1. The `Cache-Consistent` HTTP Entity Header

Tokens can be scoped to particular DNS domains in a manner similar to cookies; this allows data sources to be shared among multiple hosts within a domain. If no scope string is specified, it defaults to the value of the `Host` header provided by the client. This string is part of the token for matching purposes; this is done as a security measure [14].

The vector clock is a powerful construct, and the simple algorithm and protocol presented here can be elaborated upon in a number of ways; for example, we can parametrically relax generation number matching to a range, affording a controllably less stringent consistency model [15] which lazily approximates Δ-consistency in logical time. This and several other practical extensions to BTC are discussed in [13].

B. Requirements for BTC

Unlike other approaches, BTC will not work effectively without support from the applications behind the web server. The basis tokens essentially offer a "window" into the state of databases, files, and other resources which those applications normally insulate from the outside world. This requires that web service applications be engineered with support for this feature in mind; the complexity and cost of doing so may vary greatly with the structure, data model, and data access methods of the application.

BTC is highly scalable in the sense that servers need not maintain any per-client state. However, it does require that each cache store and index upon what may be arbitrarily many basis tokens. While we expect basis tokens to be short strings (on the same order as common URIs), the number of tokens required to support a given working set of pages can vary greatly with the structure of the backing server applications; as such, it would not be unreasonable for heuristic per-resource and per-server limits to be set.

IV. Validation

To illustrate the qualitative performance effects of BTC, we implemented a simple server-and-cache simulation which compares the performance and correctness characteristics of multiple consistency models under a range of workloads and parameters.

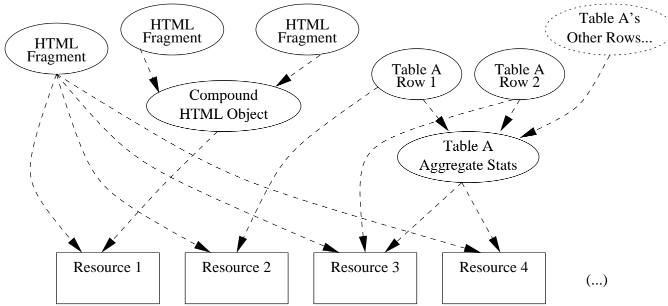

Fig. 2. Sample Object Dependence Graph (ODG)

As BTC algorithm's behavior is driven by events within the server's application logic (which provoke document changes), a document update model [16] is not sufficient; a meaningful simulation requires a meaningful model for the application. A particularly interesting and useful application to model is a modern Content Management System (CMS); we base our CMS model upon the DUP-based system [17], [3].

One of a CMS's basic jobs is to assemble *fragments* to produce complete responses. The relationships among and between fragments and resources are codified in an *object dependence graph* (ODG), a directed graph with nodes representing origin data, edges representing access to data, and other nodes representing resources and intermediary fragments. A simplified sample ODG is presented in Figure 2.

This graph provides us with all the interdependence information needed to address consistency; the most straightforward way to employ the ODG for BTC is to represent nodes of the graph with basis tokens. Thus, implementing BTC in an ODG-based CMS should be a relatively straightforward programming exercise; all that need be added are monotonically increasing generation numbers for each node, a persistent mapping from nodes to token strings, and the code to construct the appropriate `Cache-Consistent` headers from these values.

A. Simulation Design

Lacking any thorough study of ODGs found in the wild[3], our model incorporated a number of simplifications. Rather than claim our simulations are representative, we included a variety of parameters that allow us to explore our protocol's performance under a wide range of potential conditions. The set of results presented here explicitly are illustrative of the qualitative performance properties we observed under a wide range of parameterizations.

We modeled our simple CMS using a bipartite graph of datum nodes and resource nodes, built with two parameters: size and saturation. Size could be 40, 200, and 400 resources and 200, 1000, and 2000 datum nodes, respectively; saturation (the percentage of possible edges in the graph present) was independently set to 12.5%, 25%, and 50%. Each datum node can then be assigned a parameterized update process (periodic, exponential, pareto, normal). The resource nodes are assigned popularities according to a Zipf-like distribution.

In this paper we focus upon results for "small and dense" ODGs (40 resources, 200 datum nodes, 50% saturation) with exponential update processes whose means are themselves exponentially distributed. We do not model locality or popularity among origin data; while unfortunate, this simplification is motivated by the relative lack of topological studies of ODGs. We report on experiments where resource popularities are found using a Zipf parameter of 0.7, which approximates the current web [18].

For each simulation, the model produces a list of some number (5000 for small graphs) of update events timestamped according to their update processes. A list of requests with constant inter-arrival times is also synthesized, and merged with the stream of update events. The number of requests is a multiple of the number of updates: 1, 20, or 400, labeled *slow*, *medium*, and *fast*, respectively. While we would like to have modeled request arrivals more precisely [19], the rather *ad hoc* way in which the update process is constructed suggests that the marginal value of such detail would be very limited for these experiments.

Finally, this combined event list is fed to the server-cache simulator. This simulator outputs a number of cache performance metrics (discussed below) for a set of simple expiry caches (each with its own fixed TTL value) and a set of "Hybrid" caches which use both BTC and expiry driven by the same TTL values. (Of course, a single TTL value across all documents is clearly not reflective of a well-designed expiry policy; again, our goal is for these experiments to be simple and illustrative, not representative.) TTL values are normalized to the length of the event stream; a value of 1.0 means that a document fetched at the beginning of the simulation will not expire for the length of the simulation. The "pure" BTC behavior is illustrated by the Hybrid case with a TTL of 1.0.

B. Simulation Results

Graphs present the time-to-live parameter on the X axis. The Y axis is normalized to the total number of requests made in a simulation run.

Figure 3 shows the results for the small-and-dense simulation with a slow request stream. This could reflect, for example, a highly dynamic server interacting with a single user or small-population shared cache. The figure shows three parameters for each cache control algorithm: fresh responses (how many cached responses were the same as would have been provided by the server at that same point in time), response quality (a continuous variant of freshness, indicating how many of the origin data used to produce a page have not been updated at the time the cache serves it), and server load (how many requests were not served by the cache).

Notice that the TTL algorithm sheds significant server load for moderate time-to-live values, but this is accompanied by a matching falloff in the number of fresh responses; this is indicative of the large number of "false hits" as TTLs exceed the very short response freshness lifespans. The accumulation of poor quality (poor immediacy) is less dramatic; quality seems to follow its load shedding and fresh response curves at a multiplicative TTL offset. This makes intuitive sense, as it reflects the ongoing and continuous (analog v. binary) accumulation of single events, each causing a small fraction of the cached response to become stale.

At the same time, note that the Hybrid algorithm only allows about 15% of the server's load to be shed. However, its response quality remains extremely high, and the number of stale responses is held to about 10%. This is not surprising; more

[3]The observations presented in [3] are certainly interesting and illustrative, but not necessarily representative.

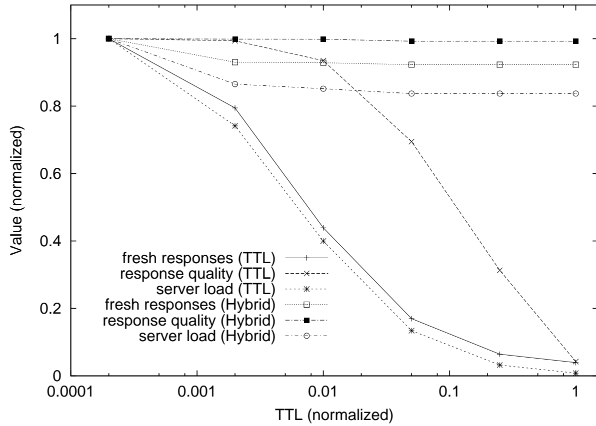

Fig. 3. Freshness, Quality, and Load - Slow Request Rate

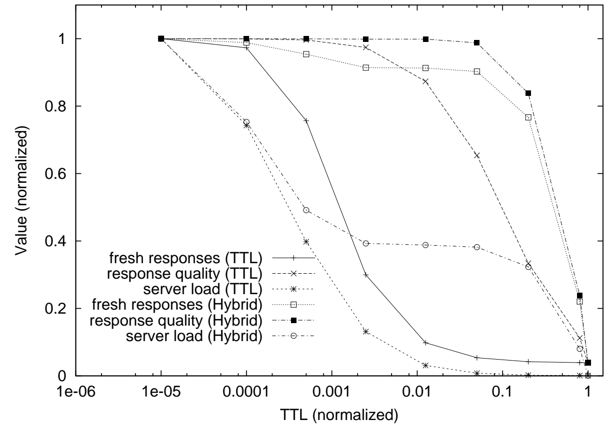

Fig. 5. Freshness, Quality, and Load - Medium Request Rate

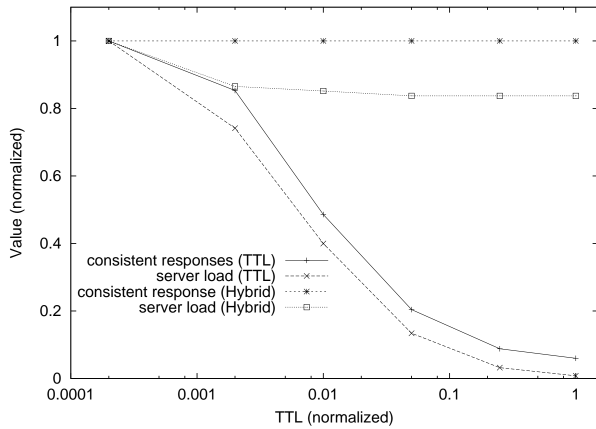

Fig. 4. Consistency and Load - Slow Request Rate

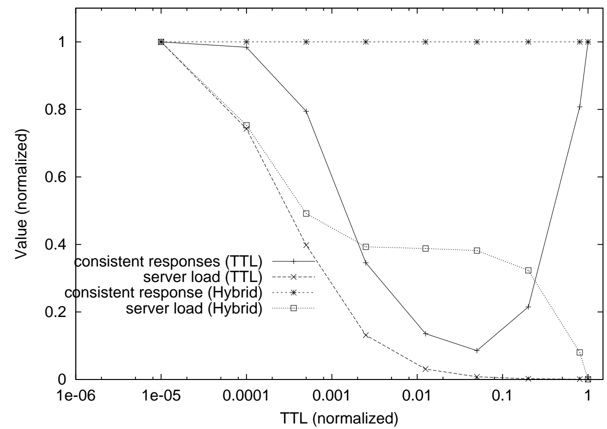

Fig. 6. Consistency and Load - Medium Request Rate

resources are updated in the average unit of time than requests are made, so it is likely that many requests are for resources that are consistency-related to already-cached responses which are then immediately obsoleted by each new response.

Figure 4 illustrates how, under the same experiments, BTC's limited reduction of server load relates to our design goal of strong consistency where TTL fails. The "consistent responses" value indicates the number of responses that do no reflect any older versions of origin data than have already been seen by the cache (*i.e.*, how many responses were "non-decreasing"); notice how server load and consistency decline in parallel for large TTLs under the TTL algorithm, while the Hybrid algorithm maintains consistency and more gradually reduces server load.

The small-and-dense setup under a medium request rate exhibits some very interesting behaviors and contrasts, as shown in figures 5 and 6. Notice particularly how, for smaller TTL values, the Hybrid algorithm sheds load almost as quickly as TTL, and levels off at a 60% cache hit rate (40% server load) over several orders of magnitude, maintaining in parallel a very high fresh response value (about 90%) while TTL's fresh response count quickly declines as load shedding increases.

TTL's quality value seems to follow its load shedding and fresh response curves at a multiplicative TTL offset; this is the same effect noted above under the slow request rate.

Quality and fresh responses for the Hybrid algorithm both deteriorate quickly under very large TTLs. This makes intuitive sense in light of Figure 6; notice how TTL's number of consistent responses actually increases for very large TTL values. This happens because, when requests arrive fast enough, the cache can become populated with a long-lived and self-consistent *snapshot* of the server's state. Under Hybrid with long lifetimes, this is exactly what happens; the cache quickly acquires a snapshot at the beginning of the simulation run, and because all the responses making up that snapshot are long-lived, it stops talking with the server and therefore stops receiving the (lazily delivered) invalidation-provoking tokens. This property for plain TTL caches across the different request rates is illustrated in Figure 7, which shows the internal consistency of TTL caches' responses at the slow, medium, and fast request rates; as the request rate increases with respect to the update rate, caches (whether TTL or BTC) are more frequently able to acquire large internally consistent snapshots of server state, significantly reducing server load but sacrificing recency. Under a high request rate this effect is amplified, but other graphs describing behavior under those conditions provide little additional insight.

It is under these higher request rate conditions that the in-

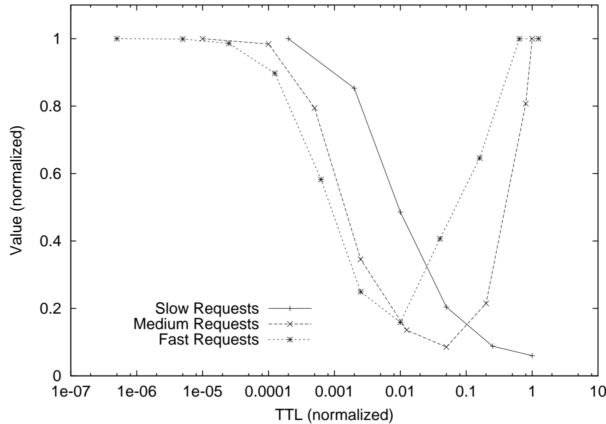

Fig. 7. TTL Response Consistency under Various Request Rates

teraction between the number of resources, the Zipf parameter, and the request rate becomes significant to the performance of the BTC algorithm; for example, it is hard to get a complete snapshot when the number of resources is particularly large relative to the request rate, or when the Zipf parameter is particularly large; at the same time, large Zipf parameters make it less likely that those rarely-accessed (and thus potential "snapshot breaking") resources will actually be requested, making it difficult for strong consistency alone to drive cache content recency.

V. Conclusion

We have described a novel protocol, Basis Token Consistency (BTC), which provides strong consistency via lazy notification to any participating cache regardless of the presence and participation of intermediaries. We then presented results from a simple simulation of a modern Content Management System (CMS) driving a set of TTL and BTC caches and compared their behaviors under a range of parameters, illustrating some of the tradeoffs and effects of each in terms of their ability to shed server load and quantitative measures of the "correctness" of the response stream delivered by each.

While BTC requires the explicit cooperation of server applications and a potential moderate increase in cache state, we believe its low implementation complexity for caches, its interoperability with the current infrastructure, and its guaranteed properties make it a desirable extension to deploy in the present-day web infrastructure.

Acknowledgments

The authors wish to thank Assaf Kfoury and the anonymous reviewers for their helpful comments on this paper.

References

[1] R. Caceres, F. Douglis, A. Feldman, G. Glass, and M. Rabinovich, "Web proxy caching: The devil is in the details," in *ACM SIGMETRICS Performance Evaluation Review*, Dec. 1998.

[2] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol – HTTP/1.1." RFC2616, 1999.

[3] J. Challenger, A. Iyengar, K. Witting, C. Ferstat, and P. Reed, "A publishing system for efficiently creating dynamic web content," in *INFOCOM (2)*, pp. 844–853, 2000.

[4] A. Goel, "View consistency for optimistic replication," Master's thesis, University of California, Los Angeles, Febrruary 1996. Available as UCLA Technical Report CSD-960011.

[5] P. Cao and C. Lui, "Maintaining strong cache consistency in the world-wide web," in *ICDCS*, 1997.

[6] B. Krishnamurthy and C. Wills, "Piggyback server invalidation for proxy cache coherency," in *Proceedings of the WWW-7 Conference*, (Brisbane, Australia), pp. 185–194, Apr. 1998.

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

[8] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar, "Engineering server-driven consistency for large scale dynamic web services," in *WWW10*, (Hong Kong), May 1-5, 2001.

[9] A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, and R. Tewari, "Cooperative leases: Scalable consistency maintenance in content distribution networks," in *WWW2002*, (Honolulu, Hawaii), May 2002.

[10] R. Tewari, T. Niranajan, and S. Ramamurthy, "WCDP 2.0: Web content distribution protocol," Feb. 2002. Internet Draft (work in progress) draft-tewardi-webi-wcdp-00.txt.

[11] C. Fidge, "Logical time in distributed computing systems," *Computer*, vol. 24, pp. 28–33, Aug. 1991.

[12] F. Mattern, "Virtual time and global states of distributed systems," in *Proc. Parallel and Distributed Algorithms Conf.*, pp. 215–226, 1988.

[13] A. D. Bradley and A. Bestavros, "Basis token consistency: Extending and evaluating a novel web consistency algorithm," in *Workshop on Caching, Coherence, and Consistency (WC3)*, (New York), June 2002.

[14] A. D. Bradley and A. Bestavros, "Basis token consistency: A practical mechanism for strong web cache consistency," Tech. Rep. BUCS-TR-2001-024, Boston University Computer Science, 2001.

[15] H. Yu and A. Vahdat, "Design and evaluation of a continuous consistency model for replicated services," in *Proceedings of Operating Systems Design and Implementation (OSDI)*, Oct. 2000.

[16] M. Reddy and G. P. Fletcher, "Intelligent web caching using document life histories: A comparison with existing cache management techniques," in *3rd International WWW Caching Workshop*, (Manchester, England), June 1998.

[17] A. Iyengar and J. Challenger, "Data update propogation: A method for determining how changes to underlying data affect cached objects on the web," Tech. Rep. RC 21093(94368), IBM T. J. Watson Research Center, 1998.

[18] P. Barford, A. Bestavros, A. Bradley, and M. Crovella, "Changes in web client access patterns : Characteristics and caching implications," *World Wide Web*, vol. 2, pp. 15–28, 1999.

[19] P. Barford and M. Crovella, "Generating representative web workloads for network and server performance evaluation," in *ACM SIGMETRICS*, 1998.